MODELING THE BEHAVIOR OF A FALLING OBJECT

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Mechanical Engineering

By

Michael J. Chen

2019

SIGNATURE PAGE

THESIS:	MODELING THE BEHAVIOR OF A FALLING OBJECT
AUTHOR:	Michael J. Chen
DATE SUBMITTED:	Spring 2019
	Department of Mechanical Engineering

Dr. Campbell A. Dinsmore _____
Thesis Committee Chair
Mechanical Engineering

Dr. Behnam Bahr _____
Mechanical Engineering

Dr. John P. Caffrey _____
Mechanical Engineering

ACKNOWLEDGEMENTS

I would like to thank my parents for their support and foresight, my professors for their knowledge and experience, and Zhen for his assistance and friendship.

ABSTRACT

It is often of interest to stabilize and reorient the attitude of falling objects. The approach of interest in this study was the utilization of aerodynamic control surfaces. However, the parameters of such a system were not well-documented. Therefore, a physical vehicle was created in order to capture the behavior of this type of system. The vehicle was modeled, fabricated, and programmed for data collection. Launch and retrieval apparatuses were fabricated for safety. Freefall data was collected for various initial conditions. The data was then processed into a common reference frame. A method of calculating the full inertia tensor of an object was developed for proper modeling. The resulting experimental data was processed for analysis and a preliminary dynamic model was created.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

The behavior of falling objects has long had significance in the scientific community as well as in daily life. A thrown ball, a dropped plate, and a falling skydiver are just some examples. In many cases, it is beneficial to stabilize motion imparted on the falling object. Whereas it is possible for an object to stabilize its orientation passively, this may require more time than is available for the object to re-orient itself. This is especially true for cats, who are able to right themselves in mid-air in order to minimize injury from falls. Techniques such as gyroscopic stabilization and parachutes are often used, but these methods also suffer from their own respective limitations such as mechanical complexity and weight. An alternative form of attitude stabilization could be created through the use of aerodynamic control surfaces. This could be applied to a vehicle in the shape of a cube. However, the behavior of this kind of vehicle in freefall is not well documented. Therefore, the purpose of this study is to capture the behavior of this vehicle. This information could then be used in the development of a control law.

The drag force on a body depends on many factors involving both the fluid and the body. The geometry, velocity, and surface material of the body play a role as well as the density and viscosity of the fluid, as elaborated in Appendix B [6]. For an object such as a cube, the aerodynamic forces and therefore the cube's physical behavior has significant dependence on the cube's orientation. The cube is not likely to have only a single face pointing toward the ground, resulting in three faces of the cube facing incoming air at any one time. The forces generated by the three faces induce moments on the cube, rotating it in space. However, this behavior is not necessarily reflected in the real world. Therefore, as a part of this effort, a physical vehicle was created to obtain its

physical behavior as it experienced freefall. Launching and retrieval apparatuses were also created in order to safely drop this vehicle and collect it after data logging. Modeling of this system required knowledge of its mass properties, namely its inertia tensor. A method of obtaining this tensor using the bifilar torsional pendulum method was investigated and utilized. A rotational model was created to capture the behavior of the vehicle provided the moments applied to the vehicle. A preliminary force model was created to ascertain these moments.

PROJECT PLAN

As indicated earlier, a dynamic model was necessary before any eventual control law could be developed. In addition, the associated physical properties necessary to develop the model had to be determined. This was the focus of the first phase for this project. Experimental work would be needed to properly quantify these physical properties. To this end, a physical vehicle would be created for experimental data collection. The vehicle would be in the general shape of a box with control surfaces mounted on four of its faces. In addition to fabricating the vehicle, apparatuses would have to be fabricated in order to safely repeat data collection efforts. Once the data was captured, work would have to be done in order to interpret it in a meaningful way. The vehicle's inertia tensor is also necessary for modeling of the vehicle behavior. Generally, the mass moment of inertia about an axis can be found experimentally. However, this leaves the products of inertia as unknowns. Therefore, a way to obtain these products and complete the inertia tensor will have to be investigated. The model of the vehicle behavior can be split up into two parts: A model describing the effects of forces on the vehicle orientation and a model to capture those forces in governing equations. Once created, the vehicle model can then be compared to experimental data and a control law could be developed.

Clearly, developing a fully-formed vehicle would be difficult to attempt initially. Therefore, an initial plan would involve developing a prototype in order to identify possible difficulties that might be encountered when designing the vehicle and avoid these issues in this vehicle.

FABRICATION AND PROGRAMMING

Initial Calculations

A reference drop height of 5 stories was chosen for this design (dropped from rest). At an average story height of 10 feet, the following formulas were used to estimate the maximum velocity experienced by the self-stabilizer under these conditions.

$$h = v_0 t - \frac{1}{2} g t^2$$

$$v = v_0 - gt$$

A maximum velocity of 17.28 m/s was liberally estimated with a free-fall time of 1.76 seconds. The maximum drag force on one flap was approximated using the drag equation:

$$F_D = \frac{1}{2} \rho C_D A_p v^2$$

The maximum drag force experienced by one flap was determined to be 1.81 N for the estimated conditions. Assuming a centroid distance of 0.75 inches (1.905cm) from the pivot point, the applied torque was determined to be 0.351 kgf-cm. This torque assisted in the selection of the servo motors used to actuate the flaps in the prototype.

Prototype

A prototype was first created in order to test the components of the vehicle. The Emax ES9258 tail rotor servo was chosen to be the actuator for each flap. With a stall torque of 2.5 kgf-cm and a speed of 0.08s/60°, this servo was more than capable of handling the forces and speed estimated. The prototype was constructed of cardboard panels and measured approximately 6”x6”x6” as shown below.

Figure 1. Prototype

The actuation of the prototype’s flaps were inspired by model aircraft. This actuation involves connecting the flap and servo through a pushrod and will be detailed later. The pushrod used was music wire commonly found at hardware stores. The power source used was 6 AA batteries in a 9V case as seen in the below figure. This was necessary to supply power to the Arduino Uno control board that controlled the flaps.

Four servos were mounted onto the interior sides of the prototype. The prototype was assembled with duct tape and hot glue for easy access and testing. The BNO055 9-axis absolute orientation sensor was chosen for this project for its integration of accelerometer, gyroscope, and magnetometer data to provide orientations in space. The sensor (an Inertial Measurement Unit) was soldered onto a data collection shield for the Arduino. The purpose of the shield was to record testing data to an SD card as a means of data acquisition.

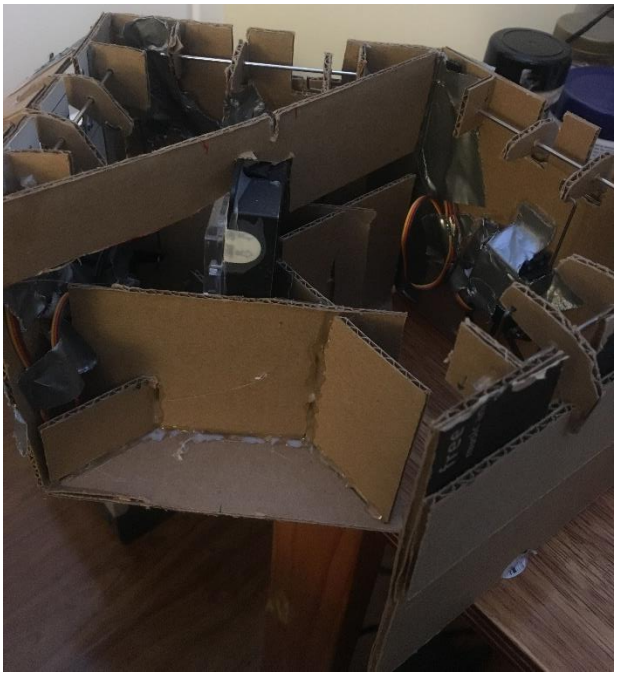

Figure 2. Interior of Prototype

The prototype was used to test the actuation of the flap mechanism as well as the data acquisition. Both of these tests were satisfactory individually. However, it was found that the Arduino could not actuate all four servos and record data simultaneously without error. To remedy this, a separate microcontroller was obtained to act as a servo controller. The servos required a separate power supply from the Arduino microcontroller for proper function. There was little room to add this to the prototype as much of the space was

consumed by the battery pack and structural supports. Therefore, a redesign using Solidworks was implemented to address these issues.

CAD Model

A CAD model of the vehicle was drafted in Solidworks. Initially, 1/8” plywood was chosen as the structural material for its durability and ready availability. Foam was also considered, but 1/8” balsa wood was decided upon for its lightness while maintaining some level of durability. The model was enlarged from 6”x6”x6” to 8”x8”x8” in order to accommodate two sources of power as shown below.

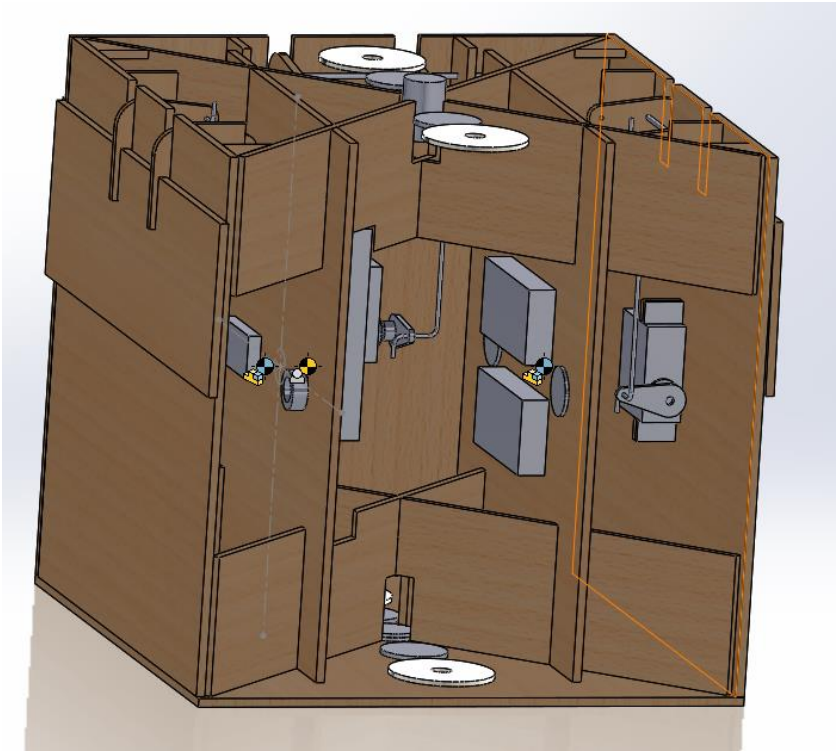

Figure 3. Interior of Vehicle

Two 2S Lithium Polymer batteries were chosen to replace the AA power supply for their compactness and light weight. However, these batteries supplied power at 7.4V, which was not compatible with the 5V servos and boards. Therefore, UBECs (Universal Battery Elimination Circuits), also known as buck converters, were utilized in order to supply appropriate voltages as seen in the below figure.

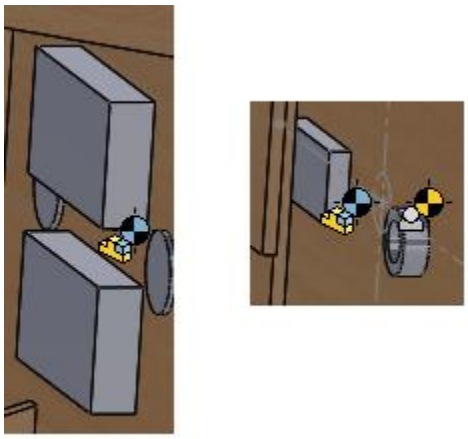

Figure 4. Lipo Batteries and Buck Converter

As stated previously, each flap was actuated by the rotation of its respective servo through the pushrod. At its default state, the flap was near-flush to the panel of the self-stabilizer and the servo horn was near-perpendicular. The flap was not allowed to freely move because of its connection to the servo horn. When actuated, the servo horn rotated away from the flap. This pulled the rocker arm down and rotated the flap about its pushrod. Thus, the orientation of the flap was controlled through the orientation of the servo horn as seen below.

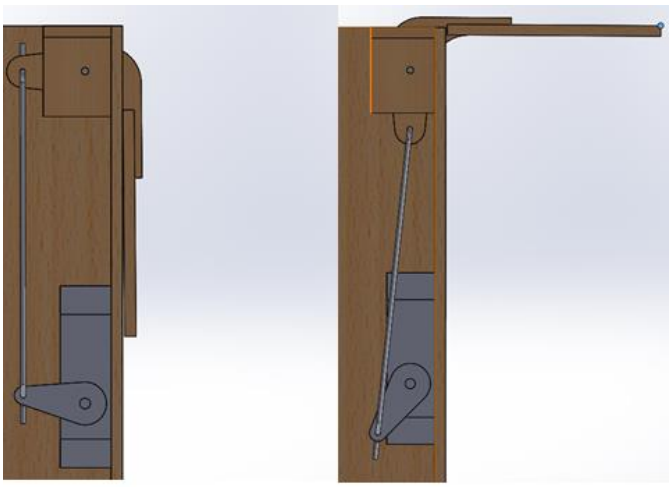

Figure 5. Flap Mechanism

The Arduino Uno was replaced with an Arduino Mega for its greater memory. The Arduino Mega was mounted on its own mounting board using nylon bolts and nuts. The two batteries were also mounted on their own mounting board with zipties and reinforced with double sided tape. Each mounting board was also fitted with a buck converter. The mounting boards were slotted in place into supports formed by cross-halving joints. A latching switch was added to the top panel of the vehicle for external activation of the control board.

Fabrication

Figure 6. Fabricated Vehicle

4” x 36” Balsa sheets provided the bulk of the structure of the vehicle as seen in the above figure. 4” x 8” cuts were joined with wood glue in order to form six 8” x 8” panels. These panels were then cut and sanded down to size for proper fit. The mounting boards were also fabricated in this fashion. Cross members were then cut and assembled into slotted “X” structures. These structures had cavities to allow for the mounting of the latching switch and servo controller. These members were glued to the top and bottom panels of the vehicle. The ballast was fixed to these panels using hot glue. The control board was mounted to its mounting board with nylon bolts and nuts. The buck convertors were mounted with adhesive tape and hot glue to their respective mounting boards.

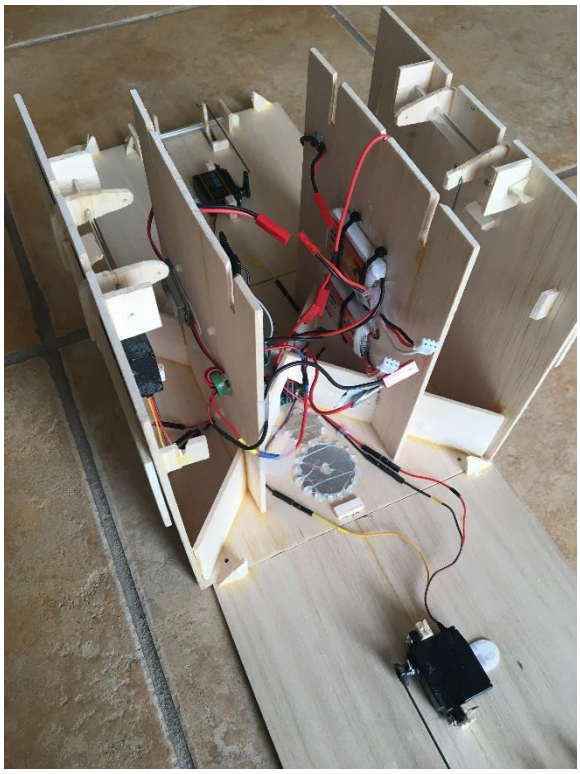

Figure 7. Vehicle with Panels Removed

The inner components of the vehicle are shown in the above figure. The two 2S LiPo batteries were similarly mounted with adhesive tape and reinforced with zipties. Adhesive tape was also used in mounting the four servos to their respective panels. Balsa "brackets" were glued to each panel and the servos were simply mounted using the supplied servo screws. Unfortunately, the shape of the supplied servo horns were not suitable for this application. Therefore, larger servo horns were cut and sanded to shape as seen below.

Figure 8. Modified Servo Horn and Supplied Servo Horns

Stainless steel shafts 2mm in diameter served as pivots for the flap to pivot about. Music wire was cut and bent to size to form pushrods for each servo. The rods were first inserted into the flap rocker arm and then the servo horn. The servo horn was then attached to the servo with a simple press fit. The flap rocker arms were attached to the flaps through glued slots. The maximum flap angle measured for a flap was roughly 76º from the flap face. Most of the electronic components were connected and tested before assembly, which is discussed later in the chapter. Two opposing sides of the vehicle were glued to the vehicle bottom and the control mounting board was also glued to the vehicle bottom.

Because of the nature of the LiPo batteries, their balance ports needed to be accessible for balance charging. Therefore the battery mounting board was left unglued for easy removal. Two side panels were directly glued to the vehicle bottom panel. The remaining side panels were attached to the top and bottom utilizing nylon bolts and nuts This allowed for easy removal of the two unglued panels.

The vehicle was weighed using a simple kitchen scale, yielding a mass of 0.56 kg. The vehicle's center of mass was placed as close to the ideal center of the box as possible when modeling. The vehicle's actual center of mass was found by hanging the vehicle at two collinear points and drawing lines directly below them. This was repeated multiple times until the intersections marked two points that correlated with the vehicle's center of mass in 3D space as shown in the table below.

Table 1. Center of Mass (from ideal center)

x (in)	y (in)	z (in)
0	0	0.0625

Electronics

The transition from the Arduino Uno to the Arduino Mega was simple, overall. The BNO055 IMU was not mounted on the Uno itself, but to a data logging shield which was compatible with both the Uno and Mega.

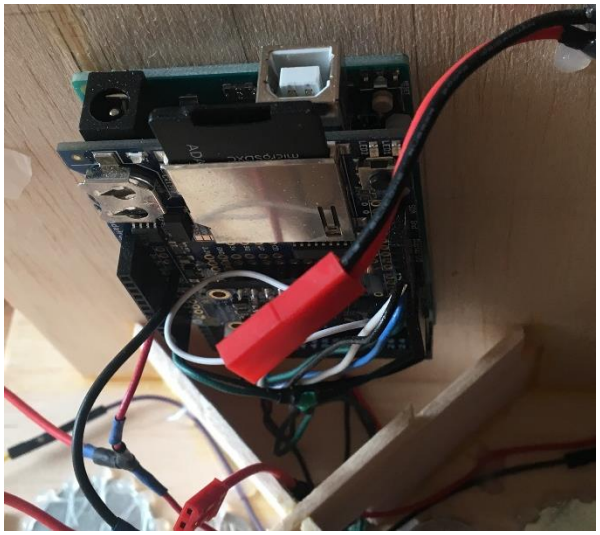

Figure 9. Arduino Mega with Data Logging Shield

Nevertheless, there was some modification necessary due to the mapping of pins. In order to log data through the data logging shield, the Arduino Uno's SPI ports were used. These ports were mapped to different pins on the Mega, requiring jumper wires from the SCK, MOSI, MISO, and SS pin of the Mega to those mapped on the shield. The Arduino Mega's I2C ports were handled similarly as they were necessary for communication with the BNO055 IMU and servo controller.

The servo controller used was the Teensy 2.0 for its small form factor and compatibility with the Arduino microcontroller. The Teensy was connected to the Arduino Mega through its SDA and SCL wires for I2C communication. The four servo control signal wires were connected to the Teensy through jumper wires soldered to the

Teensy. This allowed for complete removal of front and back panels for easy repair without soldering. Power was distributed from the buck converters to the control boards through the control board 5V pins. JST RCY connectors were used in many places in the vehicle to facilitate access to components as shown in the following figure.

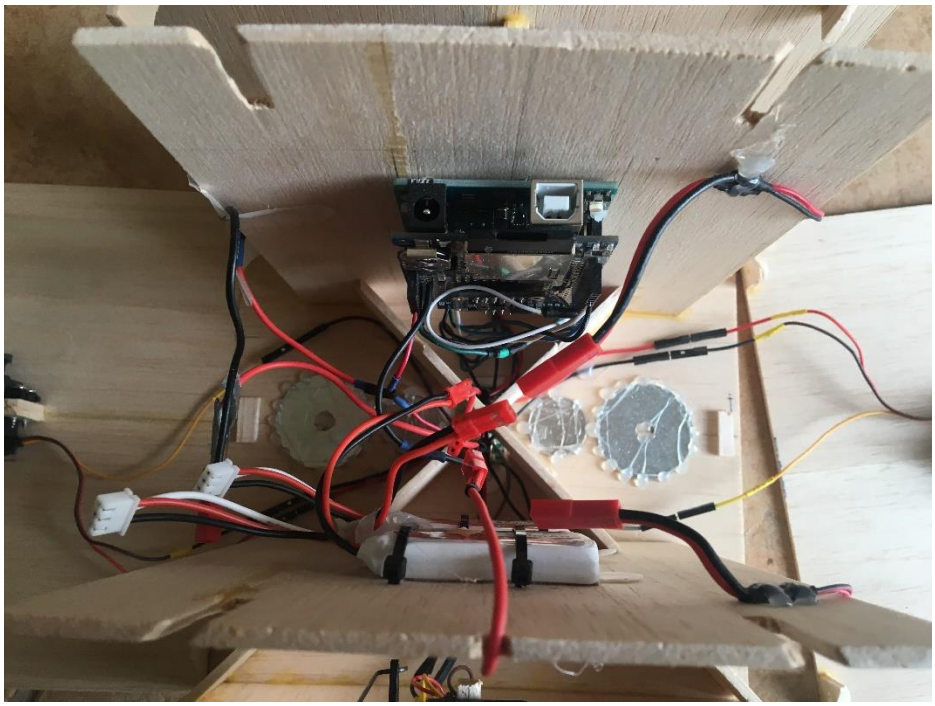

Figure 10. Top View of Vehicle Interior

The top panel was made removable for access to the Arduino Mega's USB B port, which was used to program the Arduino. To prepare the vehicle for data collection, the combined servo positive leads were first connected. Separate wires were used to distribute power such that the system could be tested when the servos were not used. The two batteries were each connected to buck converters to supply power to the system. The batteries were always disconnected when the vehicle was not in use because the buck converter LEDs drained power even if the system was not active.

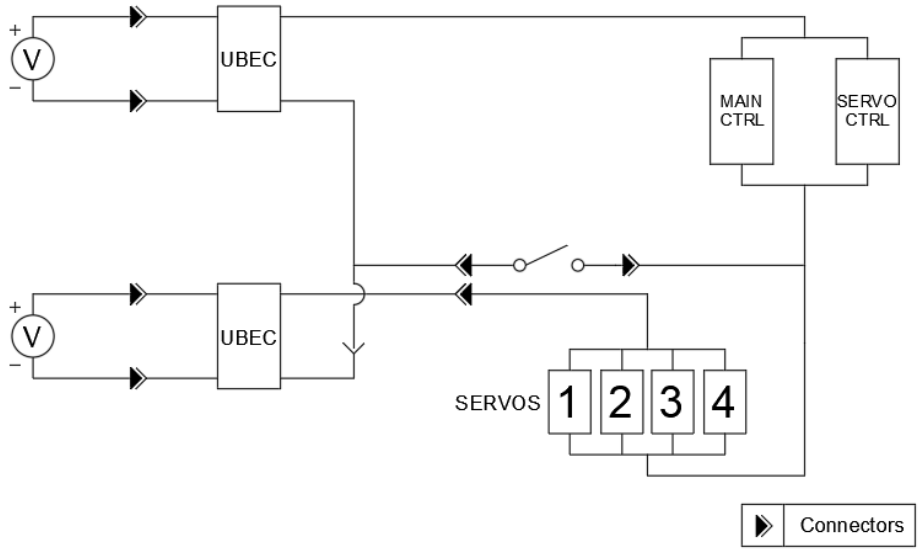

Figure 11. Wiring Schematic

As shown in the schematic above, the grounds of the buck converters were tied to those of the rest of the system through the latching switch. By connecting the system grounds at the switch, the vehicle could be turned on with one button press. These connections are shown in the below figure. The latching switch was integrated with an LED that was initially used, but found unnecessary. Consequently, the LED was left unconnected because of trouble in assembly. With the body of the vehicle complete, the vehicle now had to be programmed to effectively collect data.

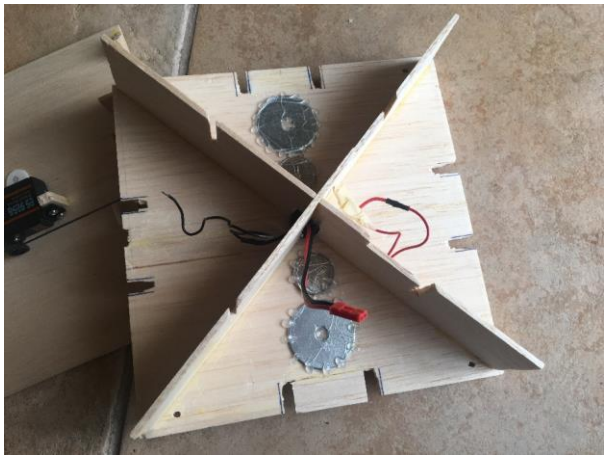

Figure 12. Vehicle Top Panel

Data Collection Programming

Initially, only one main program was created for the vehicle with the intent of encompassing IMU calibration as well as data collection. Calibration of the BNO055 mainly involved manual calibration of its magnetometer and accelerometers. The IMU gyroscope was simply calibrated by allowing the vehicle to rest for a short period of time. The magnetometer was calibrated by moving the vehicle in a figure 8 pattern in order to determine magnetic north. The accelerometer was calibrated by tilting the vehicle and holding it at various angles until it was fully calibrated. The calibration process was found to be lengthy as it could take up to 10 minutes for one calibration. Consequently, re-calibrating the vehicle after every instance of data collection was rather inconvenient and time-consuming.

Thus, two main programs were used in order to save time and effort. The first program used was a program included in the BNO055 library which saved calibration offsets to EEPROM. Thus, the values persisted through system resets and were able to be reused for successive data collections. The second program was created to read these offsets, apply the calibration, and collect data.

After calibration, the program initialized the SD card with a new logging file and sent servo positions to the Teensy controller. These positions were sent to the controller as the servos were actuated to full extension on startup. The closing of the flaps also served as a visual aid for when the vehicle was ready to be placed in the launcher. Once the flaps were closed, the program monitored accelerometer readings from the IMU. If the vehicle rested on some surface, the IMU experienced a gravitational acceleration of around $9.8 m/s^2$. Once the vehicle experienced “weightlessness”, i.e. an acceleration less

than 2 m/s^2, the vehicle started data logging and flap actuation. This data logging could have been unintentionally activated when handling the box improperly. Therefore in order to avoid this, the program was required to have been in operation for over 5 seconds for the collection of data. Data logged included: time after program start, orientation quaternions, angular velocities, and linear accelerations. After a predetermined number of microseconds from when the vehicle dropped, the flaps were closed to minimize damage on impact.

The main advantage for the selection of the BNO055 was its integrated sensor fusion algorithm. By taking data from the IMU's accelerometers, gyroscopes, and magnetometers, the algorithm was able to supply absolute orientation data in the form of quaternions which were amenable to straightforward manipulation. The fusion algorithm supplied data at a rate of 100Hz, but data was sampled at about 20 Hz when logging into a plaintext .csv file. In order to capture as much data as possible, the data was instead logged in binary as a C structure, which increased the sample rate to about 72 Hz.

The Arduino sent servo positions to the Teensy in a master-slave relationship. This necessitated a third program on the Teensy. The four flaps were assigned numbers and their positions were formatted into an array that could be sent over I2C. Once received, the data was parsed by the Teensy and each servo was actuated according to its respective position in the array.

DATA COLLECTION

Launcher & Net

A 10’x10’ net platform and a mechanical launcher were created for data collection in the interest of repeatability and safety. Dropping the vehicle from a cantilever rather than from a wall face minimized any chance of interference from the wall as the vehicle progressed downward. Eight 5’ sections of 1-1/2” PVC pipe were cut to form the frame of the net platform. Eight 2’ sections of pipe were also cut for the platform legs. The frame and legs were joined with PVC tees and 3-way elbows. A 10’ x 10’ backstop net was secured to the net frame using ball bungee cords.

Figure 13. Launcher Body

As shown in the figure, the main part of the launcher was composed of a 5’ 2”x4” with a single bracing strut and a protruding lever to resist rotation. The gripper structure was composed of various lengths of 1” x 2” arranged in an angled jaw configuration shown in the below figure. Common cabinet hinges provided the pivot points for the gripper jaws. The gripping interfaces were composed of 1/4” x 1-5/8” molding arranged in X patterns with grip pads lining their interiors.

Figure 14. Gripper (Open)

The gripping interfaces were fixed to 3/8” aluminum shafts with shaft collars. These shafts in turn were inserted into bearings that were mounted on the jaws. This allowed the vehicle to rotate while fixing its position.

The mechanism used for the launcher was inspired by crossbows. Screw eyes were installed at various places on the gripper to provide attachment points and redirection for bungee cords. In the gripper’s default state, bungee cords pulled at the backs of the jaws for an open jaw configuration. These cords provided a spring force that ensured full extension. A parachute cord tension line was attached to the edges of the jaws to close the jaws fully as shown in the following figure.

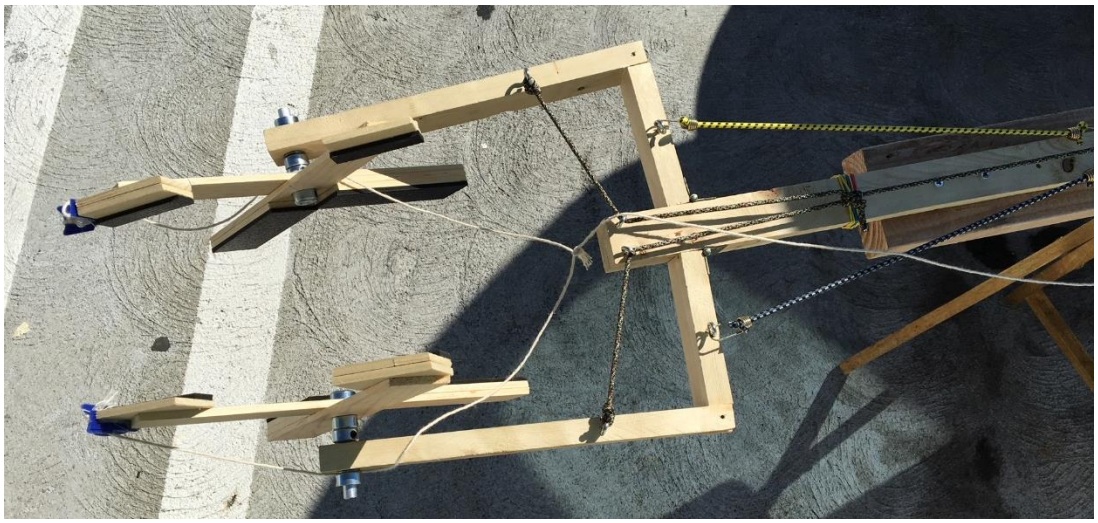

Figure 15. Gripper (Closed)

The tension line was adjusted with a pulley to clear the gripper structure. The line was attached to another pulley on the launcher body in a modified block and tackle arrangement. The pulley line was fixed to the launcher body at one end and tied into a bowline at the other. Hooked bungee cords were used to connect the pulley line to a panic snap at the fixed end of the launcher as seen below. Used primarily for horse riding, the panic snap allowed for quick release of the gripper jaws.

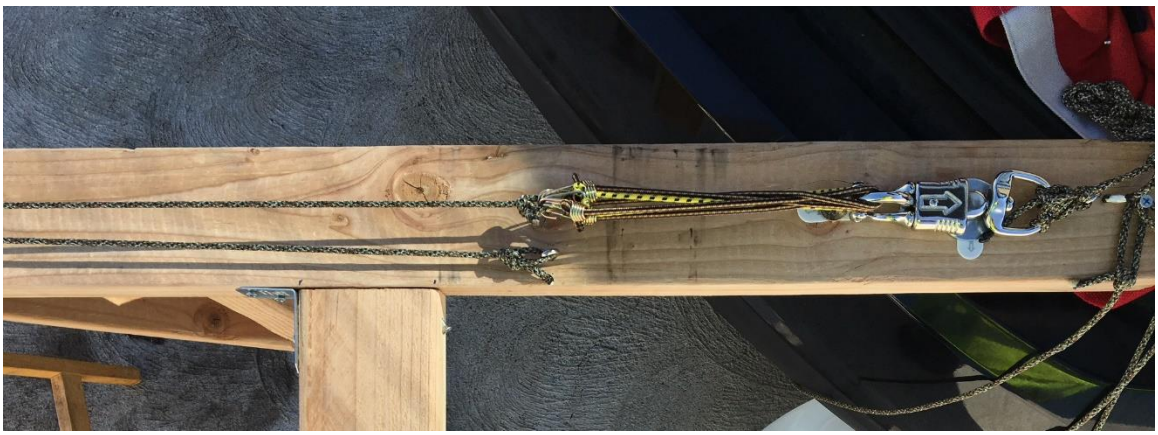

Figure 16. Panic Snap on Launcher Body

Data Acquisition Process

Data collection took place at a parking structure measuring roughly 36’ tall. The net platform was set up at the drop point with a rubber mallet and ball bungee cords as seen in the following figure. The drop point was selected where no cars or people were present and the surrounding area was cordoned off with caution tape. A person was also on standby in the area as an additional precaution.

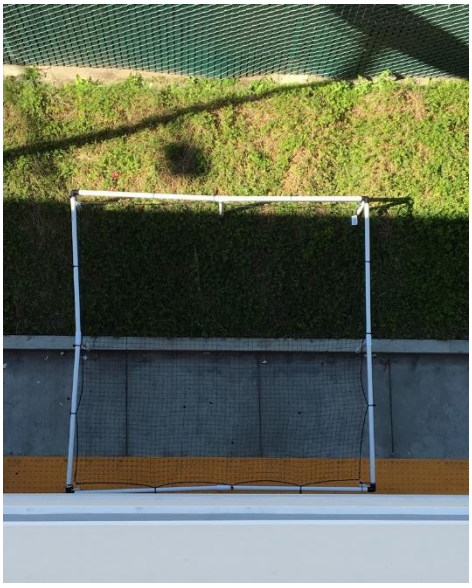

Figure 17. Net at Drop Site

At the top of the parking structure, the launcher was assembled and set between two surfaces for loading. The panic snap was first set and the hooked bungee cords were hooked one by one on to the pulley line through the panic snap release hook. This added tension gradually for better control. A dummy vehicle of similar size and weight to the vehicle was loaded into the launcher by opening the jaws and adjusting the vehicle as they closed. The dummy vehicle was made of cardboard panels and fitted with one dummy flap as shown below.

Figure 18. Dummy Vehicle

Two 1-gallon jugs of water were affixed to the butt of the launcher to secure everything in place once the launcher was positioned. With the dummy vehicle fixed in the jaws, the launcher was carried over the lip of the concrete wall of the parking lot as seen in the following figure.

Figure 19. Positioned Launcher

The launcher was then adjusted and multiple trial drops were performed. These drops were timed and the launcher & net positions were adjusted for accuracy. The drop time was recorded at an average of 1.6 seconds. In order to account for human error as well as other variables, the drop time inputted into the vehicle was 1.4 seconds.

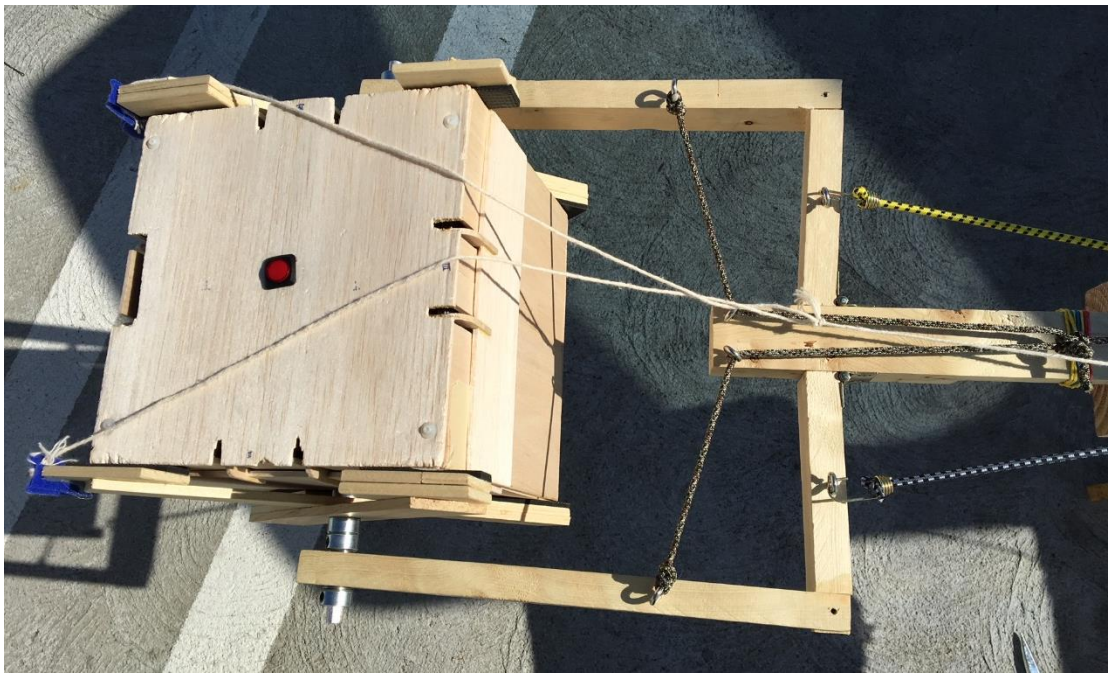

Figure 20. Gripper with Loaded Vehicle

The vehicle was first uploaded with the calibration program described in the vehicle programming section. This was done without connecting the servos to save battery life. Once calibrated, the data collection program was uploaded into the Arduino with the chosen drop time. The batteries were connected to the buck converters and the servo positive wires were also connected. The top panel was then set on top of the vehicle and the main grounds and servo ground were connected to the interior. The top panel was then bolted down with nylon bolts. The area around the vehicle was cleared since the flaps would extend fully on startup. A few seconds after this full extension, the flaps

would close, signaling that it was ready for data collection. A baseline orientation was first acquired by throwing the vehicle slightly up in the air and setting it down on a level surface. This orientation data was later used to define an inertial rotational frame. The vehicle was then put through the same launching process until adequate data was acquired. Strips of Velcro attached to twine were used to induce or halt rotation of the vehicle during launches. The twine was pulled as the jaws were released in order to produce a controlled rotation. The different initial conditions for the drops are shown in the table below. Each configuration was run a minimum of three times to acquire data.

Table 2. Initial Conditions for Data Collection

Initial Condition
No Initial Rotation
Initial Pitch
1 Flap at full extension with no Initial Rotation

The no initial rotation condition was chosen as a baseline for the modeling of the vehicle. The condition with initial pitch was performed to examine the effects of a significant rotation relative to the baseline. The 1 flap at full extension condition was also utilized to examine the effect of the flap on the vehicle behavior.

In order to properly interpret this data, two factors needed to be addressed. The relationship between different coordinate systems in the experiment needed to be defined. These were informed by measurements taken by the system. Furthermore, the values of the inertia tensor of the vehicle were necessary for a dynamic model of the vehicle behavior.

DATA PROCESSING

Frames

The default frame attached to the IMU had the x and y axes attached normal to the board face with the z-axis going into the face of the board. With the IMU mounted vertically, the y-axis pointed upward when the vehicle was at rest. Minor misalignments during mounting were corrected by finding the rotation matrix that rotated the IMU frame into a frame normal to the vehicle faces. This was accomplished by rotating the vehicle around each desired cube axis and building a rotation matrix from the recorded unit vectors. The recorded angular velocities for the z-axis are shown below, with the rotation about the desired axis clearly seen in the right side.

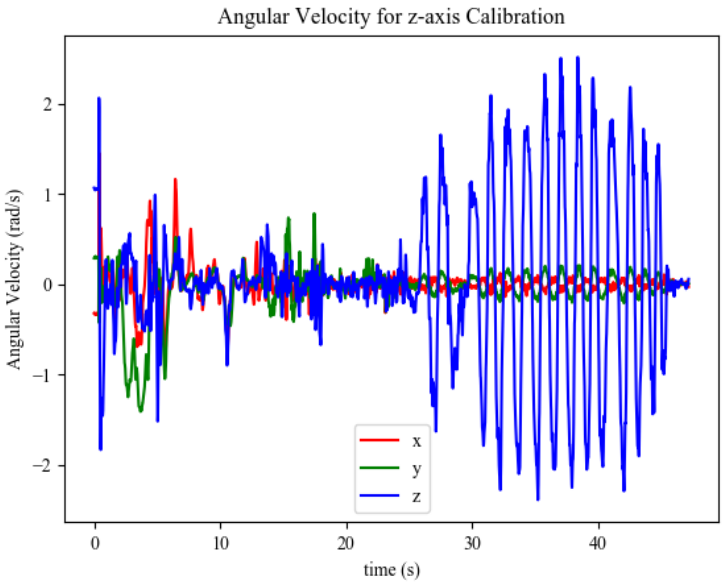

Figure 21. Raw Pitch Velocity Data

This rotational data was filtered and the unit vectors were found as seen in the below figure.

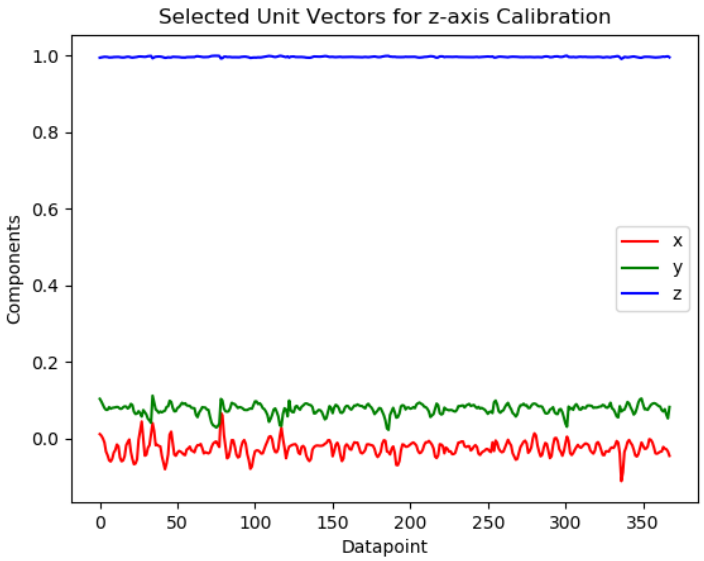

Figure 22. Unit Vectors from Pitch Velocity Data

The average unit vector seen by this rotation about the desired z-axis was thus found. This was repeated for every desired cube axis, resulting in unit vectors for the x, y, and z-axes. A rotation matrix defines the unit vectors of one frame in terms of the unit vectors of another frame. Thus, the rotation matrix that transforms the IMU frame to the desired vehicle frame can then be defined as follows.

$$R_{IMU}{}^{Box} = \begin{bmatrix} i_{xCal} & j_{xCal} & k_{xCal} \\ i_{yCal} & j_{yCal} & k_{yCal} \\ i_{zCal} & j_{zCal} & k_{zCal} \end{bmatrix}$$

Using the found unit vectors, this rotation matrix was calculated and used to correct for any misalignments of the IMU frame with the vehicle normal frame. However, an additional rotation matrix needed to be defined for a reference frame.

The IMU recorded absolute orientation with respect to magnetic north. While necessary for calibration, the exact orientation of this reference frame was unknown in the real world. Therefore, a baseline reference orientation was captured and used as the inertial “world” frame.

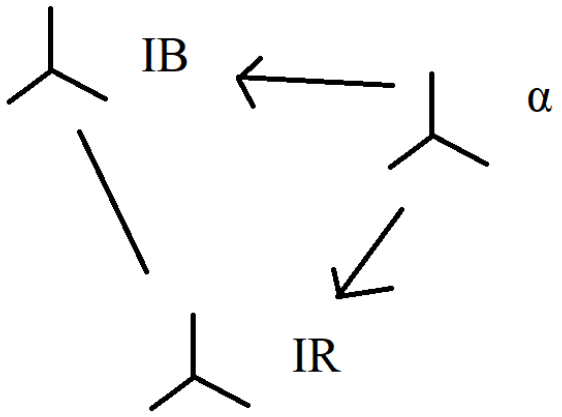

Figure 23. Rotational Frames

Let frame {α} be the magnetic reference frame, {IR} be the desired reference frame, and {IB} be the measured orientation of the IMU at any given time shown in the figure above. The IMU logged orientations were in the form of quaternions. Translated into rotation matrices, the gathered data was equivalent to the following:

$$R_{IB}{}^{\alpha} = Orientation\ of\ frame\ \{IB\}\ with\ respect\ to\ frame\ \{\alpha\}$$

$$R_{IR}{}^{\alpha} = Orientation\ of\ desired\ ref.frame\ \{IR\}\ with\ respect\ to\ frame\ \{\alpha\}$$

Defining the rotation matrix between the body and desired reference frame:

$$R_{IB}{}^{\mathrm{IR}} = Orientation\ of\ frame\ \{IB\}\ with\ respect\ to\ frame\ \{\mathrm{IR}\}$$

Given a reference rotation matrix $R_{IR}{}^{\alpha}$ and a body rotation matrix $R_{IB}{}^{\alpha}$ this orientation can be found through the transpose of the reference matrix.

$$R_{IB}{}^{\mathrm{IR}} = (R_{IR}{}^{\alpha})^{T} R_{IB}{}^{\alpha}$$

Thus, the orientation of the body with respect to the desired reference can be found at any time as long as $R_{IR}{}^{\alpha}$ and $R_{IB}{}^{\alpha}$ are known.

Applying Frames & Processing

A variety of functions were made in Python in order to process the data files into a common reference frame. Python also uses entities known as "classes" to bind data and functionality together. One such class was created for the importing of the data files generated by the vehicle. Every instance of the class opened a specified data file and imported the C structure data into different arrays for each datatype. The magnitude of the acceleration was also calculated and stored for each sample. The class incorporated various functions for the manipulation of this imported data.

A "drop only" function truncated the data such that samples only taken during freefall were retained. While data collection started with freefall, the vehicle kept recording data until it was retrieved from the net and turned off. Therefore, extraneous data was removed by locating the first sample where the magnitude of the recorded acceleration was first greater than 10 m/s^2. This sample indicated where the vehicle first hit the net and was a reliable indicator of where to remove data. Data was removed a few samples before the found sample.

A "normalize world" function was created in order to obtain the rotation matrix of the vehicle relative to a pre-defined inertial frame at every sample. The function imported a predefined reference rotation matrix from the desired inertial frame to the IMU inertial frame. A second predefined rotation matrix converted the orientation from the IMU inertial frame to the chosen body frame (normal to the vehicle faces). The function then converted the class' quaternions into rotation matrices and applied the rotation into the desired inertial frame. This resulted in rotation matrices describing the vehicle orientation

relative to the vehicle reference frame. The rotation was also applied to the angular velocity data and both sets of data were saved.

A “world base” function was created to derive the rotation matrix from the IMU inertial frame to the desired rotation frame. The function examined the data collected for the desired reference frame. The function used user-found bounds in order focus on data where the vehicle was stable during the reference frame data acquisition. The quaternions in this range were averaged and the final rotation matrix was found from this mean.

A supplementary “define world” program was created in order to utilize the world base function and define the desired inertial frame. The program called an instance of the data class using the baseline data file. A plot was generated from the quaternion data and bounds were selected from this plot where there was little change. The bounds were then passed to the world base function and the resulting rotation matrix and its inverse were saved in a reference file for future use. Thus, future programs would only have to be passed the name of the reference file for the viewing of data in the inertial frame.

A second supplementary program was created for the definition of the vehicle frame relative to the IMU attached frame. The vehicle was spun about each of the desired vehicle axes while on the gripper and the angular velocity data was recorded. The data was normalized and the unit vector in the IMU frame for each vehicle axis was determined. The unit vectors for the three vehicle axes were combined into a rotation matrix representing the vehicle frame seen by the IMU frame. However, like the other rotation matrices derived from data, this matrix did not fit the actual definition of a rotation matrix.

A problem with the derivation of rotation matrices was the loss of orthogonality. The rows and columns of a rotation matrix are supposed to represent unit vectors with magnitudes of one. However, numerical errors can cause them to be slightly larger or smaller. Rotations with these non-orthogonal matrices could cause unwanted errors when performing rotations. Therefore, a "renormalization" approach was undertaken where rotation matrices were derived from experimental data. The dot product of the 1st two rows was taken and the resulting value was taken to be the error form the orthogonal ideal. The error was split in half and the two rows were adjusted accordingly by cross-coupling. The third row of the rotation matrix was then adjusted to be orthogonal to the 1st and 2nd rows through the cross product. The row vectors were then scaled such that their magnitudes were equal to one [10].

The data collected was also curve fit for better resolution of the vehicle behavior. Interpolation of the vehicle orientations was accomplished through the use of Slerp, also known as **s**pherical **l**inear int**erp**olation [5]. Instead of linear interpolation, Slerp interpolated orientations across the surface of a sphere. The angular velocities were interpolated by first fitting a spline to the data. Then a Savitsky-Golay filter was applied for smoothing [15].

Inertia Tensor

A preliminary estimate of the inertia tensor was derived from the Solidworks model. The weight of the structure and components of the vehicle initially left it with unbalanced mass properties. The presence of products of inertia in a tensor contribute to dynamic imbalance as the rigid body moves in space. In order to minimize this imbalance and simplify the dynamic model, the vehicle's products of inertia were desired to be as small as physically possible in the vehicle frame. Therefore, weight was added at strategic locations for this balancing. 1-1/2" washers were placed on the top and bottom panels of the vehicle in order to minimize imbalances caused by the masses of the battery and control boards and place the center of mass at the box's ideal center point. Quarters were placed next to these washers for further fine tuning of this balancing.

The CAD model was refined through the placing of these masses and the products of inertia were reduced very close to zero. However, the model did not account for the wires, glue, and general non-uniformity of components. Therefore, it was necessary to obtain the actual inertia tensor of the vehicle from experimental values.

The moments of inertia about certain axes of the vehicle were determined using a bifilar torsional pendulum. Twine was used for the filars and attached to the vehicle using masking tape as shown below. The weight of these were determined to be negligible compared to the vehicle.

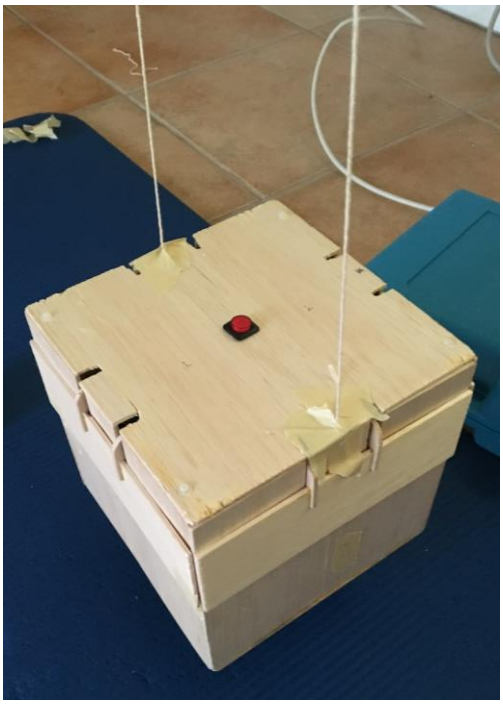

Figure 24. Vehicle Suspended as Bifilar Pendulum

By timing the period of oscillation, the moment of inertia for an axis was determined as follows:

$$I = \frac{mgr^2}{4\pi L}T^2$$

Where m·g is the weight of the vehicle, r is the radius from a filar (string) to the axis of rotation, L is the length of the string, and T is the period [11]. Three periods were taken for each axis observed, averaged, and used to find the axis' moment of inertia as shown below.

Table 3. Measured Moments of Inertia

Moment of Inertia	Normal	X Rotated 45º	Y Rotated 45º	Z Rotated 45º
Ixx	3.685E-03	Normal Ixx	3.808E-03	3.691E-03
Iyy	3.455E-03	3.735E-03	Normal Iyy	3.681E-03
Izz	3.728E-03	3.767E-03	3.833E-03	Normal Izz

The axes observed were the 3 axes normal to the vehicle faces as well as the 6 axes representing rotations of 45° from any one normal axis to be used later for the determination of the products of inertia. To this end, the rotational kinetic energy equation was looked to as inspiration for the relationship of an inertia tensor to itself in a different reference frame.

The rotational kinetic energy of a rigid body in a reference frame {0} can be found as follows:

$$T = \frac{1}{2}\{\boldsymbol{\omega}^{(0)}\}^T \boldsymbol{I}^{(0)}\{\boldsymbol{\omega}^{(0)}\}$$

Where ω is the body's angular velocity and I is the object's tensor of inertia [9]. The angular velocity can be transformed into a body frame {b} through a rotation:

$$\{\boldsymbol{\omega}^{(b)}\} = \boldsymbol{R_0}^{(b)}\{\boldsymbol{\omega}^{(0)}\}$$

And vice-versa using the transpose of the same rotation:

$$\{\boldsymbol{\omega}^{(0)}\} = \left(\boldsymbol{R_0}^{(b)}\right)^{\boldsymbol{T}}\{\boldsymbol{\omega}^{(b)}\}$$

The rotational kinetic energy of the body remains the same no matter the reference, therefore:

$$T = \frac{1}{2}\{\boldsymbol{\omega}^{(b)}\}^T \boldsymbol{I}^{(b)}\{\boldsymbol{\omega}^{(b)}\}$$

Substituting for ω^b:

$$T = \frac{1}{2}\{\boldsymbol{\omega}^{(0)}\}^T \left[\left(\boldsymbol{R_0}^{(b)}\right)^{\boldsymbol{T}} \boldsymbol{I}^{(b)} \boldsymbol{R_0}^{(b)}\right]\{\boldsymbol{\omega}^{(0)}\}$$

Equating this rotational kinetic energy to the first kinetic energy equation gives a similarity transformation for the inertia tensor.

$$\boldsymbol{I}^{(0)} = \left(\boldsymbol{R_0}^{(b)}\right)^{\boldsymbol{T}} \boldsymbol{I}^{(b)} \boldsymbol{R_0}^{(b)}\text{ , or}$$

$$\boldsymbol{I^{(0)}} = \boldsymbol{R_b}^{(0)}\, \boldsymbol{I^{(b)}} \left(\boldsymbol{R_b}^{(0)}\right)^{\boldsymbol{T}}$$

This equation allows for the transformation of the inertia tensor in the body frame to that in the inertial frame. This is useful because the inertia tensor in the body frame is a constant matrix that is independent of the motion of the body [12]. Examining this transformation in detail, let N refer to the inertia tensor in the body frame:

$$\boldsymbol{R_b}^{(0)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{23} & a_{33} \end{bmatrix}$$

$$\boldsymbol{I^{(0)}} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

$$\boldsymbol{N^{(b)}} = \begin{bmatrix} N_{xx} & -N_{xy} & -N_{xz} \\ -N_{xy} & N_{yy} & -N_{yz} \\ -N_{xz} & -N_{yz} & N_{zz} \end{bmatrix}$$

Substituting into the similarity transform:

$$\begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{23} & a_{33} \end{bmatrix} \begin{bmatrix} N_{xx} & -N_{xy} & -N_{xz} \\ -N_{xy} & N_{yy} & -N_{yz} \\ -N_{xz} & -N_{yz} & N_{zz} \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Focusing on the mass moments of inertia on the left hand side and performing the matrix multiplication on the right hand side, the following linear equations were derived:

$$I_{xx} = {a_{11}}^2 N_{xx} + {a_{12}}^2 N_{yy} + {a_{13}}^2 N_{zz} - 2a_{11}a_{12}N_{xy} - 2a_{12}a_{13}N_{yz} - 2a_{11}a_{13}N_{xz}$$
$$I_{yy} = {a_{21}}^2 N_{xx} + {a_{22}}^2 N_{yy} + {a_{23}}^2 N_{zz} - 2a_{21}a_{22}N_{xy} - 2a_{22}a_{23}N_{yz} - 2a_{21}a_{23}N_{xz}$$
$$I_{zz} = {a_{31}}^2 N_{xx} + {a_{32}}^2 N_{yy} + {a_{33}}^2 N_{zz} - 2a_{31}a_{32}N_{xy} - 2a_{32}a_{33}N_{yz} - 2a_{31}a_{33}N_{xz}$$

$$-I_{xy} = a_{21}a_{11}N_{xx} + a_{22}a_{12}N_{yy} + a_{23}a_{13}N_{zz} - (a_{21}a_{12} + a_{22}a_{11})N_{xy} - (a_{21}a_{13} + a_{23}a_{11})N_{xz} - (a_{22}a_{13} + a_{23}a_{12})N_{yz}$$
$$-I_{xz} = a_{31}a_{11}N_{xx} + a_{32}a_{12}N_{yy} + a_{33}a_{13}N_{zz} - (a_{31}a_{12} + a_{32}a_{11})N_{xy} - (a_{31}a_{13} + a_{33}a_{11})N_{xz} - (a_{32}a_{13} + a_{33}a_{12})N_{yz}$$

$$-I_{yz} = a_{31}a_{21}N_{xx} + a_{32}a_{22}N_{yy} + a_{33}a_{23}N_{zz} - (a_{31}a_{22} + a_{32}a_{21})N_{xy} - (a_{31}a_{23} + a_{33}a_{21})N_{xz} - (a_{32}a_{23} + a_{33}a_{22})N_{yz}$$

Assuming that the rotation between the tensors is about a common axis between the frames, the equations can be greatly simplified. For single rotations about the x, y, and z axes (φ, θ, and ψ), the rotation matrices for these rotations are as follows. Further information can be found in Appendix B [3].

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When inserted in to the first three equations, the zeroes inherent in the rotation matrices eliminate many terms, resulting in the following.

$$I_{xx} = {a_{11}}^2 N_{xx} + {a_{12}}^2 N_{yy} - 2a_{11}a_{12}N_{xy} for\ rotation\ about\ the\ z\ axis$$
$$I_{yy} = {a_{22}}^2 N_{yy} + {a_{23}}^2 N_{zz} - 2a_{22}a_{23}N_{yz}\ for\ rotation\ about\ the\ x\ axis$$
$$I_{zz} = {a_{31}}^2 N_{xx} + {a_{33}}^2 N_{zz} - 2a_{31}a_{33}N_{xz}\ for\ rotation\ about\ the\ y\ axis$$

Thus, the products of inertia in one frame can then be found as long as the moments of inertia are known in both frames and the angle between them.

$$-N_{xy} = \frac{I_{xx} - (\cos\psi)^2 N_{xx} - (\sin\psi)^2 N_{yy}}{-2\cos\psi\sin\psi}$$

$$-N_{yz} = \frac{I_{yy} - (\cos\varphi)^2 N_{yy} - (\sin\varphi)^2 N_{zz}}{-2\cos\varphi\sin\varphi}$$

$$-N_{xz} = \frac{I_{zz} - (\sin\theta)^2 N_{xx} - (\cos\theta)^2 N_{zz}}{-2\sin\theta\cos\theta}$$

The nine measured moments of inertia were thus used to determine the products of inertia of the vehicle. These products of inertia were close enough to zero that additional mass balancing was not deemed necessary.

Table 4. Calculated Inertia Tensor (kg-m^2)

Ixx	Iyy	Ixx	-Iyz	-Ixz	-Ixy
3.69e-3	3.48e-3	3.71e-3	-1.44e-4	-2.63e-4	-1.21e-4

These moments of inertia also were within the expected range of the Solidworks model and the values agreed relatively well.

Table 5. Modeled Inertia Tensor (kg-m^2)

Ixx	Iyy	Ixx	Iyz	Ixz	Ixy
3.32e-3	3.24e-3	3.31e-3	3.80e-06	0	1.29e-06

Vehicle Geometry

An “Area Calcs” class was made in order to interpret the geometry of the vehicle as it changed orientation. Corner points were defined from the ideal center of the vehicle. These points were in turn used to define the 6 faces that comprised the vehicle. The geometry of the flaps was more complex and was therefore handled using frames at the flap pivot points. Relative to a frame attached to a flap surface, the flap’s position in space does not change. Therefore, transformation matrices were defined such that flap-attached frames could be transformed into frames defined at the pivot points. From there, the points were transformed using another transformation matrix describing the distance between the flap pivot points and the ideal center. Thus, the geometry of the flaps could be determined given the flap angles from rest. Once all relevant points were found, the class rotated the geometry into the world frame with the inputted orientation matrix. This rotation matrix was converted in to a transformation matrix and applied to all points in order to obtain the vehicle geometry in the world frame.

Visualizing Orientation

A function was created for the visualization of the orientation of the vehicle. An instance of the geometry class was called for the points in the world frame. Lines were also created to define the edges of the vehicle in wireframe form. Colored diagonal lines were plotted where a face was detected to be facing downward. All points were converted into the 3D plot frame for plotting purposes. The image resulting from the function, however, was displayed in terms of the world frame. This function was useful in making sense of how a specific rotation matrix correlated to an orientation in space.

Orientations were also plotted over time as Euler angles as shown in the figure below. The convention utilized involved rotation about the world x-axis, then the y-axis, and finally the z-axis (also known as the xyz-convention) [3].

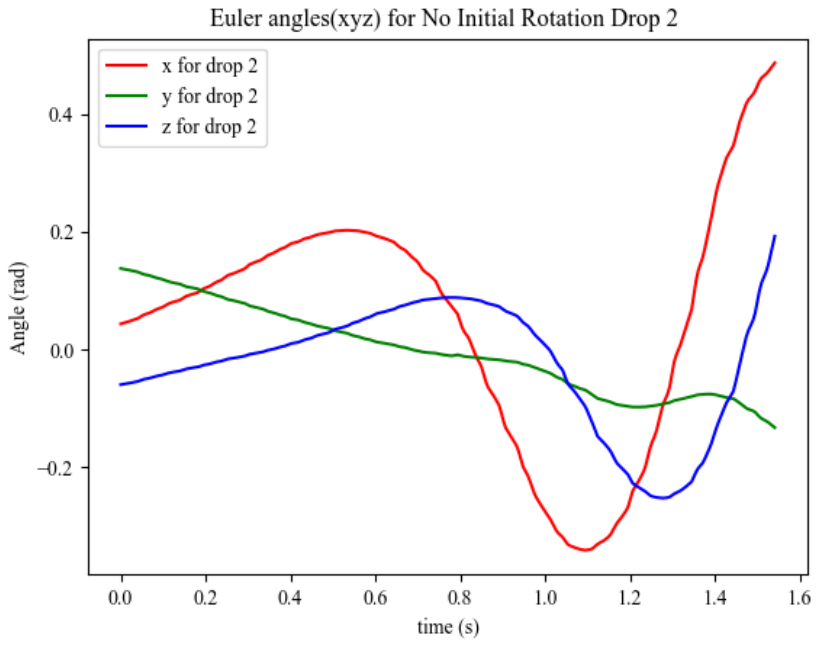

Figure 25. Euler Angles for Drop Number 2 with No Initial Rotation

For a broader view of the vehicle orientation, an animation program was created. For a specified data file, the orientations of the vehicle were plotted for every sample.

The frames were then collected and written into a video file. Thus, the behavior of the vehicle could be more readily determined visually as seen in the following figure.

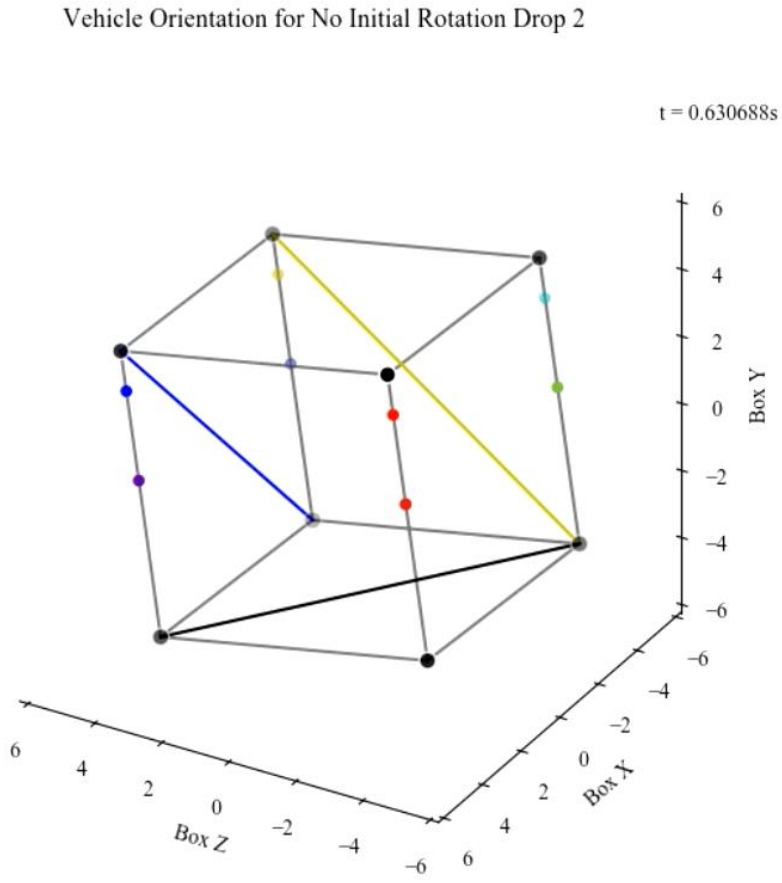

Figure 26. Vehicle Animation Still

RESULTS

Processed Data

Two angular velocity profiles for drops with no initial rotation are shown in the below figure. These profiles are shown resolved in the desired reference frame found earlier. Though both drops exhibited similar behavior, the repeatability of this behavior was heavily affected by slight differences in the initial conditions. While significant initial rotation was prevented, small initial rotations from the releasing of the vehicle were still present during data collection.

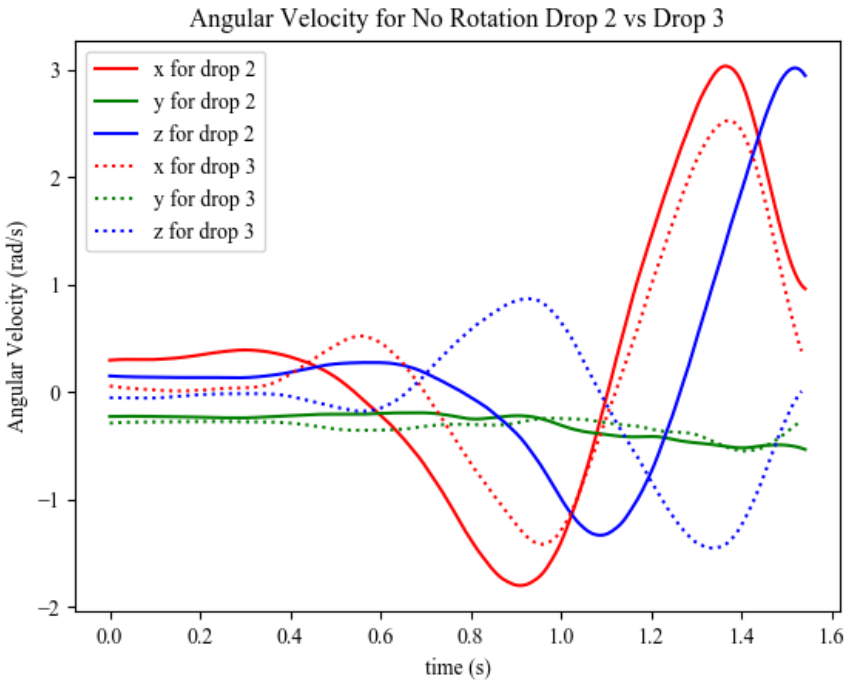

Figure 27. Comparison of Angular Velocity Profiles with No Initial Rotation

The angular velocity profile for a drop with significant rotation in the pitch direction is shown below. The profile exhibited much less variation in angular velocity due to the magnitude of the existing angular momentum. The act of starting the rotation most likely induced some twist in the launcher, resulting in some angular velocity in the

roll (x) direction. Of interest is the undulation in pitch velocity present in the latter half of the drop. This undulation may represent the rising effect of air resistance on the vehicle as it gained velocity.

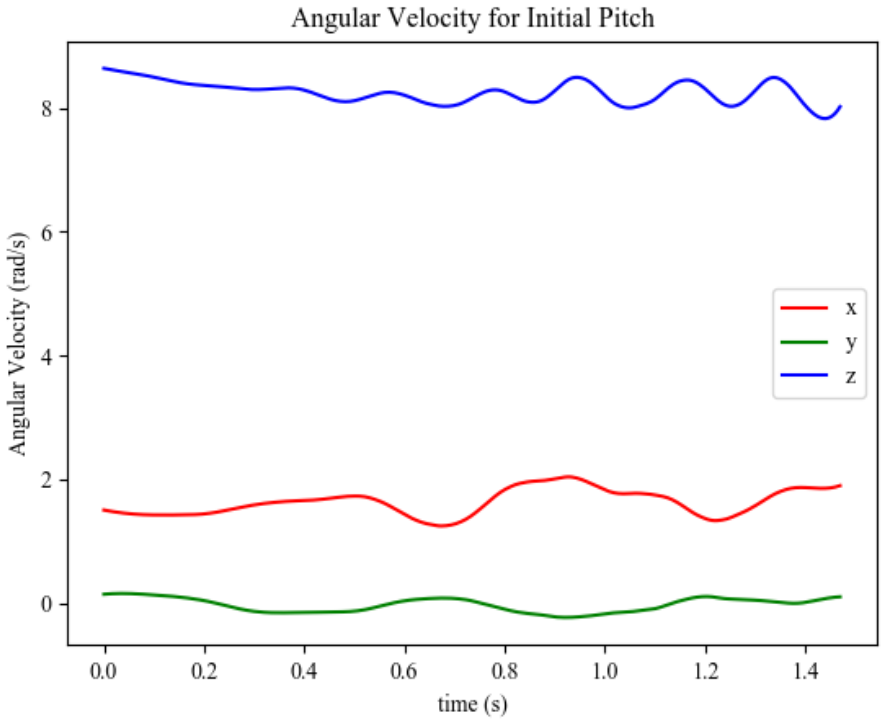

Figure 28. Angular Velocity Profile for Drop with Initial Pitch

The drops with one flap fully extended were performed with the vehicle turned about its vertical axis 180 degrees in order ensure that the gripper did not interfere with the flap as the vehicle was released. This rotation was reversed in post processing using a rotation matrix, yielding the following figure.

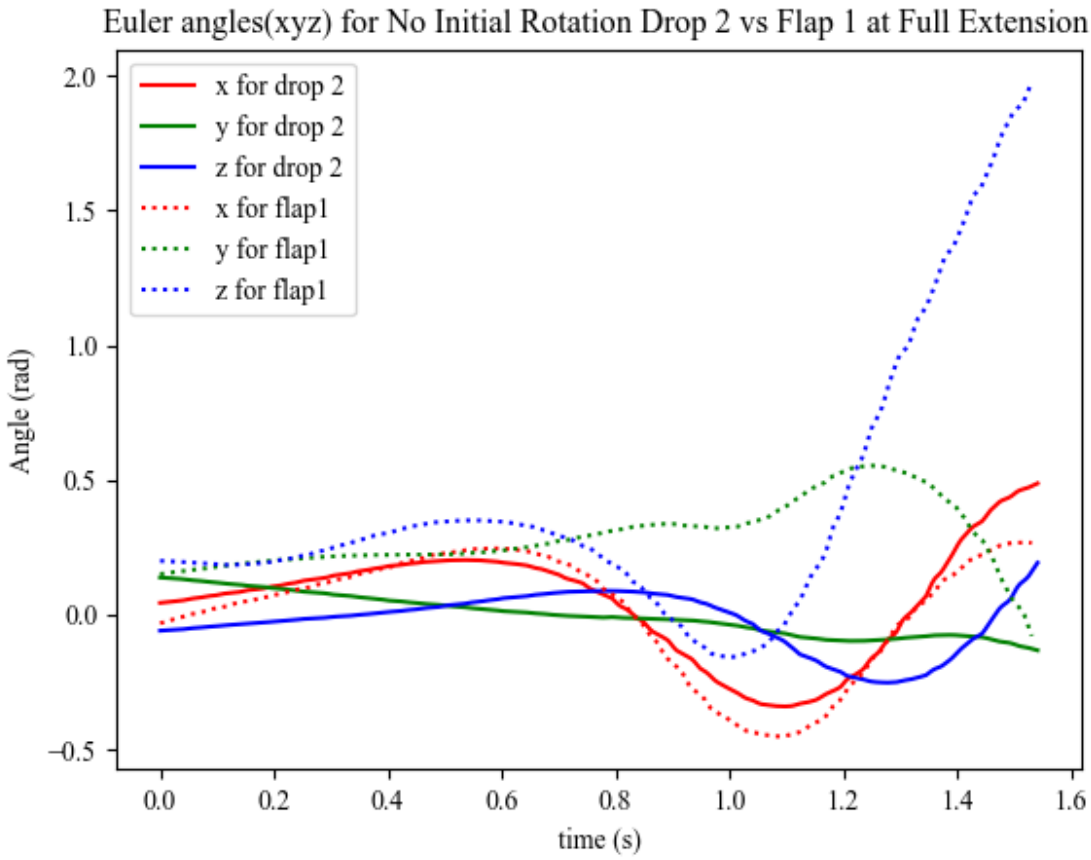

Figure 29. Euler Angles for Drop 2 vs Flap1 at Full Extension

The changes in orientation caused by the flap are clearly evident in the figure with the z-angle experiencing the most dramatic change. This was expected as the flap was extended in the x-direction, causing a change in pitch. The x angle did not exhibit much change as would be expected. However, the y angle exhibited differences because the flap was not exactly normal to airflow at all times.

Rotational Model

A rotational model was developed in order to determine the aerodynamic moments' effect on the orientation of the vehicle. These moments are related to the angular momentum of the vehicle. Angular momentum relates to the rotational motion of the body relative to its center of mass. The standard form of rigid body equations for a single rigid body with known forces are as follows:

$$m\frac{d\boldsymbol{V}}{dt} = F$$

$$\frac{d\boldsymbol{H}_G}{dt} = M_G$$

Where m, V, and F are the mass, linear velocity, and force acting on the body and H_G is the angular momentum, and M_G is the total moment about the body's center of gravity. The linear equation is equivalent to the classic Force = Mass·Acceleration equation, but instead represented as a change in linear momentum where linear momentum = M·V. Of particular interest is the angular momentum equation, wherein the change in angular momentum is equivalent to the total moment acting on the body. For more information, see Appendix B [9].

An initial angular momentum was calculated using an initial angular velocity through the inertia tensor, which was determined previously. Successive angular momenta (and angular velocities) through time could be calculated from the applied moment at every time.

For 2D problems, it is relatively easy to integrate an angular velocity into angular position. For the 3D orientations utilized in this model, a different approach was necessary. The angular velocity was first converted into a differential change in position.

This change in position was then used with Rodrigues' rotation formula, which computes the rotation matrix corresponding to an angle about a specified axis [17].

$$R_\omega = I + \tilde{\omega}\sin\theta + \tilde{\omega}^2(1-\cos\theta)$$

$$where\ \tilde{\omega} = \begin{bmatrix} 0 & -\omega_z & -\omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

I is the identity matrix, $\omega = (\omega_x , \omega_y , \omega_z)$ is the unit vector of rotation, and θ is the magnitude of that rotation. Thus, the orientation of the vehicle was determined for each time step when integrating the angular velocities.

The required inputs for this model were the vehicle orientation and angular velocities in the world frame. From these inputs, the model could represent the vehicle at all time steps, provided accurate representations of the moments experienced by the vehicle could be determined. A moment profile was derived by integrating a smoothed spline that was fit to the angular momentum data as shown below.

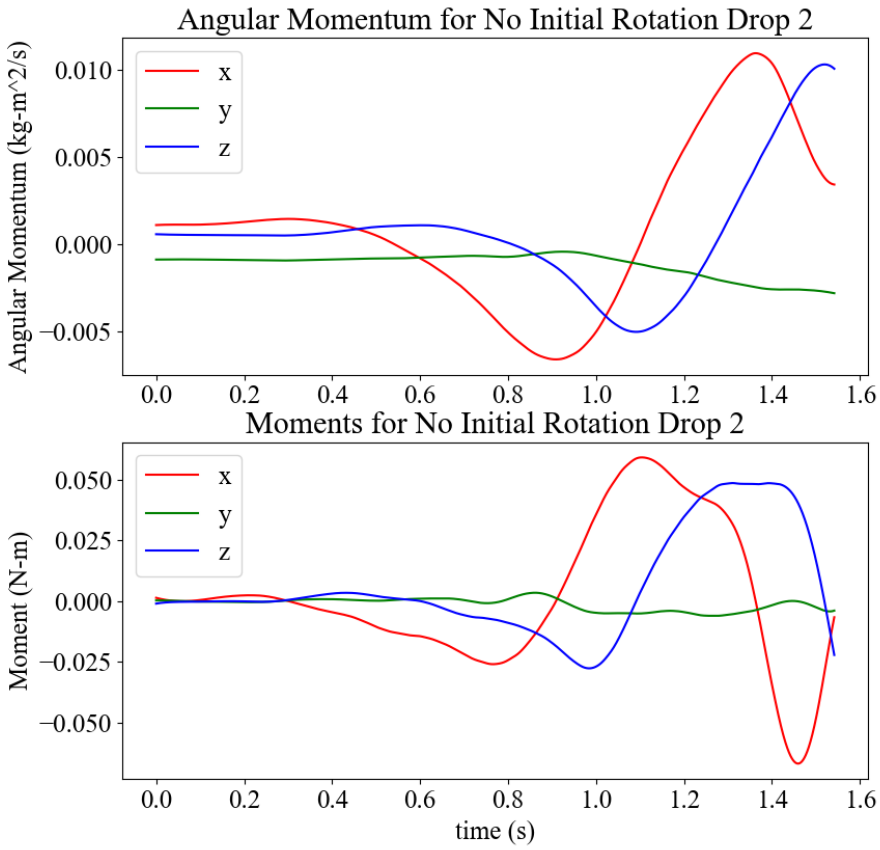

Figure 30. Angular Momentum and Moment Profile

This moment profile was inputted into the rotation model and the resulting angular velocities were compared.

.

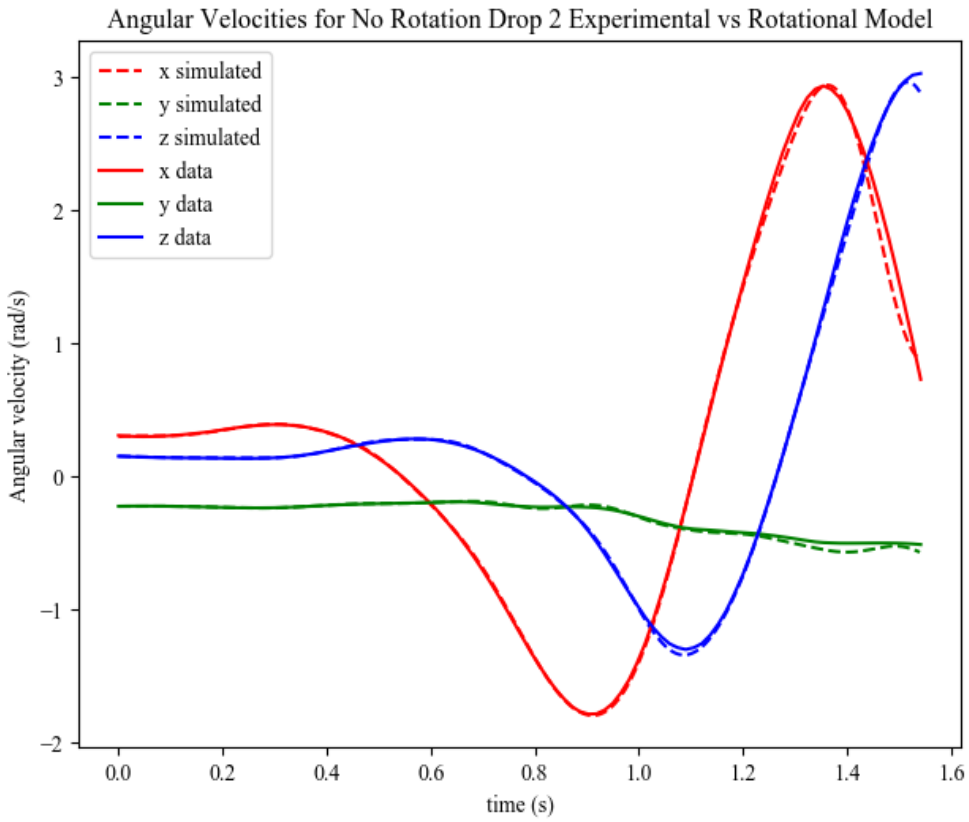

Figure 31. Angular Velocities Experimental vs Rotational Model

As seen in the figure, these angular velocities matched very well. This was expected since the model effectively integrated the moments back into angular momentums.

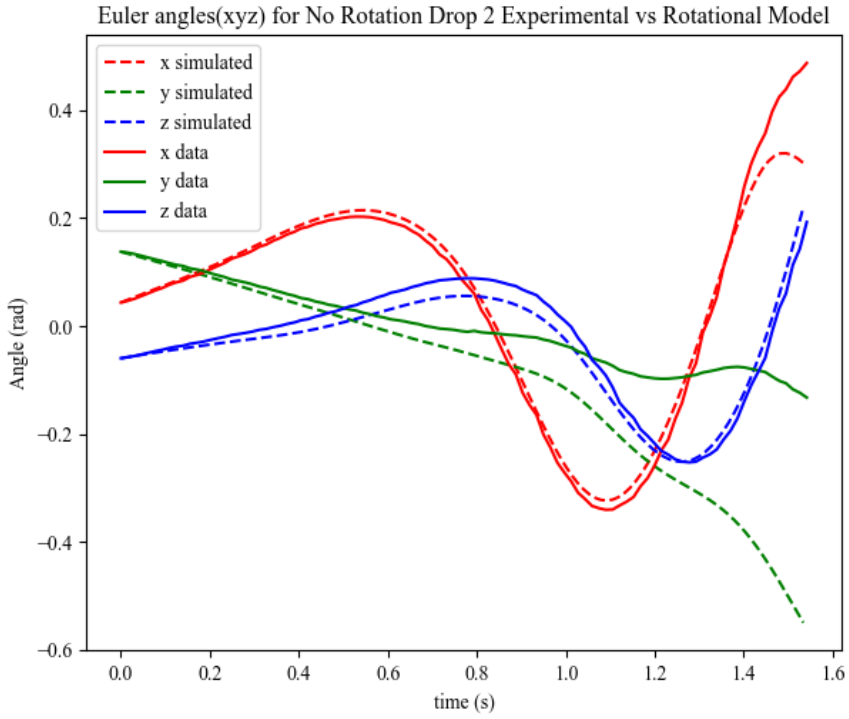

Figure 32. Euler Angles Experimental vs Rotational Model

The orientation behavior expressed by the model followed that of the experimental data in the figure above, but not exactly. This was likely because of the curve fitting of the moment profile, which was necessary in order to find the moment at every time step for integration. The error also accumulated as the integration progressed. Nevertheless, the similar behavior of the simulated rotation gave confidence to the rotational model.

Preliminary Force Model

A preliminary force model was developed for the dynamics of the vehicle. The vehicle was modeled as three flat plates experiencing air resistance as shown in the figure below.

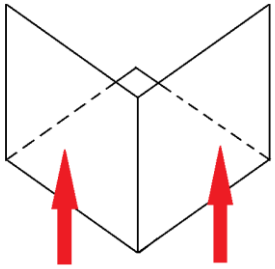

Figure 33. Three Flat Plate Model

Each face, or "plate" that faced the ground experienced an aerodynamic force. The drag and lift forces were determined from the drag and lift equations, respectively.

$$F_D = \frac{1}{2}\rho C_D A_p v^2$$

$$F_L = \frac{1}{2}\rho C_L A_p v^2$$

Where A_p refers to the projected area, C_D & C_L the drag and lift coefficients, ρ the density of air, and v the velocity.

The projected areas of the vehicle facing air resistance were needed for the aerodynamic equations. The faces of the vehicle that faced the ground at any orientation were found with an "AirChecker" function. The function accomplished this by finding the directional cosines of the given orientation with respect to a downward gravity unit vector. A truth table was then compiled that specified which faces oriented toward the ground. These faces were projected onto a plane parallel to the ground. These projected areas were calculated and the position vectors to their centroids were determined as well.

The velocities for the aerodynamic equations were found utilizing the freefall equation with quadratic air resistance [16].

$$y = \frac{(v_{terminal})^2}{g} \ln\left[\cosh\left(\frac{gt}{v_{terminal}}\right)\right]$$

To verify the validity of these velocities, the equation was plotted and compared to the integration of the gathered accelerometer data as shown below.

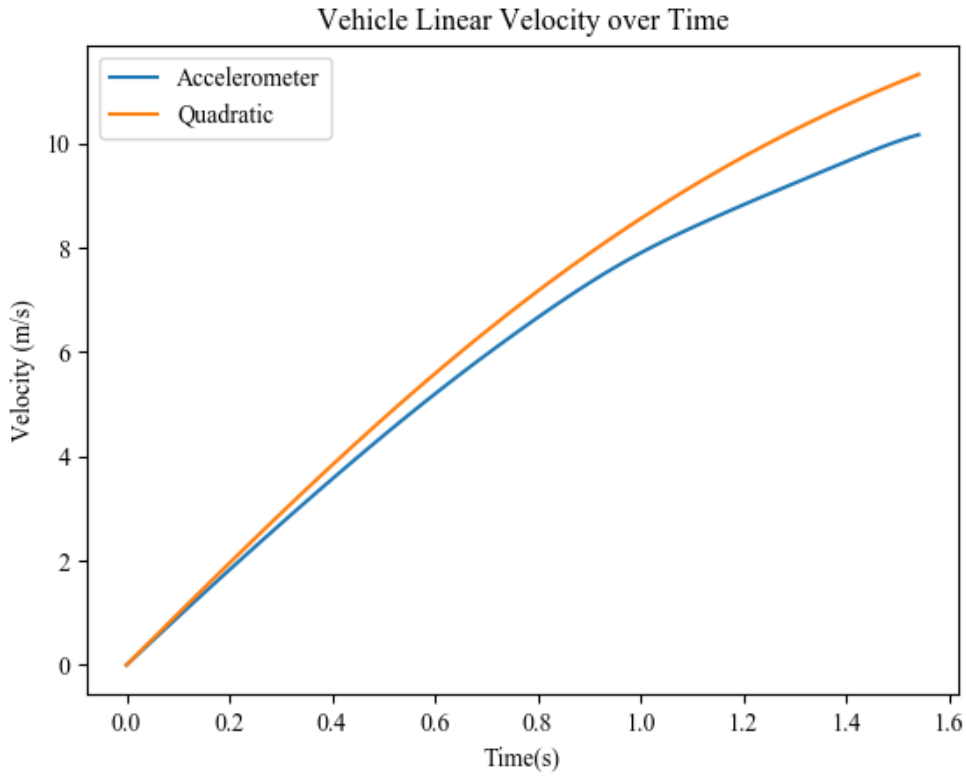

Figure 34. Accelerometer Data vs Quadratic Resistance Model

As shown in the figure, the velocities exhibited very similar behavior. The velocities were then further integrated to obtain the distance traveled. The accelerometer data resulted in a drop distance of about 30.12 ft while the quadratic resistance model had a drop distance of 32.9 ft. This model of velocity over time was found to most closely match with physical behavior since the vehicle was dropped from a roughly 36 foot tall parking structure on to a 2'-3" raised net platform. The differences in freefall velocities

may be attributed to the accumulation of accelerometer error as well as the compounding of this error with integration.

For every orientation in a time step, the projected areas and centroids, the forces and moments were determined. The determined moments were input into the rotational model, resulting in the angular velocity profile shown in the figure below.

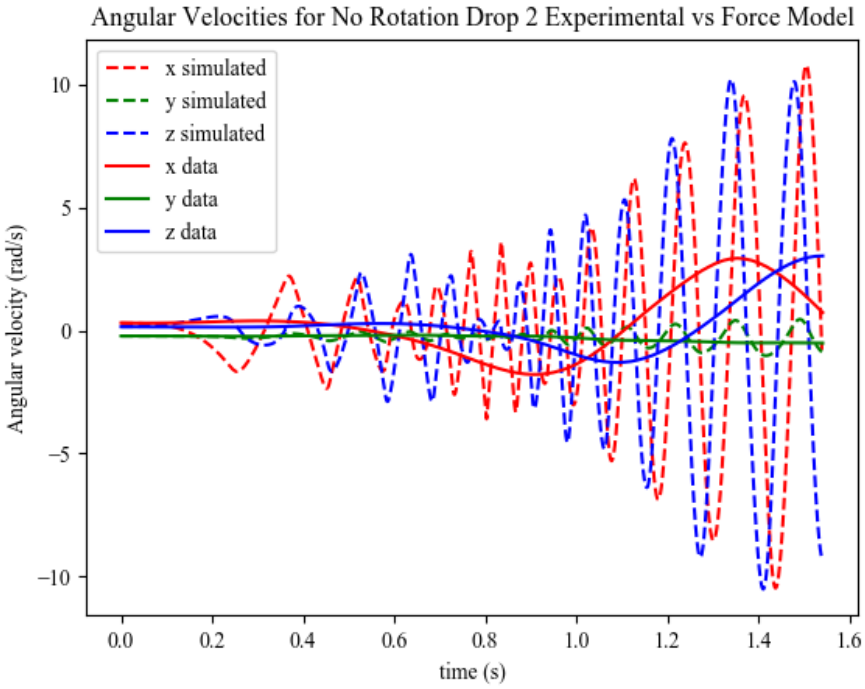

Figure 35. Angular Velocity Experimental vs Force Model

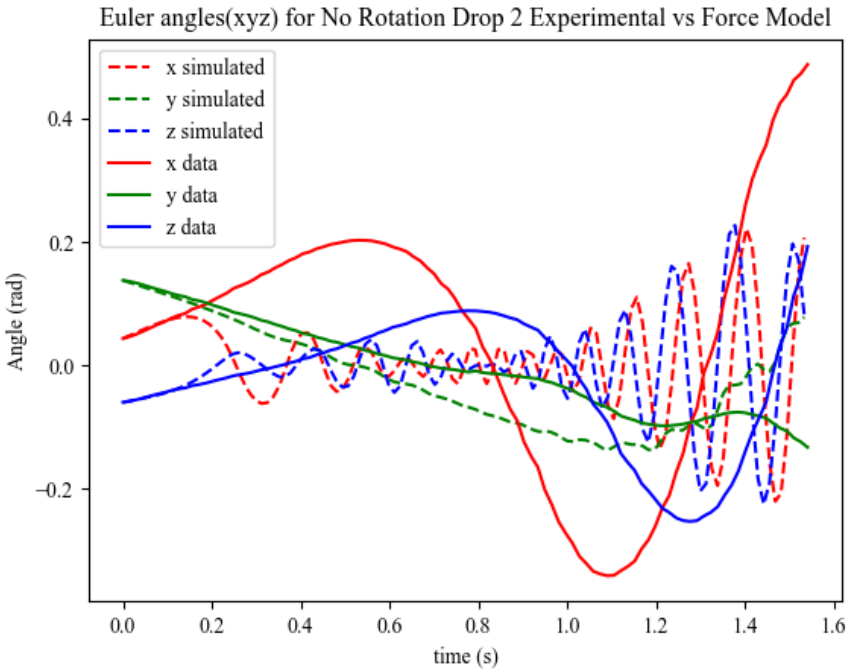

Figure 36. Euler Angles Experimental vs Force Model

The sharp changes in angular velocity in this model resulted in oscillations of the orientation as shown in the figure above. The rapid movements of the vehicle represented relatively large forces on the vehicle surfaces as it changed orientation.

This instability in the simulation most likely resulted from the simplification of aerodynamic forces on the vehicle due to the assumption of constant lift and drag coefficients for the three plates. This model could be refined by determining how these coefficients change in time.

CONCLUSION

The behavior of an object as it experienced freefall was obtained. In order to properly model this behavior, a physical vehicle was created. A method of identifying the inertia tensor of the vehicle was examined and utilized in order create a suitable model.

Data was successfully collected for various initial conditions. External apparatuses were designed for the use of this vehicle for repeatability and minimization of hazards. The data collected was transformed into a common reference frame for proper analysis. A rotational model describing the effect of aerodynamic moments on the vehicle was developed and agreed well with experimental data. A preliminary force model describing the effect of moments on the vehicle orientation was also created, but exhibited extensive disagreement with the experimental data. Aerodynamic moments are nonlinear functions and this disagreement most likely resulted from an oversimplification of the interactions involved.

REFERENCES

[1] Woodman, Oliver J. (2007). An introduction to inertial navigation. University of Cambridge Computer Laboratory.

[2] Myint Aye, Ma Myint. (2011). Analysis of Euler Angles in a Simple Two-Axis Gimbals Set. World Academy of Science, Engineering, and Technology.

[3] Slabaugh, Gregory G. (1999). Computing Euler angles from a rotation matrix.

[4] G¨una¸stı, G¨okmen (2012). Quaternions Algebra, Their Applications in Rotations and Beyond Quaternions. Linnaeus University.

[5] Dam, E. Koch, M. Lillholm, M. (1998). Quaternions, Interpolation and Animation. Technical Report DIKU-TR-98/5. University of Copenhagen.

[6] Long, L. Weiss, H. The Velocity Dependence of Aerodynamic Drag: A Primer for Mathematicians. *The American Mathematical Monthly*, Vol. 106, No. 2 (Feb., 1999), pp. 127-135

[7] "Hobby Servo Motor Tutorial". Robot Platform. 7 June 2018, http://www.robotplatform.com/knowledge/servo/servo_tutorial.html

[8] Beer, Ferdinand P. Vector Mechanics for Engineers: Statics and Dynamics. McGraw-Hill Higher Education, 2009.

[9] Gregory, R. Douglas. *Classical Mechanics.* Cambridge University Press 2006.

[10] Premerlani W, Bizard P (2009) Direction Cosine Matrix IMU: Theory, DIY DRONE: USA.

[11] Shakoori, A. Betin, A.V. Betin, D.A. Comparison of Three Methods to Determine the Inertial Properties of Free-Flying Dynamically Similar Models. *Journal of Engineering Science and Technology*, Vol. 11 No. 10 (2016) 1360-1372

[12] Spong, Mark W., et al. *Robot Modeling and Control*. Wiley, 2006.

[13] Tang, L. Shangguan, W. An improved pendulum method for the determination of the center of gravity and inertia tensor for irregular-shaped bodies. *Measurement,* 44(2011) 1849-1859

[14] Hall, Nancy. "Aerodynamic Center." *NASA*, NASA, 5 May 2015, www.grc.nasa.gov/WWW/K-12/airplane/ac.html.

[15] Lohninger, H. "Savitzky-Golay Filter." *Savitzky-Golay Filter*, 8 Oct. 2012, www.statistics4u.info/fundstat_eng/cc_filter_savgolay.html.

[16] Taylor, John Robert. *Classical Mechanics*. University Science Books, 2005.

[17] Belongie, Serge. "Rodrigues' Rotation Formula." From *MathWorld*--A Wolfram Web Resource, created by Eric W. Weisstein.

http://mathworld.wolfram.com/RodriguesRotationFormula.html

[18] Friedland, Bernard. *Control System Design: an Introduction to State-Space Methods*. Dover Publications, 2005.

APPENDIX A – FUTURE WORK

For this effort, the behavior of a falling object was captured and utilized to create a preliminary mathematical model. In order to complete the overall objectives of this project, the fluid mechanics involved in this behavior must be understood in order to provide a realistic force model that captures the behavior of the vehicle faces and the flaps. This may be done through numerical methods such as computational fluid dynamics (CFD). Once the force model is known, a control law can be developed. One method of control can be utilized through a state space representation. These two efforts are touched on in the following sections.

Rudimentary CFD Analysis

In a separate effort to capture the aerodynamic moments experienced by the vehicle, a rudimentary CFD analysis was performed using Solidworks Flow Simulation.

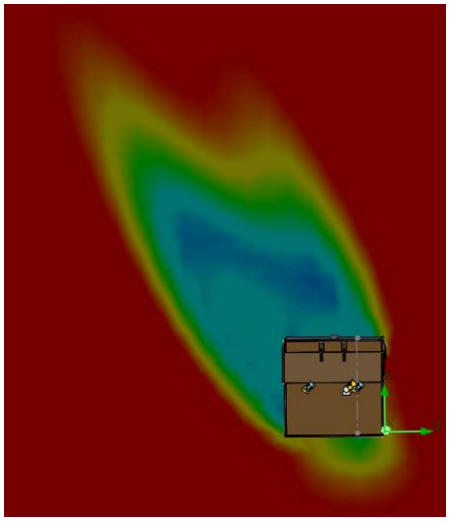

Figure 37. Solidworks Flow Simulation

The Flow Simulation used the body frame for its calculations. As the vehicle rotated in space, it experienced the air velocity from different directions. Therefore, the

air velocity was rotated into the body frame through the logged orientations. These velocities were inputted into the ambient conditions of the simulation for each time step considered. The simulation was run 50 times in order to capture the moments. These moments were then rotated back into the world frame.

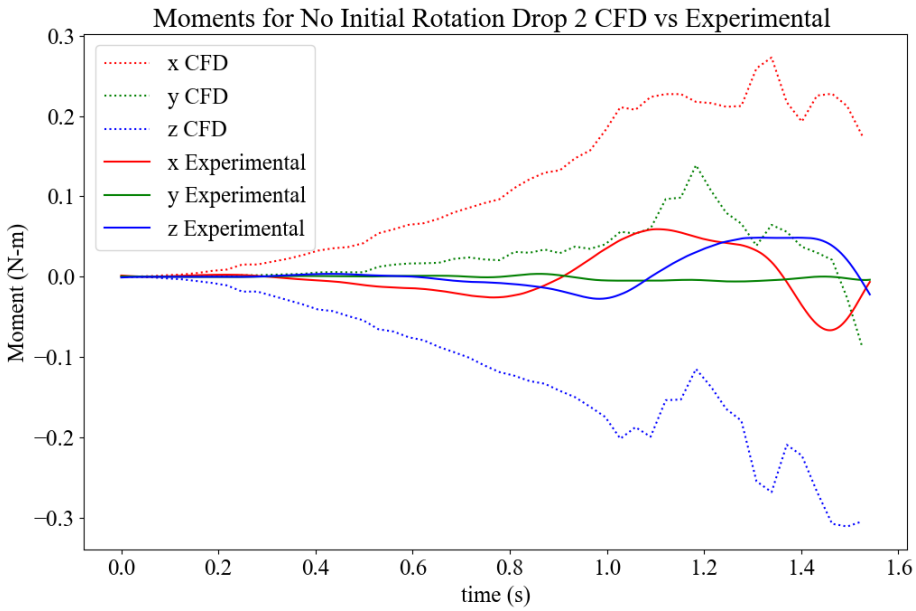

Figure 38. CFD Moment Profile

Unfortunately, the moment profile produced from this method did not resemble the moment profile from the experimental data. This may be because the simulation did not take into account that the vehicle had some angular velocity at each point in time. Future work could involve simulating the rotating vehicle as it falls through the air.

State Space Control

Assuming that the body axes are in line with the principal axes of rotation, the dynamics of the vehicle can be expressed as follows:

$$\dot{p} = \frac{L}{J_x} - \frac{J_z - J_y}{J_x} qr$$

$$\dot{q} = \frac{M}{J_y} - \frac{J_x - J_z}{J_y} pr$$

$$\dot{r} = \frac{N}{J_z} - \frac{J_y - J_x}{J_z} pq$$

$$\dot{u} = rv - qw + \frac{f_{xB}}{m}$$

$$\dot{v} = -ru + pw + \frac{f_{yB}}{m}$$

$$\dot{w} = qu - pv + \frac{f_{zB}}{m}$$

Where p, q, and r refer to the roll, pitch, and yaw and u,v, and w refer to the projections of the velocity vector on the body axes. L, M, and N, are the changes in aerodynamic moments in roll, pitch, and yaw and f refers to the forces on the body. The aerodynamic moments and forces all depend on the dynamic pressure

$$Q = \frac{1}{2}\rho V^2$$

$$V = (u^2 + v^2 + w^2)^{1/2}$$

Where ρ is the air density and V is is the speed of the object. Thus, the aerodynamic forces and moments can be expressed in the following form:

$$f_x = QAC_x$$

$$f_y = QAC_y$$

$$f_z = QAC_z$$

$$L = lQAC_L$$

$$M = lQAC_M$$

$$N = lQAC_N$$

C refers to dimensionless aerodynamic coefficients, A is a reference area, and l is a reference length. These aerodynamic coefficients are functions of the object velocity components and the deflections of movable control surfaces from their positions of reference.

Assuming that the speed of the vehicle is approximately constant, the u and w components of the velocity vector can be replaced with state variables denoting the direction of the velocity vector relative to the vehicle axes. Let α be the angle of the vehicle face with respect to the velocity vector in the pitch direction and β the angle of the vehicle face with respect that in the roll direction.

$$\dot{\alpha} = \tan^{-1}\left(\frac{w}{v}\right) \simeq \frac{w}{v}$$

$$\dot{\beta} = \tan^{-1}\left(\frac{u}{v}\right) \simeq \frac{u}{v}$$

For many aircraft control systems, the dynamics are often linearized about some operating condition or “flight regime” where it is assumed that velocity and attitude are constant [18]. Control surfaces are set to these conditions and the control system is designed to correct small motions.

Therefore the equations can be simplified:

$$\dot{p} = \frac{L(p, q, r, \Delta v, \alpha, \beta, deflections)}{J_x}$$

$$\dot{q} = \frac{M(p, q, r, \Delta v, \alpha, \beta, deflections)}{J_y}$$

$$\dot{r} = \frac{N(p, q, r, \Delta v, \alpha, \beta, deflections)}{J_z}$$

$$\Delta\dot{v} = Y(\Delta v, \alpha, \beta, deflections)$$

$$\dot{\alpha} = -q + Z(\Delta v, q, \alpha, \beta, deflections)/\mathrm{V}$$

$$\dot{\beta} = -r + X(\Delta v, r, \alpha, \beta, deflections)/\mathrm{V}$$

Applying a Taylor's series expansion on the right hand sides and ignoring the higher order terms, the system is able to be linearized about the constant velocity and attitude.

$$\dot{p} = \frac{1}{J_x}\left(L_p p + L_q q + L_r r + L_{\Delta v}\Delta v + L_\alpha \alpha + L_\alpha \beta + L_{\psi_n}\psi_n\right)$$

$$\dot{q} = \frac{1}{J_y}\left(M_p p + M_q q + M_r r + M_{\Delta v}\Delta v + M_\alpha \alpha + M_\alpha \beta + M_{\psi_n}\psi_n\right)$$

$$\dot{r} = \frac{1}{J_z}\left(N_p p + N_q q + N_r r + N_{\Delta v}\Delta v + N_\alpha \alpha + N_\alpha \beta + N_{\psi_n}\psi_n\right)$$

$$\Delta\dot{v} = Y_{\Delta v}\Delta v + Y_\alpha \alpha + Y_\alpha \beta + Y_{\psi_n}\psi_n$$

$$\dot{\alpha} = \frac{Z_p}{V}p + \frac{Z_q}{V}q - q + \frac{Z_r}{V}r + \frac{Z_{\Delta v}}{V}\Delta v + \frac{Z_p}{V}\alpha + \frac{Z_p}{V}\beta + \frac{Z_{\psi_n}}{V}\psi_n$$

$$\dot{B} = \frac{X_p}{V}p + \frac{X_q}{V}q + \frac{X_r}{V}r - r + \frac{X_{\Delta v}}{V}\Delta v + \frac{X_p}{V}\alpha + \frac{X_p}{V}\beta + \frac{X_{\psi_n}}{V}\psi_n$$

Where ψ_n refers to flap deflection for flap n.

The state variables are thus chosen as the following:

$$\underline{\boldsymbol{x_1 = p}}$$

$$x_2 = \dot{x_1}$$

$$\underline{\boldsymbol{x_3 = q}}$$

$$x_4 = \dot{x_3}$$

$$\underline{\boldsymbol{x_5 = r}}$$

$$x_6 = \dot{x_5}$$

$$\underline{\boldsymbol{x_7 = \Delta v}}$$

$$x_8 = \dot{x_7}$$

$$\underline{\boldsymbol{x_9 = \alpha}}$$

$$x_{10} = \dot{x_9}$$

$$\underline{\boldsymbol{x_{11} = \beta}}$$

$$x_{12} = \dot{x_{11}}$$

The state equation is then:

$$\begin{bmatrix} x_2 \\ x_4 \\ x_6 \\ x_8 \\ x_{10} \\ x_{12} \end{bmatrix} = \begin{bmatrix} L_p & L_q & L_r & L_{\Delta v} & L_\alpha & L_\beta \\ M_p & M_q & M_r & M_{\Delta v} & M_\alpha & M_\beta \\ N_p & N_q & N_r & N_{\Delta v} & N_\alpha & N_\beta \\ 0 & 0 & 0 & Y_{\Delta v} & Y_\alpha & Y_\beta \\ \frac{Z_p}{V} - 1 & \frac{Z_q}{V} & 0 & 0 & \frac{Z_\alpha}{V} & 0 \\ \frac{X_p}{V} & \frac{X_q}{V} - 1 & 0 & 0 & 0 & \frac{X_\beta}{V} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_3 \\ x_5 \\ x_7 \\ x_9 \\ x_{11} \end{bmatrix} + \begin{bmatrix} L_{\psi 1} & L_{\psi 2} & L_{\psi 3} & L_{\psi 4} \\ M_{\psi 1} & M_{\psi 2} & M_{\psi 3} & M_{\psi 4} \\ N_{\psi 1} & N_{\psi 2} & N_{\psi 3} & N_{\psi 4} \\ Y_{\psi 1} & Y_{\psi 2} & Y_{\psi 3} & Y_{\psi 4} \\ \frac{Z_{\psi 1}}{V} & \frac{Z_{\psi 2}}{V} & \frac{Z_{\psi 3}}{V} & \frac{Z_{\psi 4}}{V} \\ \frac{X_{\psi 1}}{V} & \frac{X_{\psi 2}}{V} & \frac{X_{\psi 3}}{V} & \frac{X_{\psi 4}}{V} \end{bmatrix} \cdot \begin{bmatrix} \psi_1 \\ \psi_1 \\ \psi_3 \\ \psi_4 \end{bmatrix}$$

APPENDIX B - LITERATURE REVIEW

IMU (Inertial Measurement Unit)

A common problem in navigation is the tracking of the orientation and position of an object relative to a known starting point. Inertial navigation is a technique which uses accelerometers and gyroscopes to accomplish this. An inertial measurement unit (IMU) is composed of three orthogonal rate-gyroscopes and three orthogonal accelerometers in order to make this possible. From the acceleration and angular velocity provided by an IMU, the orientation and position can be obtained via integration. However, error can accumulate during these integrations, especially when integrating the acceleration twice. Therefore, the raw data is often processed along with magnetometer data in order to achieve accurate readings. Most IMUs are categorized as either stable platform or strapdown. The frame of reference that a device uses determines which type it is. The space in which the device is navigating in is defined as the global frame, while the navigation system's frame of reference is defined as the body frame. In stable platform systems, sensors are mounted on a platform that is not allowed any external rotation. The platform is continuously aligned with the global frame by mounting the platform to a gimbal system which allow three axes of freedom. The gyroscopes on the platform detect rotation and utilize motors to negate those rotations and keep the platform aligned with the global frame. Orientation can be determined using angle pick-offs. In a strapdown system, sensors are fixed onto the device and output data in the body frame. From this data, the desired orientation and position can be achieved by the action of external actuators. A stable platform system tends to be physically larger and mechanically complex than strapdown systems. Still, strapdown systems have the drawback of

requiring more complex computation. Strapdown systems have become the dominant type because of the decreasing cost of these computations [1].

Representing Orientation

Two ways of representing orientations are Euler angles and Quaternions. Euler angles represent any orientation in terms of three rotations. Three axes are commonly used to define 3-D space [3]. Thus, conventions are used with Euler angles in order to obtain repeatable results. One Euler angle convention is the Z-X'-Z'' convention. This involves a rotation of φ about the z axis, another by θ about the new x axis, and finally by ψ about the new z axis [2].This convention is known as an intrinsic convention, where rotations are performed with respect to the transformed coordinate axes. This can be interpreted as fixing the world and rotating the object. Conversely, an extrinsic convention involves fixing the object and rotating the world around it. An example of an extrinsic convention would be a rotation about the x-axis, y-axis, and z-axis while ignoring the transformed axes (also known as the xyz-convention) [3].

Euler angle rotations are applied using a 3-dimensional rotation matrix that is composed of three independent rotations that result in the desired rotation. If a rotation is applied about the x, y, and z axes by Euler angles of φ, θ, and ψ, respectively, the independent rotations matrices are as follows:

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

However, the sequence of rotations is important when composing the rotation matrix. This is a consequence of both the nature of matrix multiplication and of rotation.

As an example, utilizing the xyz convention mentioned previously, the product of these matrices is:

$$R = R_z(\psi)R_y(\theta)R_x(\varphi)$$

$$= \begin{bmatrix} \cos\theta\cos\psi & -\cos\varphi\sin\psi + \sin\varphi\sin\theta\cos\psi & \sin\varphi\sin\psi + \cos\varphi\sin\theta\cos\psi \\ \cos\theta\sin\psi & -\cos\varphi\cos\psi + \sin\varphi\sin\theta\sin\psi & \sin\varphi\cos\psi + \cos\varphi\sin\theta\sin\psi \\ -\sin\theta & \sin\varphi\cos\theta & \cos\varphi\cos\theta \end{bmatrix}$$

If one were to use a different convention, the values of the Euler angles would not be the same. If given a rotation matrix, the Euler angles can also be computed be equating each element of the rotation matrix with its corresponding element within the product matrix. This results in a system of equations that can be solved for the Euler angles. [3].

Though Euler angles are intuitive in a sense, they also encounter problems in certain applications. One of these is gimbal lock. In matrices, the phenomenon of gimbal lock is identified with mathematical singularities. Physically, gimbal lock involves the alignment of two rotational axes such that a degree of freedom is lost. Thus, certain orientations can become problematic when dealing with Euler angles. Furthermore, representing Euler angles mathematically can become tiresome [4]. The Euler angles could be derived earlier because of the chosen convention. However, any Euler Angles derived from a rotation matrix are ambiguous due to the variety of conventions available. Furthermore, a rotation may be represented by many different rotation matrices, which increases the ambiguity. Interpolation is also difficult because normally the three axes are interpolated independently. This ignores the interdependency between the axes, and results in a less direct path. The object is forced to rotate a certain amount about each axis instead of rotating to the end orientation directly [5].

Quaternions were invented by Sir William Hamilton as a way to represent complex numbers in the 3rd dimension. A part of this motivation was to find a description of a rotation in space that corresponded to complex numbers, wherein a multiplication corresponded to a scaling and a rotation. Quaternions involve imaginary units that satisfy the following conditions:

$$i^2 = j^2 = k^2 = ijk = -1$$

A quaternion is composed of a scalar part and a vector part:

$$q = s + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

Quaternions represent rotation as a rotation angle about a rotation axis. The mapping between rotations and quaternions are thus unambiguous with the exception that every rotation can be represented by two quaternions. A specific rotation can be achieved by rotating in the opposite direction about the opposite axis. In practical use, only unit quaternions are used to represent pure rotation (therefore no longer scaling via multiplication). A unit quaternion is a quaternion whose norm is equal to 1:

$$\|q\| = 1$$

Thus the set of unit quaternions comprises a unit sphere in four-dimensional space. Since gimbal lock occurs by the nature of Euler angles, quaternions avoid this completely. Computing successive rotations is faster computationally and more stable numerically than Euler angles. With quaternions, it is simpler to extract the axis and angle of rotation. Furthermore, interpolation from one orientation to another is more straightforward than with Euler angles [5].

Gyroscopes

The conventional mechanical gyroscope utilizes a spinning mass mounted on two gimbals. The spinning mass will resist changes in orientation as a consequence of the conservation of angular momentum. When this gyroscope is rotated, the orientation of the wheel will remain the same and the angles between adjacent gimbals will change. Orientation can be measured using the angles between these adjacent gimbals. While useful for orientation, mechanical gyroscopes suffer from multiple disadvantages. One of which is that they are composed of moving parts whose friction causes drift over time. Reducing this friction is costly and mechanical gyroscopes also require a few minutes to warm up.

Whereas the mechanical gyroscope measures orientation, most modern gyroscopes measure angular velocity. Optical gyroscopes operate based on the Sagnac effect. Two light beams are fired in opposite directions. If rotated, the beam in the direction of rotation will have a longer path than the beam against the direction of rotation. Optical gyroscopes only require a few seconds to start and have no moving parts, which are significant advantages over mechanical gyroscopes. However, the accuracy of gyroscopes are constrained by the length of their light transmission path and therefore their size. Furthermore, these gyroscopes are very costly compared to MEMS gyroscopes.

MEMS sensors have low part counts and are relatively cheap to manufacture. They operate using the Coriolis effect. In a reference frame rotating at an angular velocity, a mass moving with a certain velocity experiences a force that is a function of these three components. Vibrating elements are used in MEMS gyroscope to measure the

Coriolis effect. The simplest vibrating element geometry uses a single mass which vibrates along a drive axis. When the gyroscope is rotated, a secondary vibration is created along the perpendicular sense axis. The angular velocity can then be measured. MEMS gyroscopes are far less accurate than optical gyroscopes, but they have many advantages. MEMS gyroscopes are smaller, consume less power, and have shorter start up times [1].

Error in MEMS Gyroscopes

MEMS gyroscopes produce an orientation signal that is integrated (from angular velocity) and have various sources of error. When the gyroscope is not undergoing any rotation, it nevertheless produces an output. The bias of a rate gyroscope is the average of this output. When a constant bias error is integrated, the orientation produced has an angular error which grows linearly with time. However, this error can be compensated by subtracting the bias from the output once the constant bias error of the rate gyroscope is known. This involves taking a long term average of the gyroscope's output while it is not being rotated. The output of a MEMS gyroscope is disturbed by thermo-mechanical noise that fluctuates at a rate much greater than the sensor can sample. The noise introduces a zero-mean random walk error into the integrated signal with a standard deviation that is proportional to the square root of time. Manufacturers commonly specify this noise in terms of the integrated signal using angle random walk (ARW) that has units $degrees/\sqrt{h}$. This gives the standard deviation of the orientation error after *h* hours. Noise can be specified using power spectral density and FFT noise density. Flicker noise causes the bias of a MEMS gyroscope to wander. It is observed usually at low frequencies in electronic components and tends to be overshadowed by white noise at higher frequencies.

Bias stability shows how the bias of a device can change over a specified period of time (typically around 100 seconds) in fixed conditions, usually with constant temperature. Using the random walk model, if there is a known bias at time t, a 1σ bias stability of 0.01°/h over 100 seconds means that the bias after 100 seconds is a variable whose value should be the known bias with standard deviation 0.01°/hr. Practically, this

effect on orientation is constrained within some range and this random walk model is only a good approximation of what happens for short periods of time. Changes in environment and sensor self-heating cause temperature fluctuations that affect the bias. These effects are not included in the bias stability measurements obtained under fixed conditions. Residual bias due to temperature fluctuations cause an error which grows linearly with time. Often, the relationship between bias and temperature is highly nonlinear for MEMS sensors. To correct for bias caused by temperature, most inertial measurement units (IMUs) contain internal temperature sensors.

Errors due to calibration are usually measurable and correctable. These errors usually produce bias errors that are only seen while the gyroscope is turning. The errors lead to the accumulation of more drift in the integrated signal. The magnitude of this signal is proportional to the rate and duration of the motions involved. Different gyroscopes have different sources of error. In MEMS gyroscopes, noise errors and uncorrected bias errors are usually the most important sources of error, either from uncorrected temperature fluctuations or an error in initial bias estimation. The angle random walk can be used as a lower bound for the uncertainty found when integrating a rate gyroscope's signal [1].

Aerodynamic Drag

The modeling of drag is often more empirical than mathematical. Usually it relies on the results of many wind-tunnel experiments. There is a significant amount of theoretical work, but there are large gaps in the understanding of the basic properties of the Navier-Stokes equation (which can describe the motion of a body moving through a fluid). In air, the drag force on an object can be well approximated by a force that is proportional to the square of the velocity. This leads to a nonlinear equation of motion for the falling body. The drag force on a body depends on many factors involving both the fluid and the body. The geometry, velocity, and surface material of the body play a role as well as the density and viscosity of the fluid. The Reynolds number is important in finding the drag force and depends on: the density (ρ) and viscosity (μ) of the fluid, the velocity of the fluid (v), and a characteristic length of the body (d):

$$R = \frac{\rho d v}{\mu}$$

Laminar flow is usually associated with small Reynolds numbers while turbulent flow is usually associated with large Reynolds numbers. If the Reynolds number is small (much less than 1), then the Navier-Stokes equation can be considerably simplified and the equation of motion is reduced to a linear partial differential equation, also known as creeping flow. Flow may remain laminar when the Reynolds number is large, but not too large. This can be studied using the Navier-Stokes equation in the thin boundary layer where the flow is assumed to be laminar. The resulting equations are named Prandtl's equations, which lead to the conclusion that the drag force is independent of viscosity at least for a certain range of Reynolds numbers. For these Reynolds numbers, the dimensionless quantity known as the drag coefficient can be utilized.

$$C_D = \frac{F_D}{0.5\rho A v^2}$$

Where C_D is the drag coefficient, F_D is the drag force, ρ is the density of the fluid, A is the projected area, and v is the velocity. This coefficient depends on the shape and surface characteristics of the body and the Reynolds number [6].

Servos

Servo motors have gained popularity in the robotics community because of their affordability, simplicity, and ease of use. They are rather easy to control and do not require external H-bridges or complicated external circuitry. The "spline" of a servo is the output shaft that connects the servo to whatever it is driving. The motor in a servo is usually a three pole ferrite DC motor. Servos contain gear sets that reduce the speed and increase the torque of the output shaft. The transit time of a servo is usually measured as the time it takes to move the output shaft 60 degrees from its current position. The gears of servos are commonly seen to be made of either nylon, metal, or karbonite. Nylon gears are cheap and light, but produce little strength. Metal gears have far more torque, but wear quickly and loose accuracy. Karbonite is a kind of reinforced plastic that is far stronger than nylon gears, but is also far more expensive. The control board of a servo controls the position of the motor and has an integrated H-bridge for counter-rotation. The potentiometer of a servo monitors the position of the output shaft and is often the first thing to fail in a servo. Most manufacturers used the same closed loop design for servos. A modulated square-wave pulse is used to control a servo. This is known as pulse-width-modulation (PWM) or pulse-duration-modulation (PDM). Assuming appropriate signal voltage, the factors to be considered are the frequency and the duty cycle of the servo. In order to hold or move position, a servo expects continuous pulses. The frequency of the servo is the number of times a positive pulse is sent to the servo in a unit time. The duty cycle is the width of the square wave and an important factor for the servo's angular position [7]. An example would be 1ms for the minimal angle, 1.5ms for the central angle, and 2ms for the maximum angle.

Angular Momentum of a Rigid Body

The angular momentum of a body about its center of mass can be determined from the angular velocity of the body in three-dimensional motion. The operation that transforms the angular velocity vector to the angular momentum vector is specified by the inertia tensor of the body. This 3x3 matrix is composed of moments and products of inertia.

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

I_{xx} , I_{yy}, and I_{zz} are referred to as the moments of inertia, and I_{xy} , I_{yz}, and I_{xz} are referred to as the products of inertia. If one were to use a different system of axes, this matrix would change, but the resulting transformation would remain the same.

The moments of inertia of common shapes are well-defined. If a composite body is composed of these simple shapes, it is relatively easy to determine its moment of inertia. The moments of inertia of the component parts about the desired axis must first be calculated and can then be summed to determine the moment of inertia of the composite body.

Principal axes of inertia can be selected such that all the products of inertia of a given body are zero. This results in a diagonalized matrix with three moments of inertia which are known as the principal moments of inertia. If these moments are equivalent, then the angular momentum ($\vec{h}$) is proportional to the angular velocity and their respective vectors are collinear. This can be represented by:

$$\vec{h} = \begin{bmatrix} J_x\omega_x \\ J_y\omega_y \\ J_z\omega_z \end{bmatrix}$$

However, most inertia tensors are not in this form in most reference frames This results in different directions between the angular velocity and angular momentum. They are only the same when two of the three components of the angular velocity are zero. This can be interpreted as when the angular velocity is directed along one of the coordinate axes. Therefore, the angular momentum of a rigid body and its angular velocity will have the same direction if, and only if, the angular velocity is directed along a principal axis of inertia [8].

APPENDIX C – VEHICLE CODE

Vehicle Calibration

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <EEPROM.h>


/* This driver uses the Adafruit unified sensor library (Adafruit_Sensor),
   which provides a common 'type' for sensor data and some helper functions.

   To use this driver you will also need to download the Adafruit_Sensor
   library and include it in your libraries folder.

   You should also assign a unique ID to this sensor for use with
   the Adafruit Sensor API so that you can identify this particular
   sensor in any data logs, etc.  To assign a unique ID, simply
   provide an appropriate value in the constructor below (12345
   is used by default in this example).

   Connections
   ===========
   Connect SCL to analog 5
   Connect SDA to analog 4
   Connect VDD to 3-5V DC
   Connect GROUND to common ground

   History
   =======
   2015/MAR/03  - First release (KTOWN)
   2015/AUG/27  - Added calibration and system status helpers
   2015/NOV/13  - Added calibration save and restore
   */

/* Set the delay between fresh samples */
#define BNO055_SAMPLERATE_DELAY_MS (100)

Adafruit_BNO055 bno = Adafruit_BNO055(55);

/**************************************************************************/
/*
    Displays some basic information on this sensor from the unified
    sensor API sensor_t type (see Adafruit_Sensor for more information)
    */
/**************************************************************************/
void displaySensorDetails(void)
{
    sensor_t sensor;
    bno.getSensor(&sensor);
    Serial.println("------------------------------------");
```

```
  Serial.print("Sensor:       "); Serial.println(sensor.name);
  Serial.print("Driver Ver:   "); Serial.println(sensor.version);
  Serial.print("Unique ID:    "); Serial.println(sensor.sensor_id);
  Serial.print("Max Value:    "); Serial.print(sensor.max_value); Serial.println(" xxx");
  Serial.print("Min Value:    "); Serial.print(sensor.min_value); Serial.println(" xxx");
  Serial.print("Resolution:   "); Serial.print(sensor.resolution); Serial.println(" xxx");
  Serial.println("------------------------------------");
  Serial.println("");
  delay(500);
}

/**************************************************************************/
/*
  Display some basic info about the sensor status
  */
/**************************************************************************/
void displaySensorStatus(void)
{
  /* Get the system status values (mostly for debugging purposes) */
  uint8_t system_status, self_test_results, system_error;
  system_status = self_test_results = system_error = 0;
  bno.getSystemStatus(&system_status, &self_test_results, &system_error);

  /* Display the results in the Serial Monitor */
  Serial.println("");
  Serial.print("System Status: 0x");
  Serial.println(system_status, HEX);
  Serial.print("Self Test:     0x");
  Serial.println(self_test_results, HEX);
  Serial.print("System Error:  0x");
  Serial.println(system_error, HEX);
  Serial.println("");
  delay(500);
}

/**************************************************************************/
/*
  Display sensor calibration status
  */
/**************************************************************************/
void displayCalStatus(void)
{
  /* Get the four calibration values (0..3) */
  /* Any sensor data reporting 0 should be ignored, */
  /* 3 means 'fully calibrated" */
  uint8_t system, gyro, accel, mag;
  system = gyro = accel = mag = 0;
  bno.getCalibration(&system, &gyro, &accel, &mag);

  /* The data should be ignored until the system calibration is > 0 */
  Serial.print("\t");
  if (!system)
  {
```

```
    Serial.print("! ");
  }

  /* Display the individual values */
  Serial.print("Sys:");
  Serial.print(system, DEC);
  Serial.print(" G:");
  Serial.print(gyro, DEC);
  Serial.print(" A:");
  Serial.print(accel, DEC);
  Serial.print(" M:");
  Serial.print(mag, DEC);
}

/**************************************************************************/
/*
  Display the raw calibration offset and radius data
  */
/**************************************************************************/
void displaySensorOffsets(const adafruit_bno055_offsets_t &calibData)
{
  Serial.print("Accelerometer: ");
  Serial.print(calibData.accel_offset_x); Serial.print(" ");
  Serial.print(calibData.accel_offset_y); Serial.print(" ");
  Serial.print(calibData.accel_offset_z); Serial.print(" ");

  Serial.print("\nGyro: ");
  Serial.print(calibData.gyro_offset_x); Serial.print(" ");
  Serial.print(calibData.gyro_offset_y); Serial.print(" ");
  Serial.print(calibData.gyro_offset_z); Serial.print(" ");

  Serial.print("\nMag: ");
  Serial.print(calibData.mag_offset_x); Serial.print(" ");
  Serial.print(calibData.mag_offset_y); Serial.print(" ");
  Serial.print(calibData.mag_offset_z); Serial.print(" ");

  Serial.print("\nAccel Radius: ");
  Serial.print(calibData.accel_radius);

  Serial.print("\nMag Radius: ");
  Serial.print(calibData.mag_radius);
}


/**************************************************************************/
/*
  Arduino setup function (automatically called at startup)
  */
/**************************************************************************/
void setup(void)
{
  //Clears EEPROM
  for (int i = 0 ; i < EEPROM.length() ; i++) {
```

```
  EEPROM.write(i, 0);
}

Serial.begin(9600);
delay(1000);
Serial.println("Orientation Sensor Test"); Serial.println("");

/* Initialise the sensor */
if (!bno.begin())
{
   /* There was a problem detecting the BNO055 ... check your connections */
   Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
   while (1);
}

int eeAddress = 0;
long bnoID;
bool foundCalib = false;

EEPROM.get(eeAddress, bnoID);

adafruit_bno055_offsets_t calibrationData;
sensor_t sensor;

/*
*  Look for the sensor's unique ID at the beginning oF EEPROM.
*  This isn't foolproof, but it's better than nothing.
*/
bno.getSensor(&sensor);
if (bnoID != sensor.sensor_id)
{
   Serial.println("\nNo Calibration Data for this sensor exists in EEPROM");
   delay(700);
}
else
{
   Serial.println("\nFound Calibration for this sensor in EEPROM.");
   eeAddress += sizeof(long);
   EEPROM.get(eeAddress, calibrationData);

   displaySensorOffsets(calibrationData);

   Serial.println("\n\nRestoring Calibration data to the BNO055...");
   bno.setSensorOffsets(calibrationData);

   Serial.println("\n\nCalibration data loaded into BNO055");
   foundCalib = true;
}

delay(1000);

/* Display some basic information on this sensor */
displaySensorDetails();
```

```
/* Optional: Display current status */
displaySensorStatus();

bno.setExtCrystalUse(true);

sensors_event_t event;
bno.getEvent(&event);
if (foundCalib){
  Serial.println("Move sensor slightly to calibrate magnetometers");
  while (!bno.isFullyCalibrated())
  {
    bno.getEvent(&event);
    delay(BNO055_SAMPLERATE_DELAY_MS);
  }
}
else
{
  Serial.println("Please Calibrate Sensor: ");
  while (!bno.isFullyCalibrated())
  {
    bno.getEvent(&event);

    Serial.print("X: ");
    Serial.print(event.orientation.x, 4);
    Serial.print("\tY: ");
    Serial.print(event.orientation.y, 4);
    Serial.print("\tZ: ");
    Serial.print(event.orientation.z, 4);

    /* Optional: Display calibration status */
    displayCalStatus();

    /* New line for the next sample */
    Serial.println("");

    /* Wait the specified delay before requesting new data */
    delay(BNO055_SAMPLERATE_DELAY_MS);
  }
}

Serial.println("\nFully calibrated!");
Serial.println("--------------------------------");
Serial.println("Calibration Results: ");
adafruit_bno055_offsets_t newCalib;
bno.getSensorOffsets(newCalib);
displaySensorOffsets(newCalib);

Serial.println("\n\nStoring calibration data to EEPROM...");

eeAddress = 0;
bno.getSensor(&sensor);
bnoID = sensor.sensor_id;
```

```
  EEPROM.put(eeAddress, bnoID);

  eeAddress += sizeof(long);
  EEPROM.put(eeAddress, newCalib);
  Serial.println("Data stored to EEPROM.");

  Serial.println("\n--------------------------------\n");
  delay(500);
}

void loop() {
  /* Get a new sensor event */
  sensors_event_t event;
  bno.getEvent(&event);

  /* Display the floating point data */
  Serial.print("X: ");
  Serial.print(event.orientation.x, 4);
  Serial.print("\tY: ");
  Serial.print(event.orientation.y, 4);
  Serial.print("\tZ: ");
  Serial.print(event.orientation.z, 4);

  /* Optional: Display calibration status */
  displayCalStatus();

  /* Optional: Display sensor status (debug only) */
  //displaySensorStatus();M

  /* New line for the next sample */
  Serial.println("");

  /* Wait the specified delay before requesting new data */
  delay(BNO055_SAMPLERATE_DELAY_MS);
}
```

Vehicle Data Collection

```
#include <SPI.h>
#include <SdFat.h>
#include <Wire.h>
#include "RTClib.h"
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <StateSpaceControl.h>
#include <EEPROM.h>


//Stores data in C structure for faster data colelction
struct datastore {
   uint32_t microsec;
   float q0;
   float q1;
   float q2;
   float q3;
   float xRvel;
   float yRvel;
   float zRvel;
   float xaccel;
   float yaccel;
   float zaccel;
};

SdFat SD; //Use alternative library for SD writing

long droptime = 1400000; //Set time for drop

Adafruit_BNO055 bno = Adafruit_BNO055(55);//set up IMU

// how many microseconds between grabbing data and loggsing it. 1000 ms is once a second
#define LOG_INTERVAL  12000 // mills between entries (reduce to take more/faster data)

// how many microseconds before writing the logged data permanently to disk
// set it to the LOG_INTERVAL to write each time (safest)
// set it to 10*LOG_INTERVAL to write all data every 10 datareads, you could lose up to
// the last 10 reads if power is lost but it uses less power and is much faster!
#define SYNC_INTERVAL 500000 // mills between calls to flush() - to write data to the card
uint32_t syncTime = 0; // time of last sync()

#define ECHO_TO_SERIAL   0 // echo data to serial port
#define WAIT_TO_START    0 // Wait for serial input in setup()
#define DETECT_DROP      0 //Toggle action at end of drop
RTC_PCF8523 RTC; // define the Real Time Clock object

// for the data logging shield, we use digital pin 10 for the SD cs line
const int chipSelect = 53;
```

```
int ledPin = 9; //pin callout for LED
int ledState = LOW; //initial state for LED

bool dropaction = true; //Flag for drop action
bool startdrop = false; //Flag for start of drop
int startcount = 0;     //Flag for monitored acceleration
bool startransmit = false; //Flag for positions being sent to servos

// Initialize time variables for doing many things at one time
unsigned long previousLedMicros = 0;
unsigned long previousCalMicros = 0;
unsigned long previousReadMicros = 0;
unsigned long previousStopMicros = 0;


// the logging file
File logfile;

void error(char *str)
{
 Serial.print("error: ");
 Serial.println(str);
 while(1);
}

void setup(void)
{ //Stops drop action if flag is false
 #if DETECT_DROP
 dropaction = false;
 #endif

 //Initializes LED
 pinMode(ledPin, OUTPUT);

 //Serial error checking
 Serial.begin(9600);
 Serial.println();

  /* Initialise the sensor */
 if(!bno.begin())
 {
  /* There was a problem detecting the BNO055 ... check your connections */
  Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
  while(1);
 }

 //Reads calibration data off of EEPROM
 int eeAddress = 0;
 long bnoID;
 EEPROM.get(eeAddress, bnoID);
 adafruit_bno055_offsets_t calibrationData;
```

```
  //Sets calibration from read data
  eeAddress += sizeof(long);
  EEPROM.get(eeAddress, calibrationData);
  bno.setSensorOffsets(calibrationData);

  //hold
  delay(1000);

  bno.setExtCrystalUse(true);

//Looks at WAIT_TO_START flag for testing
#if WAIT_TO_START
  Serial.println("Type any character to start");
  while (!Serial.available());
#endif //WAIT_TO_START

  // initialize the SD card
  Serial.print("Initializing SD card...");

  // make sure that the default chip select pin is set to
  // output, even if you don't use it:
  pinMode(53, OUTPUT); //53 for MEGA
  digitalWrite(chipSelect, HIGH);

 // see if the card is present and can be initialized:
  if (!SD.begin(chipSelect)) {
   error("Card failed, or not present");
  }
  Serial.println("card initialized.");

  // create a new file
  char filename[] = "LOGGER00.dat";
  for (uint8_t i = 0; i < 100; i++) {
    filename[6] = i/10 + '0';
    filename[7] = i%10 + '0';
    if (! SD.exists(filename)) {
      // only open a new file if it doesn't exist
      logfile = SD.open(filename, FILE_WRITE);
      break;  // leave the loop!
    }
  }

//  if (! logfile) {
//    error("couldnt create file");
//  }


  Serial.print("Logging to: ");
  Serial.println(filename);

  // connect to RTC
  Wire.begin();
  if (!RTC.begin()) {
```

```
#if ECHO_TO_SERIAL
   Serial.println("RTC failed");
#endif  //ECHO_TO_SERIAL
 }

//Sends initial servo position data to slave controller
Wire.beginTransmission(8); // transmit to device #8
Wire.write("<35,44,36,36>");//sends one byte
Wire.endTransmission();    // stop transmitting
}

void loop(void)
{ //Stores number of microseconds after program start
  unsigned long currentMicros = micros();

  //Set syncTime
  syncTime = currentMicros;

  //Turns LED on if fully calibrated
  if (bno.isFullyCalibrated())
  {         ledState = HIGH;
        digitalWrite(ledPin, ledState);}

  //performs actions if dropaction flag is true
  if (dropaction == true) {

   //Checks if box has been dropped
   CheckDrop(currentMicros);
   //Shuts flaps after predetermined number of microseconds
   ShutFlap(currentMicros);

   //Starts logging data once box is recongized to be falling
   if (startdrop == true) {
   LogData(currentMicros);}

  }
    }

//Logs data in C Structure
void LogData(long currentMicros)
   {

//Logs every n number of microseconds determined by LOG_INTERVAL
 if(currentMicros - previousReadMicros >= LOG_INTERVAL)
  { previousReadMicros = currentMicros;
    //Obtain data from IMU
    imu::Vector<3> accel = bno.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
    imu::Vector<3> angvel = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
    imu::Quaternion quat = bno.getQuat();

    //Initialize structure
    struct datastore myData;
```

```
  //Store data in structure
  myData.microsec = currentMicros;
  myData.q0 = quat.w();
  myData.q1 = quat.x();
  myData.q2 = quat.y();
  myData.q3 = quat.z();
  myData.xRvel = angvel.x();
  myData.yRvel = angvel.y();
  myData.zRvel = angvel.z();
  myData.xaccel = accel.x();
  myData.yaccel = accel.y();
  myData.zaccel = accel.z();

  //Pass structure to logfile
  logfile.write((uint8_t *)&myData, sizeof(myData));

 //Serial echo for testing
 #if ECHO_TO_SERIAL
  Serial.println();
 #endif // ECHO_TO_SERIAL
  }

 // Writes data to disk
 if ((currentMicros - syncTime) < SYNC_INTERVAL){
 syncTime = currentMicros;
 // blink LED to show we are syncing data to the card & updating FAT!

 logfile.flush();}

  }

// Checks if magnitude of acceleration vector passes threshold, starts data capture
void CheckDrop(long currentMicros)
//Gets acceleration vector, starts flap action if box experiences "weightlessness" and certain number of
seconds has passed
{ imu::Vector<3> accel = bno.getVector(Adafruit_BNO055::VECTOR_ACCELEROMETER);
    if ((pow(accel.x(),2)+pow(accel.y(),2)+pow(accel.z(),2) < 2) and (currentMicros > 5000000))
      {startcount =1;}

  //Starts flap action if not already started
  if((startcount == 1) && (startransmit == false))
  {
     startdrop = true; //Signals start of drop for data logging
     ledState = HIGH;
     digitalWrite(ledPin, ledState); //Turns LED on
     Wire.beginTransmission(8); // transmit to device #8 (Teensy)
     Wire.write("<35,44,36,36>");//sends servo positions to Teensy. Change for different angles. Format
<Flap1, Flap 2, etc> – change to <115,120,124,110> for full extension
     Wire.endTransmission();    // stop transmitting
     startransmit = true; //Flags start of Flap action
     previousStopMicros = currentMicros; //Logs current time

  }
```

```
}

//Shuts all flaps after predetermined length of time defined by droptime variable
void ShutFlap(long currentMicros)
//Checks if box flaps have actuated
{if (startransmit == true)
 {

 //checks if number of microseconds has passed since flap actuation
 if((currentMicros - previousStopMicros) >= droptime)
   {  //Sends flap positions to Teensy to close flaps
      Wire.beginTransmission(8); // transmit to device #8
      Wire.write("<35,44,36,36>");//sends one byte
      Wire.endTransmission();    // stop transmitting

   }


 }

}
```

Teensy Servo Controller

```
#include <Wire.h>
#include <Servo.h>

//Toggle Serial
#define SEE_SERIAL 0

const byte numChars = 32;
char receivedChars[numChars];   // an array to store the received data
char tempChars[numChars];

// variables to hold the parsed servo positions
int flapOne = 0;
int flapTwo = 0;
int flapThree = 0;
int flapFour = 0;

boolean newData = false;

// Initialize servo pins
const int servoPin1 = 19;
const int servoPin2 = 18;
const int servoPin3 = 16;
const int servoPin4 = 17;

//Servo initialization
Servo servo1;
Servo servo2;
Servo servo3;
Servo servo4;

void setup() {
   //start receiving I2C
   Wire.begin(8);
   Wire.onReceive(recvWithStartEndMarkers);

   #if SEE_SERIAL
      Serial.begin(9600);
      Serial.println("Start words");
   #endif

   //Attach Servos
   servo1.attach(servoPin1);
   servo2.attach(servoPin2);
   servo3.attach(servoPin3);
   servo4.attach(servoPin4);
}

void loop() {
   if (newData == true) {
   strcpy(tempChars, receivedChars);
```

```
      // this temporary copy is necessary to protect the original data
      //   because strtok() used in parseData() replaces the commas with \0
   //Parse data and move servos
   parseData();
   moveServo();
   #if SEE_SERIAL
      showParsedData();
   #endif

   newData = false;
   }
}


//Parses data, stores servo positions
void parseData() {      // split the data into its parts

   char * strtokIndx; // this is used by strtok() as an index

   strtokIndx = strtok(tempChars,",");      // get the first part - the string
   flapOne = atoi(strtokIndx);     // convert this part to an integer

   strtokIndx = strtok(NULL, ","); // this continues where the previous call left off
   flapTwo = atoi(strtokIndx);     // convert this part to an integer

   strtokIndx = strtok(NULL, ",");
   flapThree = atoi(strtokIndx);     // convert this part to an integer

   strtokIndx = strtok(NULL, ",");
   flapFour = atoi(strtokIndx);     // convert this part to an integer

}

//Move servo to position
void moveServo() {
   servo1.write(flapOne);
   servo2.write(flapTwo);
   servo3.write(flapThree);
   servo4.write(flapFour);
}

//Show data being received
void showParsedData() {
   Serial.print("ONE = ");
   Serial.println(flapOne);
   Serial.print("TWO = ");
   Serial.println(flapTwo);
   Serial.print("THREE = ");
   Serial.println(flapThree);
   Serial.print("FOUR = ");
   Serial.println(flapFour);
```

APPENDIX D – DATA PROCESSING CODE

Import Fall Data

```
from ctypes import *
import numpy as np
import pickle
from scipy.spatial.transform import Slerp
from scipy.spatial.transform import Rotation as R
from scipy.signal import savgol_filter
from scipy.interpolate import make_interp_spline, BSpline
from scipy import interpolate

#Creates c strucutre for importing of binary data
class Falldata(Structure):
   _fields_ = [('microsec', c_uint32),
            ('q0', c_float),
            ('q1', c_float),
            ('q2', c_float),
            ('q3', c_float),
            ('xangvel', c_float),
            ('yangvel', c_float),
            ('zangvel', c_float),
            ('xaccel', c_float),
            ('yaccel', c_float),
            ('zaccel', c_float)]

#Data object to store and modify imported data
class Data(object):
   def __init__(self,filepath):
      #initialize drop interval variables
      self.start = 0
      self.stop = 0

      #Opens data file, stores data
      with open(filepath, 'rb') as file:
         #Initilizes arrays for time, quaternions, acclerometer, and ghyroscope data
         self.time = []
         self.q0 = []
         self.q1 = []
         self.q2 = []
         self.q3 = []
         self.xangvel = []
         self.yangvel = []
         self.zangvel = []
         self.xaccel = []
         self.yaccel = []
         self.zaccel = []
         self.xang_accel = []
         self.yang_accel = []
         self.zang_accel = []
         self.rmatrices = []
         self.rawm = []
```

```
        self.w = []
        self.omega = []
        self.euler = []
        x = Falldata()

        #Imports data line by line into arrays
        while file.readinto(x) == sizeof(x):
           self.time.append(x.microsec)
           self.q0.append(x.q0)
           self.q1.append(x.q1)
           self.q2.append(x.q2)
           self.q3.append(x.q3)
           self.xangvel.append(x.xangvel*np.pi/180)
           self.yangvel.append(x.yangvel*np.pi/180)
           self.zangvel.append(x.zangvel*np.pi/180)
           self.xaccel.append(x.xaccel)
           self.yaccel.append(x.yaccel)
           self.zaccel.append(x.zaccel)

        #Stores magnitude of acceleration
        self.accel = np.array([np.sqrt(a**2 +b**2 +c**2) for a,b,c in zip(self.xaccel, self.yaccel, self.zaccel)])
        self.omega = np.array([np.sqrt(a**2 +b**2 +c**2) for a,b,c in zip(self.xangvel, self.yangvel,
self.zangvel)])

   #Truncates data such that only the drop data is analyzed
   def droponly(self):
        #Selects start time as 2 samples after when accleration is less than 4 m/s^2
        self.start = (self.accel < 4).argmax()+2
        #Selects stop time as the sample before first time accleration is felt above 10 m/s^2
        self.stop = [i for i,v in enumerate(self.accel) if abs(v)>10 and i > self.start][0]-1

        #Redefines data such that only the values between start and stop are observed
        self.time = self.time[self.start:self.stop]
        self.q0 = self.q0[self.start:self.stop]
        self.q1 = self.q1[self.start:self.stop]
        self.q2 = self.q2[self.start:self.stop]
        self.q3 = self.q3[self.start:self.stop]
        self.xangvel = self.xangvel[self.start:self.stop]
        self.yangvel = self.yangvel[self.start:self.stop]
        self.zangvel = self.zangvel[self.start:self.stop]
        self.xaccel = self.xaccel[self.start:self.stop]
        self.yaccel = self.yaccel[self.start:self.stop]
        self.zaccel = self.zaccel[self.start:self.stop]
        self.accel = self.accel[self.start:self.stop]
        self.omega = self.omega[self.start:self.stop]


   # Normalizes angular velocity & orientation into world frame,
   # refpath should be Baseline, not Inertia
   def normalizeworld(self,basepath):

      tangvel = []
      #import saved reference orientation from baseline file
```

```
R0,R0inv = pickle.load(open(basepath, "rb"))

#import rotation matrices between frames
Rcal, Rcalinv = pickle.load(open("K:\CppThesis\ProtoData\InertialCal.pickle", "rb"))

#Prepare angular velocity array
angvel = np.array([self.xangvel, self.yangvel, self.zangvel])
angvel = np.transpose(angvel)

#Get length of array
rows = angvel.shape[0]

#Creates rotation matrices for each sample, converts those matricces
# and the angular velocities into world frame and
for i in range(0, rows):
   #Get single velocity vector
   vel_vector = angvel[i]

   #Get quaternion
   q0 = self.q0[i]
   q1 = self.q1[i]
   q2 = self.q2[i]
   q3 = self.q3[i]



   #Convert quaternion into rotation mztrix
   R11 = 1 - 2 * (q2 ** 2 + q3 ** 2)
   R12 = 2 * (q1 * q2 - q0 * q3)
   R13 = 2 * (q0 * q2 + q1 * q3)
   R21 = 2 * (q1 * q2 + q0 * q3)
   R22 = q0 ** 2 - q1 ** 2 + q2 ** 2 - q3 ** 2
   R23 = 2 * (q2 * q3 - q0 * q1)
   R31 = 2 * (q1 * q3 - q0 * q2)
   R32 = 2 * (q0 * q1 + q2 * q3)
   R33 = q0 ** 2 - q1 ** 2 - q2 ** 2 + q3 ** 2

   # Prepare rows
   row1 = np.array([R11, R12, R13])
   row2 = np.array([R21, R22, R23])
   row3 = np.array([R31, R32, R33])

   # Corrects for orthonormality of rotation matrix. Procedure from
   # William Premerlani & Paul Bizard's Direction Cosine Matrix IMU: Theory
   # https://drive.google.com/file/d/0B9rLLz1XQKmaZTlQdV81QjNoZTA/view

   error = np.dot(row1, row2)

   row1_ort = row1 - (error / 2) * row2
   row2_ort = row2 - (error / 2) * row1
   row3_ort = np.cross(row1_ort, row2_ort)

   row1 = .5 * (3 - np.dot(row1_ort, row1_ort)) * row1_ort
```

```
        row2 = .5 * (3 - np.dot(row2_ort, row2_ort)) * row2_ort
        row3 = .5 * (3 - np.dot(row3_ort, row3_ort)) * row3_ort

        # Final rotation matrix in default IMU reference frame
        R_imu_alpha = np.array([row1, row2, row3]).reshape(3, 3)

        # self.append.rawm(R_imu_alpha)

        #Convert into imu orientation in imu world imu frame
        R_imu_imu = np.dot(R0inv, R_imu_alpha)



        #Convert rotation matrix into orientaion in box world frame
        Rx = np.dot(Rcalinv,np.dot(R_imu_imu,Rcal ))

        #Log orientation in world frame
        self.rmatrices.append(Rx)


        #Convert velocity vector into world frame, log velocity vector
        tangvel.append(np.dot(Rcalinv,np.dot(R_imu_imu, vel_vector)))

    #Reformat velocity array
    tangvel = np.array(tangvel)
    tangvel = np.transpose(tangvel)

    #Redefine angular velocities into world frame
    self.xangvel = tangvel[0]
    self.yangvel = tangvel[1]
    self.zangvel = tangvel[2]


# Defines rotation matrix of world/inertial frame from baseline file, inputs are time
# bounds of data
def worldbase(self,left_bound,right_bound):

    #np for manipulation
    time = np.array(self.time)

    #defines indexes of data between time bounds
    base_index = np.where((time >= left_bound) & (time <= right_bound))

    #import quaternions for manipulation, trim to between bounds
    q0 = np.array(self.q0)
    q1 = np.array(self.q1)
    q2 = np.array(self.q2)
    q3 = np.array(self.q3)

    q0 = np.mean(q0[base_index])
    q1 = np.mean(q1[base_index])
    q2 = np.mean(q2[base_index])
    q3 = np.mean(q3[base_index])
```

```
    #Rotation matrices from quaternions
    R11 = 1 - 2 * (q2 ** 2 + q3 ** 2)
    R12 = 2 * (q1 * q2 - q0 * q3)
    R13 = 2 * (q0 * q2 + q1 * q3)
    R21 = 2 * (q1 * q2 + q0 * q3)
    R22 = q0 ** 2 - q1 ** 2 + q2 ** 2 - q3 ** 2
    R23 = 2 * (q2 * q3 - q0 * q1)
    R31 = 2 * (q1 * q3 - q0 * q2)
    R32 = 2 * (q0 * q1 + q2 * q3)
    R33 = q0 ** 2 - q1 ** 2 - q2 ** 2 + q3 ** 2

    #Prepare rows
    row1 = np.array([R11, R12, R13])
    row2 = np.array([R21, R22, R23])
    row3 = np.array([R31, R32, R33])

    # Corrects for orthonormality of rotation matrix. Procedure from
    # William Premerlani & Paul Bizard's Direction Cosine Matrix IMU: Theory
    # https://drive.google.com/file/d/0B9rLLz1XQKmaZTlQdV81QjNoZTA/view

    error = np.dot(row1, row2)

    row1_ort = row1 - (error / 2) * row2
    row2_ort = row2 - (error / 2) * row1
    row3_ort = np.cross(row1_ort, row2_ort)

    row1 = .5 * (3 - np.dot(row1_ort, row1_ort)) * row1_ort
    row2 = .5 * (3 - np.dot(row2_ort, row2_ort)) * row2_ort
    row3 = .5 * (3 - np.dot(row3_ort, row3_ort)) * row3_ort

    # Final rotation matrix and inverse
    R0 = np.array([row1, row2, row3]).reshape(3, 3)

    R0inv = np.linalg.inv(R0)


    return R0,R0inv

#Data interpolation and spline fitting
def interp(self,numpts,filterwindow,r):
    #Rotates data 180 degrees about y-axis - (For flap1 drop) if r ==1
    if r == 1:
        n = R.from_dcm([[-1,0,0],[0,1,0],[0,0,-1]])
        for i,e in enumerate(self.rmatrices):
            self.rmatrices[i] = n.apply(self.rmatrices[i])
    #get rotations
    rotations = R.from_dcm(self.rmatrices)

    #Sperical Linear Interpolation
    slerp = Slerp(self.time,rotations)

    #Times from new data points
```

```
stime = np.linspace(np.array(self.time).min(), np.array(self.time).max(), numpts)

#Spline fit
spl = make_interp_spline(self.time, self.xangvel, k=3)  # BSpline object
#New data points
s_xangvel = spl(stime)
#Smoothing
self.xangvel = savgol_filter(s_xangvel, filterwindow, 3)

#Spline fit
spl = make_interp_spline(self.time, self.yangvel, k=3)  # BSpline object
#New data points
s_yangvel = spl(stime)
#Smoothing
self.yangvel = savgol_filter(s_yangvel, filterwindow, 3)

#Spline fit
spl = make_interp_spline(self.time, self.zangvel, k=3)  # BSpline object
#New data points
s_zangvel = spl(stime)
#Smoothing
self.zangvel = savgol_filter(s_zangvel, filterwindow, 3)

# Spline fit
spl = make_interp_spline(self.time, self.accel, k=3)  # BSpline object
# New data points
s_accel = spl(stime)
#Smoothing
self.accel = savgol_filter(s_accel, filterwindow, 3)

#New times and new rotation matrices
self.time = stime
self.rmatrices = slerp(stime).as_dcm()

#Also stores rotations as Euler angles
self.euler = slerp(stime).as_euler('xyz')
```

World Reference File Creator

```
import matplotlib.pyplot as plt
from ImportFallData import *
import pickle
from pathlib import Path

# Looks at baseline file for inertial reference orientation
data_folder = Path("D:\Imps\CppThesis\ProtoData\Feb21_19/")

#Filepath
filepath = data_folder / "Baseline.dat"

#Calld data object for use
base = Data(filepath)

#Variable to choose bounds from graph or do manually
view = False

#Plots quaternions over time, asks for inputs of desired range, closes
if view:
   plt.plot(base.time,base.q0,'r')
   plt.plot(base.time,base.q1,'g')
   plt.plot(base.time,base.q2,'b')
   plt.plot(base.time,base.q3,'k')

   plt.title(filepath)
   plt.pause(0.1)

   left_bound = input("Left bound of stable reference orientation")
   right_bound = input("Right bound of stable reference orientation")

   plt.close()
#if plot not needed, use hardcoded values for bounds
elif not view :

   #Hardcoded bounds for desired dataset
   left_bound = 3.25e7
   right_bound = 3.75e7

#Call data class function to get inertial rotation matrix with inverse
Ri,Ri_inv = base.worldbase(left_bound,right_bound)

#Save variables in file for later use
refpath = data_folder / (filepath.stem + ".pickle")

with open(refpath,"wb") as f:
   pickle.dump([Ri,Ri_inv],f)

#print rotation matrix for review
print(Ri)
```

IMU to Body Axes File Creator

```
# Finds the rotation matrix that transforms the IMU frame to the desired reference frame of the cat box.
import matplotlib.pyplot as plt
from ImportFallData import *
from pathlib import Path
import pickle

#Folder paths
#data_folder = Path("D:\Imps\CppThesis\ProtoData\Feb21_19/")
data_folder = Path("D:\Imps\CppThesis\ProtoData")

# Respective files for vector calibration. Each file contains data in which the box was rotates about the
desired inertial axis
pitchpath = data_folder / "pitch_cal.dat"
rollpath = data_folder / "roll_cal.dat"
yawpath = data_folder / "yaw_cal.dat"

#Load data
pitch = Data(pitchpath)
roll = Data(rollpath)
yaw = Data(yawpath)

#Set bounds of data where data looks promising from visualized graphs
pitch_lbound = 3.9e7
pitch_rbound = 5.28e7
roll_lbound = 6.02e7
roll_rbound = 7.09e7
yaw_lbound = 3.07e7
yaw_rbound = 4.23e7

#Import data into numpy arrays for easier manipulation
pitchtime = np.array(pitch.time)
rolltime = np.array(roll.time)
yawtime = np.array(yaw.time)

pitch_x = np.array(pitch.xangvel)
pitch_y = np.array(pitch.yangvel)
pitch_z = np.array(pitch.zangvel)

roll_x = np.array(roll.xangvel)
roll_y = np.array(roll.yangvel)
roll_z = np.array(roll.zangvel)

yaw_x = np.array(yaw.xangvel)
yaw_y = np.array(yaw.yangvel)
yaw_z = np.array(yaw.zangvel)

#Obtain indices of data between desired bounds
pitch_index = np.where((pitchtime >= pitch_lbound) & (pitchtime <= pitch_rbound))
roll_index = np.where((rolltime >= roll_lbound) & (rolltime <= roll_rbound))
```

```
yaw_index = np.where((yawtime >= yaw_lbound) & (yawtime <= yaw_rbound))


#isolate data between desired bounds
p = [pitch_x[pitch_index], pitch_y[pitch_index], pitch_z[pitch_index]]
r = [roll_x[roll_index], roll_y[roll_index], roll_z[roll_index]]
y = [yaw_x[yaw_index], yaw_y[yaw_index], yaw_z[yaw_index]]

#Finds data where angular velocity is below threshold, deletes that data from consideration
pitch0_index = np.where(p[2] < 100)
p0 = np.delete(p[0],pitch0_index)
p1 = np.delete(p[1],pitch0_index)
p2 = np.delete(p[2],pitch0_index)

#Form new array
p = [p0,p1,p2]
#transpose for normalization
p = np.transpose(p)

#normalize data
p /= np.sqrt((p ** 2).sum(-1))[..., np.newaxis]

#transpose again for grouping
p = np.transpose(p)


#Repeat for other 2 rotations
roll0_index = np.where(r[0]< 100)
r0 = np.delete(r[0],roll0_index)
r1 = np.delete(r[1],roll0_index)
r2 = np.delete(r[2],roll0_index)

r = [r0,r1,r2]
r = np.transpose(r)
r /= np.sqrt((r ** 2).sum(-1))[..., np.newaxis]
r = np.transpose(r)

yaw0_index = np.where(y[1] < 100)
y0 = np.delete(y[0],yaw0_index)
y1 = np.delete(y[1],yaw0_index)
y2 = np.delete(y[2],yaw0_index)

y = [y0,y1,y2]
y = np.transpose(y)
y /= np.sqrt((y ** 2).sum(-1))[..., np.newaxis]
y = np.transpose(y)

#Used to select bounds for inertial frame calibration data
# plt.plot(y[0],'r')
# plt.plot(y[1],'g')
# plt.plot(y[2],'b')

# plt.show()
```

```
#Take averages of normalized data for insertion into rotation matrix
R11 = np.mean(r[0])
R12 = np.mean(r[1])
R13 = np.mean(r[2])
R21 = np.mean(y[0])
R22 = np.mean(y[1])
R23 = np.mean(y[2])
R31 = np.mean(p[0])
R32 = np.mean(p[1])
R33 = np.mean(p[2])

row1 = np.array([R11,R12,R13])
row2 = np.array([R21,R22,R23])
row3 = np.array([R31,R32,R33])

# Corrects for orthonormality of rotation matrix. Procedure from
# William Premerlani & Paul Bizard's Direction Cosine Matrix IMU: Theory
# https://drive.google.com/file/d/0B9rLLz1XQKmaZTlQdV81QjNoZTA/view

error = np.dot(row1,row2)

row1_ort = row1-(error/2)*row2
row2_ort = row2-(error/2)*row1
row3_ort = np.cross(row1_ort,row2_ort)

row1 = .5*(3-np.dot(row1_ort,row1_ort))*row1_ort
row2 = .5*(3-np.dot(row2_ort,row2_ort))*row2_ort
row3 = .5*(3-np.dot(row3_ort,row3_ort))*row3_ort


#Final rotation matrix and inverse, where Rcal is the box frame referenced in the IMU frame
Rcal= np.array([row1, row2, row3]).reshape(3, 3)
Rcalinv = np.linalg.inv(Rcal)

#Sets path for save file
refpath = data_folder / "InertialCal.pickle"

#Uses pickle library to save matrices as variables for later use
with open(refpath,"wb") as f:
   pickle.dump([Rcal,Rcalinv],f)
```

Vehicle Geometry

```
import numpy as np
from AirChecker import AirChecker
from shapely.geometry import Polygon

#Calculates the projected areas of the box and finds their centroids in the world frame

#double dot product
def dbldot(third, first, second):
   product = np.dot(third, np.dot(first, second))
   return product

#Augmentation of 3x3 rotation matrix to 4x4 transformation matrix
def make4(threebythree):
   return np.vstack((np.hstack((threebythree, np.transpose([[0, 0, 0]]))), [[0, 0, 0, 1]]))

#4x4 transformation matrix to rotation amtrix
def make3(fourbyfour):
   return fourbyfour[0:3,0:3]

#Rotate position vectors into world frame
def faceinWorld(points, RinWorld):
   spoints = points
   spoints = np.transpose(np.array(spoints))
   RW = RinWorld
   pdata = np.dot(RW, spoints)

   return pdata



#Defines corner points of box from ideal center
FRONT_lt = [4, 4, 4]
FRONT_rt = [4, 4, -4]
FRONT_rb = [4, -4, -4]
FRONT_lb = [4, -4, 4]
BACK_rb = [-4, -4, -4]
BACK_lb = [-4, -4, 4]
BACK_lt = [-4, 4, 4]
BACK_rt = [-4, 4, -4]

#Created class instance for the box at a certain orientation with certain flap angle
class AreaCalcs(object):
   #Input orientation, flap angles
   def __init__(self,RinWorld,a1,a2,a3,a4):

      RinWorld = np.array(RinWorld)

      #Determine which faces of the box face the ground
      truthtable = AirChecker(RinWorld)
```

```
#Define faces in terms of corner points
f1 = np.array([FRONT_lt, FRONT_rt , FRONT_rb, FRONT_lb])
f2 = np.array([FRONT_rt, FRONT_rb, BACK_rb, BACK_rt])
f3 = np.array([BACK_lt, BACK_rt , BACK_rb, BACK_lb])
f4 = np.array([FRONT_lt, FRONT_lb, BACK_lb, BACK_lt])
f5 = np.array([FRONT_lb, FRONT_rb, BACK_rb, BACK_lb])
f6 = np.array([FRONT_lt, FRONT_rt, BACK_rt, BACK_lt])

#Face array
faces = [f1,f2,f3,f4,f5,f6]

#Define location of flap points relative to a frame located at the pivot point
flap_pt1 = [0.5,-3.125,4,1]
flap_pt2 = [0.5, -3.125, -4, 1]
flap_pt3 = [0.5, -0.625, -4, 1]
flap_pt4 = [0.5, -0.625, 4, 1]

flap_pts = np.transpose(np.array([flap_pt1,flap_pt2,flap_pt3,flap_pt4]))

#Rotation matrices for flaps dependent on flap angle
Ta1_angle = [[np.cos(a1),-np.sin(a1),0,0],
        [np.sin(a1),np.cos(a1),0,0],
        [0,0,1,0],
        [0,0,0,1]]

Ta2_angle = [[np.cos(a2),-np.sin(a2),0,0],
        [np.sin(a2),np.cos(a2),0,0],
        [0,0,1,0],
        [0,0,0,1]]

Ta3_angle = [[np.cos(a3),-np.sin(a3),0,0],
        [np.sin(a3),np.cos(a3),0,0],
        [0,0,1,0],
        [0,0,0,1]]

Ta4_angle = [[np.cos(a4),-np.sin(a4),0,0],
        [np.sin(a4),np.cos(a4),0,0],
        [0,0,1,0],
        [0,0,0,1]]

#Transformation matrices from ideal center to flap pivot points
T_a1 = [[1,0,0,3.5],
     [0,1,0,3.5],
     [0,0,1,0],
     [0,0,0,1]]

T_a2 = [[0,0,1,0],
     [0,1,0,3.5],
     [-1,0,0,-3.5],
     [0,0,0,1]]

T_a3 = [[-1,0,0,-3.5],
     [0,1,0,3.5],
```

```
      [0,0,-1,0],
      [0,0,0,1]]

T_a4 = [[0,0,-1,0],
      [0,1,0,3.5],
      [1,0,0,3.5],
      [0,0,0,1]]

#makes transformation matrix from orientation in world
RinWorld4 = make4(RinWorld)

#Transformations from flap coordinates to positions in world frame
T_flap10 = dbldot(RinWorld4,T_a1,Ta1_angle)
T_flap20 = dbldot(RinWorld4,T_a2, Ta2_angle)
T_flap30 = dbldot(RinWorld4,T_a3, Ta3_angle)
T_flap40 = dbldot(RinWorld4,T_a4, Ta4_angle)

#Apply transformations to flap coordinates
flap_bxpts1 = np.dot(T_flap10 , flap_pts)
flap_bxpts2 = np.dot(T_flap20 , flap_pts)
flap_bxpts3 = np.dot(T_flap30 , flap_pts)
flap_bxpts4 = np.dot(T_flap40 , flap_pts)

#Gather
flaps = [flap_bxpts1,flap_bxpts2,flap_bxpts3,flap_bxpts4]

#Format
for i,pts in enumerate(flaps):
   flaps[i] = make3(pts)

#Store
self.flaps = flaps

#Rotate face points into world frame
for i,pts in enumerate(faces):
   faces[i] = faceinWorld(pts,RinWorld)

#Store
self.faces = faces

#Isolate faces projected downwards
downfaces = np.array(faces)[truthtable]

#Obtain normals of faces
normals = [RinWorld[:,0],-RinWorld[:,2],-RinWorld[:,0],RinWorld[:,2],-RinWorld[:,1],RinWorld[:,1]]
downN = np.array(normals)[truthtable]

#Initialize face properties
centroids = np.zeros([downfaces.shape[0],3,1])
projareas = np.zeros([downfaces.shape[0],1,1])
liftvecs = np.zeros([downfaces.shape[0],3,1])

#Finds projected areas of downward faces and their respective centroids
```

```
for i, pts in enumerate(downfaces):
    #get coordinates of projected face in x-z plane
    twoD = np.transpose(np.array([pts[0], pts[2]]))

    #Create polygon object from points
    shadow = Polygon(twoD)

    #constant for equation of the face plane ax+by+cz + d = 0
    d = -np.dot(downN[i],np.transpose(pts)[0])

    #finds y value of centroid projected back onto face
    proj = -(downN[i][0]*shadow.centroid.x + downN[i][2]*shadow.centroid.y+d)/downN[i][1]

    #Stores centroid vector
    centroids[i][0] = shadow.centroid.x
    centroids[i][1] = proj
    centroids[i][2] = shadow.centroid.y

    #Stores projected area
    projareas[i] = shadow.area

    #Unit vectors of lift for each face
    liftvecs[i][0] = shadow.centroid.x
    liftvecs[i][1] = 0
    liftvecs[i][2] = shadow.centroid.y

    liftnorm = np.linalg.norm(liftvecs[i])
    if liftnorm == 0:
        liftvecs[i] = [[0],[0],[0]]
    else:
        liftvecs[i] = -liftvecs[i]/liftnorm


#Centroids exported in units of m
self.centroids = .0254*centroids
self.area = projareas*0.00064516
self.liftvecs = liftvecs
```

Air Checker

```
import numpy as np


#function that uses directional cosines to check what faces face the ground

def AirChecker(RinWorld):
    #directional cosines of orientation with respect to downward gravity unit vector
    dircos = np.dot([0, -1, 0], RinWorld)
    #tolerance of directional cosines
    tol = 0.001
    #initialize truthtable
    truthtable = []

    #Creates truthtable through appending
    for i in dircos:

        #Determines exposed faces using the directional cosines
        if abs(i) > tol:

            if i < tol:
                truthtable.append(False)
                truthtable.append(True)

            if i > tol:
                truthtable.append(True)
                truthtable.append(False)

        else:
            truthtable.append(False)
            truthtable.append(False)
    #Reformats into [Face1, Face2, etc]
    index = [0,5,1,4,3,2]
    truthtable = np.array(truthtable)[index]

    return truthtable
```

Inertia Tensor Calculation (MATLAB)

```
%mass moments of inertia in vehicle frame
Nxx = 0.003685;
Nyy = 0.003455;
Nzz = 0.003728;


%rotated Z moments
Ixx1 = 0.003691;
Iyy1 = 0.003681;
Izz1 = Nzz;

%rotated Y moments
Ixx2 = 0.003808;
Iyy2 = Nyy;
Izz2 = 0.003833;

%rotated X moments
Ixx3 = Nxx;
Iyy3 = 0.003735;
Izz3 = 0.003767;


theta = 45

r = @(Ixx,Iyy,Izz) (Ixx-(cosd(theta)^2).*Nxx-
(sind(theta)^2).*Nyy)./(2.*cosd(theta).*sind(theta));
s = @(Ixx,Iyy,Izz) (Izz-(cosd(theta)^2).*Nxx-
(cosd(theta)^2).*Nyy)./(2.*cosd(theta).*cosd(theta));
t = @(Ixx,Iyy,Izz) (Iyy-(cosd(theta)^2).*Nyy-
(sind(theta)^2).*Nzz)./(2.*cosd(theta).*sind(theta));


Nxy = r(Ixx1,Iyy1,Izz1)
Nxz = s(Ixx2,Iyy2,Izz2)
Nyz = t(Ixx3,Iyy3,Izz3)
```

APPENDIX E– VISUALIZATION CODE

Orientation Visualizer

```
from AreaCalcs import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#Checks Orientation of box graphically
def CheckOrientationV2(RinWorld,a1,a2,a3,a4):
   RinWorld = np.array(RinWorld)

   truthtable = AirChecker(RinWorld)

   #Rotation for plot
   Rg = [[0, 0, 1], [1, 0, 0], [0, 1, 0]]

   #Rotates points into plot frame
   def plotready(points):
      spoints = np.array(points)

      pdata = np.dot(Rg,spoints)

      return pdata

   checker = AreaCalcs(RinWorld,a1,a2,a3,a4)
   flaps = [0,0,0,0]
   faces = [0,0,0,0,0,0]

   #Rotate flap points into plot frame
   for i,pts in enumerate(checker.flaps):
      flaps[i] = plotready(np.array(pts))

   #pts for plotting grouped by flap
   fp1 = flaps[0]
   fp2 = flaps[1]
   fp3 = flaps[2]
   fp4 = flaps[3]

   # Rotate face points into plot frame
   for i,pts in enumerate(checker.faces):
      faces[i] = plotready(np.array(pts))

   #Defines lines that make up shape of box
   L1 = np.array([FRONT_lt, FRONT_rt , FRONT_rb,
FRONT_lb,BACK_lb,BACK_rb,BACK_rt,BACK_lt,FRONT_lt])
   L2 = np.array([FRONT_lt,FRONT_lb])
   L3 = np.array([FRONT_rt,BACK_rt])
   L4 = np.array([BACK_lt,BACK_lb])
   L5 = np.array([BACK_rb,FRONT_rb])
```

```
#transforms lines into world frame, then into plot frame
L1 = plotready(np.dot(RinWorld,np.transpose(L1)))
L2 = plotready(np.dot(RinWorld,np.transpose(L2)))
L3 = plotready(np.dot(RinWorld,np.transpose(L3)))
L4 = plotready(np.dot(RinWorld,np.transpose(L4)))
L5 = plotready(np.dot(RinWorld,np.transpose(L5)))

#Pts of diagonals of faces

x1 = [faces[0][0][0],faces[0][0][2]]
y1 = [faces[0][1][0],faces[0][1][2]]
z1 = [faces[0][2][0],faces[0][2][2]]

x2 = [faces[1][0][0],faces[1][0][2]]
y2 = [faces[1][1][0],faces[1][1][2]]
z2 = [faces[1][2][0],faces[1][2][2]]

x3 = [faces[2][0][0],faces[2][0][2]]
y3 = [faces[2][1][0],faces[2][1][2]]
z3 = [faces[2][2][0],faces[2][2][2]]

x4 = [faces[3][0][0],faces[3][0][2]]
y4 = [faces[3][1][0],faces[3][1][2]]
z4 = [faces[3][2][0],faces[3][2][2]]

x5 = [faces[4][0][0],faces[4][0][2]]
y5 = [faces[4][1][0],faces[4][1][2]]
z5 = [faces[4][2][0],faces[4][2][2]]

x6 = [faces[5][0][0],faces[5][0][2]]
y6 = [faces[5][1][0],faces[5][1][2]]
z6 = [faces[5][2][0],faces[5][2][2]]

#Corner Pts
ptx = [faces[0][0],faces[2][0]]
pty = [faces[0][1],faces[2][1]]
ptz = [faces[0][2],faces[2][2]]

#Plot Figure
fig = plt.figure(figsize=plt.figaspect(1)*1.5)

#3D plot config
ax = fig.add_subplot(111, projection='3d',proj_type='ortho')
#plot corner pts
ax.scatter(ptx,pty,ptz,color='k',edgecolor='w',s = 60)
#Constrian aspect ratio
ax.set_aspect(1)

#Plot lines of box
ax.plot(L1[0],L1[1],L1[2],color = 'k',alpha = 0.5)
ax.plot(L2[0],L2[1],L2[2],color = 'k',alpha = 0.5)
ax.plot(L3[0],L3[1],L3[2],color = 'k',alpha = 0.5)
ax.plot(L4[0],L4[1],L4[2],color = 'k',alpha = 0.5)
```

```
ax.plot(L5[0],L5[1],L5[2],color = 'k',alpha = 0.5)

#Plot flap pts
ax.scatter(fp1[0],fp1[1],fp1[2],color = 'r')
ax.scatter(fp2[0],fp2[1],fp2[2],color = 'c')
ax.scatter(fp3[0], fp3[1], fp3[2], color='y')
ax.scatter(fp4[0], fp4[1], fp4[2], color='b')


#Plot diagonals of faces if they are facing down
if truthtable[0]:
   ax.plot(x1,y1,z1,color = 'r')
if truthtable[1]:
   ax.plot(x2,y2,z2,color = 'c')
if truthtable[2]:
   ax.plot(x3,y3,z3,color = 'y')
if truthtable[3]:
   ax.plot(x4,y4,z4,color = 'b')
if truthtable[4]:
   ax.plot(x5,y5,z5,color = 'k')
if truthtable[5]:
   ax.plot(x6,y6,z6,color = 'g')

#Axis Labels
ax.set_xlabel('Box Z')
ax.set_ylabel('Box X')
ax.set_zlabel('Box Y')

#Axis limits
ax.set_xlim(-6,6)
ax.set_ylim(-6, 6)
ax.set_zlim(-6, 6)

#Invert axes for proper view
ax.invert_xaxis()
ax.invert_yaxis()
ax.grid(False)


# plt.show()

return fig
```

Animator

```
from AreaCalcs import *
import matplotlib.pyplot as plt
from matplotlib import animation
from ImportFallData import *
from pathlib import Path
from scipy.optimize import curve_fit
from scipy.signal import savgol_filter
from scipy.interpolate import make_interp_spline, BSpline
from CheckOrientationV2 import CheckOrientationV2
import matplotlib.image as mpimg
import io
import urllib, base64
import cv2
import os

import matplotlib

plt.rcParams["font.family"] = "Times New Roman"

del matplotlib.font_manager.weight_dict['roman']
matplotlib.font_manager._rebuild()
#folderpaths
data_folder = Path("K:\CppThesis\ProtoData")
subfolder = "Feb21_19/"


# Looks at Inertial file for reference orientation
refpath = "K:\CppThesis\ProtoData\Feb21_19\Baseline.pickle"

#filename for animation
filename = "Feb21_19 Flap1 at 76 degrees 1.dat"

#concetnating filepath
filepath = data_folder / (subfolder + filename)

#Get filename for animation title
filename, file_extension = os.path.splitext(filename)

#Import data
Data = Data(filepath)
Data.droponly()
Data.normalizeworld(refpath)
#Data.interp(100,31,0)

#obtain rotation matrices
rmatrices = Data.rmatrices
rmatrices = np.array(rmatrices)

#obtain time
```

```
time = np.array(Data.time)

#Format time to time from drop in seconds
time = (np.array(time)-time[0])/1000000

#placeholders
ims = []

stuff = []
stuffy = []
stuffz = []

#Get frame
data = CheckOrientationV2(np.array(rmatrices[0]), 0, 0, 0, 0)

#Save frame in temp file
data.savefig('temp/temp.png')
img = cv2.imread('temp/temp.png')

#Get frame dimensions
height,width,channels = img.shape

# initialize video writer
fourcc = cv2.VideoWriter_fourcc('M','J','P','G')
fps = 30

#file output
video_filename = 'Videos/{}.avi'.format(filename+'int')
out = cv2.VideoWriter(video_filename, fourcc, fps, (width, height))

#For every sample, save orientation as frame with formatting
for i in range(0,time.size):

   data = CheckOrientationV2(np.array(rmatrices[i]),0,0,0,0)
   data.text(0.8,0.8,'t = {}s'.format(time[i]))
   data.suptitle('Vehicle Orientation for flap', y=0.9)

   data.savefig('temp/temp.png')
   img = cv2.imread('temp/temp.png')
   out.write(img)
   plt.close()

#Close videowriter
out.release()
```

APPENDIX F– MODELING CODE

Preliminary Force Model

```
from catboxdynamics import *
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path
from scipy import integrate
from ImportFallData import *
import csv

#Creates dynamic model of catbox using torque

refpath = "K:\CppThesis\ProtoData\Feb21_19\Baseline.pickle"

Ri, Ri_inv = pickle.load(open(refpath, "rb"))

data_folder = Path("K:\CppThesis\ProtoData")
subfolder = "Feb21_19/"

filepath2 = data_folder / (subfolder + "Feb21_19 No Rotation 2.dat")

Data2 = Data(filepath2)
Data2.droponly()
Data2.normalizeworld(refpath)
Data2.interp(100,21,0)
time2 = np.array(Data2.time)
time2 = (np.array(time2)-time2[0])/1000000


offtest = np.array([[0.9975,-0.0499,0.04997],
                    [0.0524115,0.9973,-0.04992],
                    [-0.04735,0.052412,0.99750]])

snaps = []
gtime = []

Ixx = 0.003685
Iyy = 0.003477
Izz = 0.003706

Ixy = -0.000121
Iyz = -0.000263
Izx = -0.000144

#Inertia Tensor of catbox in body frame
I_matrix = np.array([[Ixx, Ixy, Izx],
                     [Ixy, Iyy, Iyz],
                     [Izx, Iyz, Izz]])
```

```
#Starting orientation

RinWorld = np.array([[ 0.98876234,  0.06565313,  0.13430821],
 [-0.05915998,  0.99690457, -0.05178198],
 [-0.13729212,  0.0432544,   0.98958574]])

#Starting angular velocity
initangv = [[0.2998],[-0.22432],[0.1532]]

#Intialize angular velocity array
omega = np.array(initangv)

#Starting torque
torque0,area = catboxdynamics(RinWorld,0,0,0,0,0)

#Initialize loop variable
torque_old = np.array([[0],[0],[0]])

#angmom = [np.array([[0],[0],[0]])]

I_inworld0 = np.dot(RinWorld, np.dot(I_matrix, np.transpose(RinWorld)))

#Starting angular momentum
angmom = [np.dot(I_inworld0,omega)]

#Time interval
time = 1.54

#time step for integration
dt = 0.0001

#Number of times to loop
elements = np.int(time/dt)

#Starting orientation in loop
orientation = RinWorld

count = 0
time = [0]

#Calculates angular velocity, angular momentum as function of time
for i in range(1,elements):
   t = i*dt
   torque,totalarea = catboxdynamics(orientation,0,0,0,0,t)

   #integrates torque to obtain angular momentum
   amoment= (torque+torque_old)*dt/2 + angmom[i-1]

   #Changes inertia tensor to world frame
   I_inworld = np.dot(orientation, np.dot(I_matrix, np.transpose(orientation)))
```

```
    #change step for integration
    torque_old = torque

    #angular velocity from inertia tensor and angular momentum
    omegat = np.array(np.dot(np.linalg.inv(I_inworld),amoment))

    #norm of angular velocity
    n_omega = np.linalg.norm(omegat)

    #unit vectors of angular velocity
    k = np.transpose(omegat/n_omega)[0]

    #Skew-symmetric from of angular velocity
    Kmatrix = np.array([[0, -k[2], k[1]],
                        [k[2], 0, -k[0]],
                        [-k[1], k[0], 0]])

    #1st term of Rodriguez equation
    rtf = np.add(np.identity(3),np.sin(n_omega*dt)*Kmatrix)


    #Rodrigues equation for rotation of rotation matrix through angle theta
    Rdt = np.add(rtf,(1-np.cos(n_omega*dt))*np.dot(Kmatrix,Kmatrix))

    #Updated orientation from application of angular velocity
    orientation = np.array(np.dot(orientation,Rdt))



    #log calculated angular velocity
    omega = np.append(omega,omegat,axis=1)


    #log calculated angular momentum
    angmom.append(amoment)


    time.append(t)


    if t > count:
        snaps.append(orientation)
        gtime.append(t)
        count = count + 0.013

angvels = omega
angmom = angmom


#Uses pickle library to save matrices as variables for later use
with open('timerots',"wb") as f:
    pickle.dump([gtime,snaps],f)
```

```
ploteuler = 1
plotvel = 0
if ploteuler:

    euler = np.transpose(Data2.euler)
    eulerx = euler[0]
    eulery = euler[1]
    eulerz = euler[2]
    print(Data2.euler)

    rotations = R.from_dcm(snaps)
    euler_m = rotations.as_euler('xyz')

    euler_m = np.transpose(euler_m)
    print(euler_m)
    euler_mx = euler_m[0]
    euler_my = euler_m[1]
    euler_mz = euler_m[2]

    plt.plot(gtime,euler_mx,'r',label = 'x simulated',linestyle = '--')
    plt.plot(gtime,euler_my,'g',label = 'y simulated',linestyle = '--')
    plt.plot(gtime,euler_mz,'b',label = 'z simulated',linestyle = '--')


    plt.plot(time2,eulerx,'r',label = 'x data')
    plt.plot(time2,eulery,'g',label = 'y data')
    plt.plot(time2,eulerz,'b',label = 'z data')


    plt.title('Euler angles(xyz) for No Rotation Drop 2 Experimental vs Force Model')
    plt.xlabel('time (s)')
    plt.ylabel('Angle (rad)')
    plt.legend()

elif plotvel:

    print(angvels)

    plt.plot(time,angvels[0],'r',label = 'x simulated',linestyle = '--')
    plt.plot(time,angvels[1],'g',label = 'y simulated',linestyle = '--')
    plt.plot(time,angvels[2],'b',label = 'z simulated',linestyle = '--')
    #plt.plot(time,angmom[0][0])

    plt.plot(time2,Data2.xangvel,'r',label = 'x data')
    plt.plot(time2,Data2.yangvel,'g',label = 'y data')
    plt.plot(time2,Data2.zangvel,'b',label = 'z data')


    plt.title('Angular Velocities for No Rotation Drop 2 Experimental vs Force Model')
    plt.xlabel('time (s)')
    plt.ylabel('angular velocity (rad/s)')
    plt.legend()
```

plt.show()

Dynamics for Force Model

```
import pickle
from CheckOrientationV2 import CheckOrientationV2
from AreaCalcs import *

#Preliminary evaluation of box dynamics with given orientations and times

def catboxdynamics(RinWorld,a1,a2,a3,a4,t):

    rho = 1.225

    Cd = 1.05
    Cl = 2*np.pi

    m = .558

    g = 9.80665

    Ixx = 0.003685
    Iyy = 0.003477
    Izz = 0.003706

    Ixy = -0.000121
    Iyz = -0.000263
    Izx = -0.000144

    def terminalv(m, g, Cd, rho, totalarea):
        return np.sqrt((2 * m * g) / (Cd * rho * totalarea))

    def v(t,totalarea):
        term = terminalv(m, g, Cd, rho, totalarea)
        # print('terminal velocity:{} '.format(term))
        return term * np.tanh(g * t / term)

    I_matrix = np.array([[Ixx, Ixy, Izx],
                         [Ixy, Iyy, Iyz],
                         [Izx, Iyz, Izz]])

    I_inworld = np.array(np.dot(RinWorld,np.dot(I_matrix,np.transpose(RinWorld))))

    T_cmb = np.array([[1, 0, 0, 0],
                      [0, 1, 0, 0],
                      [0, 0, 1, .00159],
                      [0, 0, 0, 1]])

    T_bcm = np.linalg.inv(T_cmb)

    instance = AreaCalcs(RinWorld,a1,a2,a3,a4)
```

```
area = np.array(instance.area)

totalarea = np.sum(area)

dragF = np.zeros(area.shape)
liftF = np.zeros(area.shape)

centroids = instance.centroids
liftvecs = instance.liftvecs

dragmoment = np.zeros(centroids.shape)
liftmoment = np.zeros(centroids.shape)

velocity2 = v(t,0.0412902)**2
#print(velocity2)

for i,centroid in enumerate(centroids):

    centroid4 = np.append(centroid,[1])

    posvec = np.dot(T_bcm,centroid4)
    posvec = posvec[0:3]
    dragF[i] = (1/2)*rho*Cd*area[i]*velocity2
    liftF[i] = (1/2)*rho*Cl*area[i]*velocity2
    #print(dragF)

    dragmoment[i] = np.transpose(np.cross(np.transpose(posvec), dragF[i] * [0, 1, 0]))
    liftmoment[i] = np.transpose(np.cross(np.transpose(posvec), liftF[i] * np.transpose(liftvecs[i])))

#print(moment)
totalmoment = np.sum(np.add(np.array(dragmoment),np.array(liftmoment)),axis = 0)
#print(totalmoment)
ang_Accel = np.dot(np.linalg.inv(I_inworld),totalmoment)


# print(totalmoment)
return totalmoment,totalarea
```