MODELING AND SIMULATION OF A DISTRIBUTED MULTI-BEHAVIORAL CONTROL FOR FULLY AUTONOMOUS QUADROTORS

by

CHRISTIAN MICKLISCH and NATHAN HILLIARD

Advisor
DR. HALA ELAARAG

A senior research paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in the Department of Mathematics and Computer Science in the College of Arts and Science at Stetson University DeLand, Florida

Spring Term 2015
Abstract

In this paper we present a real time scalable and adaptable system to control and coordinate the movements of fully autonomous quadrotors. Flocking describes the movement of a group of agents such as birds, insects, mammals, or robots cohesively traveling in a coordinated way. A fully autonomous quadrotor is an aerial drone that utilizes four propellers to achieve stable flight through the use of obstacle avoidance, attitude control, and altitude control algorithms. In this research we created Avian; a group of coordinated autonomous quadrotors that employs parallel computing and dynamic behavior application to formulate a real time scalable and adaptable system. We present a comprehensive mathematical model for the Avian system. We also develop a distributed simulator to accurately show the capabilities of the attitude controllers, altitude controller, and collision detection methods by accurately simulating the environment, the reactions of the sensors in the environment, and how the quadrotors utilize the sensor data to interact in the environment. We developed several obstacle detection methods and an altitude controller while simulating several algorithms for obstacle avoidance, attitude and altitude control and flocking techniques. The simulator allows for the comparison of the performance of different configurations and control of quadrotor systems in various environments.
1. Introduction

In most previous research, quadrotors and flocking were designed with a static system. Communication and control were centralized through a base station. Avian, however, provides a dynamic system that implements a multi-behavioral system to control multiple fully autonomous quadrotors. The system is dynamic in the sense that one can add more members to the group at any time, without the need to reconfigure or restart the system, as well as change which behaviors certain quadrotors are applying and which they aren’t.

The key benefits of this model are that each quadrotor is capable of changing its current way of behaving at any time. Rather than the base station applying rules to the entire set of quadrotors collectively, each quadrotor can dynamically enable, disable, edit the parameters of, and apply individual behaviors at will without affecting the other quadrotors.

In order to prove the feasibility of this system, a simulation was created based on the mathematical models for a quadrotor, obstacle detection and avoidance, attitude and altitude control, and various behaviors. To create a realistic virtual environment, special care was taken to ensure that basic laws of physics are applied with a large number of objects running in parallel to ensure smooth execution.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 explains the mathematical model of the autonomous quadrotor, the algorithms used for obstacle avoidance, attitude control, altitude control, and flocking. There are several algorithms for each of these components, allowing us to choose which one will work best for
our system. Section 4 discusses the simulation, Avian, the details developed for testing the feasibility of using cloud computing for flocking with autonomous quadrotors, and the performance of the various algorithms described in section 3. Section 5 shows the results of the algorithms described in section 3 by using the simulation described in section 4. Section 6 discusses simulation verification. Finally section 7 concludes the paper.
2. Related Works

There are several proposals in the literature that attempt to create quadrotors that are controlled by separate units. The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control (STARMAC) is one of the more notable ones [1, 2, 3]. STARMAC seems to have a host of foci. Two of the more important ones are the research on aggressive maneuvering [1], and acrobatics in a real life setting [2]. STARMAC’s actual control and decision making was done via a computer which was used to notify the quadrotor to fill in the parameters for the actions it needed to take at that moment e.g. how high to fly, what position to end at, when to flip….etc. [2].

Grzonka et al. [4] created an autonomous indoor quadrotor that involved intent collision detection and mapping to provide itself with a proper route that would aid in reaching its destination. The control was done onboard the quadrotor, yet it was designed to get inputs via the user over a laptop feed as to where it needed to start and stop. Any complex calculations were done on an offboard computer that then returned the results for the next iteration of control.

Miller [5] designed and created a micro quadrotor that weighed about 70 grams, making it one of the lightest autonomous quadrotors. Millers design was to use 2 attitude control systems, one on board the quadrotor, and another system on a base station that would integrate with a camera to constantly analyze the position of the quadrotor. The quadrotor send the base station its current attitude feed from its Inertial Measurement Unit (IMU), and
then received a result defined by a fusion of both its own IMU feed and the feed produced by the camera analysis system.

Vasarhelyi et al. [6], used multiple quadrotors that had each quadrotor communicate with each other to form a small cluster of quadrotors. This allowed to build a flocking mechanism where each quadrotor knew where its neighbor was and therefore flew in a group. Each quadrotor did its own calculations on the movement that it needed to do in relation to the group. That is, each quadrotor did its own collision detection and flight path creation. A ground control station was used to control the entire group’s velocity and its final position.

Kushleyev et al. [7] designed a flock of 20 miniature quadrotors that did all of its navigational control calculations on a central computer off of the quadrotor. The central computer used a camera to track all of the current locations of each quadrotor. Each quadrotor only had a singular onboard computer and some sensors to track its attitude and adjust it to what the central computer specifies.

Oweis et al. [8] brought the flocking control of several aerial-systems to a single server by sending commands based on several flocking rules. The flocking rules observed are i.) “separation” which ensures that each unit does not get too close to other units near it to prevent crowding, ii.) “alignment” which makes sure that each unit generally matches the velocities of the units near it to ensure all units are moving in a common direction, iii.) “cohesion” which keeps each unit moving towards the average position of its neighboring units, iv.) “targeting” which forces each unit to move towards a target point in order to keep
the flock moving towards a common point, v.) “avoidance” which forces each unit to avoid a particular point in space making the point globally repulsive, which is similar to the separation rule.

Our research is set apart from previous research done in the field of controlling multiple quadrotors as ours not only utilizes a distributed backend to provide a dynamic and scalable control system but also operates with the ability for each quadrotor to selectively apply certain behaviors at any time. Our research also allows for all of this to be possible without the necessity for the Global Positioning System (GPS) to coordinate locations, as it is instead based off of the relative positioning calculated by the Inertial Measurement Unit, IMU. This allows for our model to operate indoors safely without dependency on a positioning system that is unreliable in scenarios that require secluded indoor or underground operation.

Our simulator, “Avian,” is set apart by all other quadrotor simulations as it allows us to simulate the different sensors and their outputs as if they were operating in real time. This allows us to design a more true to life simulator and multicopter implementation giving us the capability of seeing how the quadrotor will not only react in different environments but also with the limitations set by the different sensors utilized.
3. Mathematical Models

3.1 Mathematical Model of the Quadrotor

The quadrotor is a rigid frame with its center of gravity at its center. The control points are the motors, which are completely equidistant from each other in a square-like format. There are 2 sets of motors, 2 in each set, when placed on the frame the one set rotates opposite of the other and both set form a “+” shape shown in Figure 3.1.

The layout of the motors allows the direction of thrust to stay the same resulting in a much simpler mathematical representation and the rigidity of the quadrotor adds to its simplicity.

In relation to the environment that the quadrotor will be moving in it must be able to move linearly and angularly in a three-dimensional space. That means that it will need to move in the linear and angular directions as well. The linear directions will be defined by our cartesian coordinate system in which we will have 3 axes named x, y, and z. The z axis defines positions of height, the x axis defines positions of length, and the y axis is the width. The angular directions are defined as such: $\Phi$ is the roll angle around the x-axis, $\theta$ is the pitch angle around the y-axis, and $\psi$ is the yaw angle around the z-axis. Thus our coordinate system, $\mathbf{q}$, is then defined as shown in Equation 3.1 where $\eta$ is the linear coordinate system and $\xi$ is the angular coordinate system.
Using this format, when the quadrotor’s motors are aligned on the x-axis and y-axis where we assume that the front of the quadrotor is on the positive side of the x-axis, the back of the quadrotor is on the negative side of the x-axis, the right of the quadrotor is on the positive side of the y-axis, the left of the quadrotor is on the negative side of the y-axis, the top of the quadrotor is on the positive side of the z-axis, and the bottom side of the quadrotor is the negative side of the z-axis.

If the quadrotor wants to move forward then it must decrease the thrust of the motor on the positive side of the x-axis and increase the thrust of the motor on the negative side of the x-axis as shown in Figure 3.1.A. If the quadrotor wants to move right then it must decrease the thrust of the motor on the positive side of the x-axis and then increase the thrust of the motor on the negative side of the x-axis as shown in Figure 3.1.B. If the quadrotor wants to move upward then it must increase the thrust of all four motors, allowing it to move in the positive z-axis as shown in Figure 3.1.C. For the quadrotor to move clockwise or counterclockwise it depends on the direction the motors are spinning on the x-axis and y-axis. To turn the quadrotor clockwise the motors that move clockwise must increase and the motors that move counterclockwise must decrease. To move counter clockwise the motors moving clockwise must increase and the motors moving counterclockwise must decrease as shown in Figure 3.1.D. For the quadrotor to stably hover without any change in the $\phi$, $\theta$, or $\psi$ it must
have the motors moving at such a force that they counteract any external forces being acted on the quadrotor.

Figure 3.1: Rotational Quadrotor Movement

3.1.1 Forces and Moments

The quadrotor itself has several aerodynamic forces acting on itself. These forces are produced via a spinning rotor blade. Gary Fay [34] discussed aerodynamic forces while working on the Mesiopter project. In his research, he derived several forces specifically for quadrotors, based off of the aerodynamic effects caused by a helicopter and blade movement through flight. In [35] only 4 forces were implemented in the design and control of
autonomous quadrotors. In this research, we follow the suggestion in [35] and we implement
the thrust force, the hub force, the drag moment or the torques, and the rolling and pitching
moment.

3.1.1.1 Thrust Force

We calculate the thrust force as in [34] by integrating the vertical forces acting on all
the blade elements as shown in Equations 3.2-3.5. In Equation 3.2, \( T \) is the resulting thrust
force, \( C_T \) is the coefficient of the thrust force shown in Equation 3.3, \( \rho \) is the air density of
the surrounding air, \( A \) is the disk area of the propeller, \( \Omega \) is the angular rate of the propeller,
and \( R_{rad} \) is the rotors radius. In equation 3.3, \( \theta_{tw} \) is the twist pitch of the propeller, \( \theta_0 \) is the
pitch of incidence of the propeller, \( \lambda \) is the inflow ratio shown in Equation 3.4, \( \mu \) is the rotor
advance ratio shown in Equation 3.5, \( \sigma \) is the solidity ratio of the propeller, and \( a \) is the lift
slope of the propeller.

\[
T = C_T \rho A (\Omega R_{rad})^2 \tag{3.2}
\]

\[
\frac{C_T}{\sigma a} = \left( \frac{1}{6} + \frac{1}{4} \mu^2 \right) \theta_0 - \left( 1 + \mu^2 \right) \frac{\theta_{tw}}{8} - \frac{1}{4} \lambda \tag{3.3}
\]

In Equation 3.4 \( v_1 \) is the inflow velocity, \( z \) is the linear velocity in the z direction, \( \Omega \)
in this case is the speed of the propeller, and \( R \) is the radius of the propeller. In relation to the
inflow velocity below, \( W \) is the weight of the quadrotor in relation to the thrust, where \( d \) is
the diameter of the propeller, \( \Omega \) is again the speed of the propeller, \( P_c \) is the power constant,
and \( F \) is the power factor.
\[ \lambda = \frac{v_1 - \dot{z}}{\Omega R} \]  

(3.4)

where,

\[ v_1 = \sqrt{-\frac{V^2}{2} + \sqrt{\left(\frac{V^2}{2}\right)^2 + \left(\frac{W}{2\rho A}\right)^2}} \]

\[ W = \left(\frac{\pi d^2 A}{2} \left(P_c \Omega^F\right)\right)^{\frac{1}{3}} \]

In Equation 3.5 \( V \) is the horizontal velocity, where \( \dot{x} \) is the velocity in the x direction and \( \dot{y} \) is the velocity in the y direction.

\[ \mu = \frac{V}{\Omega R} \]  

(3.5)

where,

\[ V = \sqrt{\dot{x}^2 + \dot{y}^2} \]

3.1.1.2 Hub Force

The hub force is calculated by integrating all of the horizontal forces acting on all the blade elements as shown in Equation 3.6 [34]. In this equation, \( H \) is the total hub force acting on the system, \( \Omega \) is again the angular rate of the propeller, \( C'_H \) is the coefficient of the hub force as shown in Equation 3.7, and \( C'_d \) is the drag coefficient of the blades at 70% radial station [35].
3.1.1.3 Drag Moment

The drag moment, or the torque, of the quadrotor equates to all of the aerodynamic forces occurring around the center of the rotor as shown in Equation 3.8 [34]. This allows us to determine the amount of power that is required to spin the motor. In this equation, $Q$ is the total torque or drag moment, $\Omega$ is the angular rate of the propeller, and $C_Q$ is the drag moment coefficient shown in Equation 3.9 [35].

$$Q = C_Q \rho A \left( \Omega R_{rad} \right)^2 R_{rad}$$  \hspace{1cm} (3.8)

$$C_Q = \frac{1}{\sigma a} \frac{1}{8a} \left( 1 + \mu^2 \right) C_d + \frac{\lambda}{4} \left( \frac{\theta_0}{2} \theta_{tw} - \frac{\theta_{tw}}{2} \right)$$  \hspace{1cm} (3.9)

3.1.1.3 Rolling Moment

The rolling moment occurs during forward flight when the “advancing blade is producing more lift than the retreating blade,” as in Equation 3.10. In this equation $R_m$ is the rolling moment in equation 3.11, $\Omega$ is the angular rate of the propeller, and $C_{R_m}$ is the rolling moments coefficient [35].
\[ R_m = C_{Rm} \rho A \left( \Omega R_{rad} \right)^2 R_{rad} \]  \hfill (3.10)

\[ \frac{C_{Rm}}{\sigma a} = -\mu \left( \frac{1}{6} \theta_0 - \frac{1}{8} \theta_{tw} - \frac{1}{8} \lambda \right) \]  \hfill (3.11)

### 3.1.2 Moment Of Inertia

The structure of the frame is assumed to be a cuboid in the center with two perpendicular cylinders crossing each other in the center of the cuboid. The moment of inertia of the two cylinders are represented in Equation 3.12 [12], where \( l \) is the length of the cylinder, and \( m_r \) is the mass of the two cylinders. The moment of inertia of the cuboid is represented in Equation 3.13, where \( h \) is the height, \( w \) is the width, \( l \) is the length, and \( m_c \) is the mass of the cuboid. The complete inertial frame is then a combination of \( I_r \) and \( I_c \) as shown in Equation 3.14. A simplified version is shown in Equation 3.15.

\[
I_r = \begin{bmatrix}
\frac{1}{12} m_r l^2 & 0 & 0 \\
0 & \frac{1}{12} m_r l^2 & 0 \\
0 & 0 & \frac{1}{12} m_r l^2 \\
\end{bmatrix}
\]  \hfill (3.12)

\[
I_c = \begin{bmatrix}
\frac{1}{12} m_c (h^2 + w^2) & 0 & 0 \\
0 & \frac{1}{12} m_c (h^2 + l^2) & 0 \\
0 & 0 & \frac{1}{12} m_c (w^2 + l^2) \\
\end{bmatrix}
\]  \hfill (3.13)

\[
I = I_c + I_r = \\
\begin{bmatrix}
\frac{1}{12} m_c (h^2 + l^2) + \frac{1}{12} m_r l^2 & 0 & 0 \\
0 & \frac{1}{12} m_c (h^2 + l^2) + \frac{1}{12} m_r l^2 & 0 \\
0 & 0 & \frac{1}{12} m_c (w^2 + l^2) + \frac{1}{12} m_r l^2 \\
\end{bmatrix}
\]  \hfill (3.14)
\[ I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \]  

(3.15)

where,

\[ I_{xx} = \frac{1}{12} m_c (h^2 + w^2) + \frac{1}{12} \frac{m_r}{2} l^2 \]

\[ I_{yy} = \frac{1}{12} m_c (h^2 + l^2) + \frac{1}{12} \frac{m_r}{2} l^2 \]

\[ I_{zz} = \frac{1}{12} m_c (w^2 + l^2) + \frac{1}{12} m_r l^2 \]

3.1.3 Equations of motion

The equations of motion are defined by the equations for acceleration given by Samir Bouabdallah [35] in equations 3.16 - 3.21 using the equations described above from 3.2-3.15. \( A_c \) is the surface area of the quadrotor, \( C_x \) and \( C_y \) are the friction constants in the x and y directions, \( h \) is the vertical height of the center quadrotor to the center of the propellers, and \( l \) is the horizontal length from the center of the quadrotor to the center of the propeller.

\[ \ddot{x} = \frac{(\sin(\psi) \sin(\phi) + \cos(\psi) \sin(\theta) \cos(\phi)) \sum_{i=1}^{4} T_i - \sum_{i=1}^{4} H_x i - \frac{1}{2} C_x A_c \beta \dot{x}}{m} \]

(3.16)

\[ \ddot{y} = \frac{(-\cos(\psi) \sin(\phi) + \sin(\psi) \sin(\theta) \cos(\phi)) \sum_{i=1}^{4} T_i - \sum_{i=1}^{4} H_x i - \frac{1}{2} C_y A_c \beta \dot{y}}{m} \]

(3.17)
3.2 Attitude Control

There has been much research in the best forms of attitude control of the quadrotor, that is, how the quadrotor is controlled via an input and what its reactions are to be. Comparison of the best method is key to building a proper controlling mechanism tailored specifically to the quadrotors design as each attitude control method has its up and down sides.

Kemper [12] concludes that the best control method is a combination of the PID and PD controller as its attitude control algorithm. Hoffman et al. [3] also utilized a PID and PD controller but used the PID controller for low velocity flight and the PD controller for high velocity flight. Dikmen et al. [17] compared several attitude controllers and decided that the sliding mode technique had the best performance "especially on higher initial conditions."
3.2.1 Attitude Control method 1

In this method, the quadrotor requires 3 different PID controllers for the $x$, $y$, and $z$ linear positions [12]. The $\phi$, $\theta$, and $\psi$ angular coordinates are controlled by their own respective PD controllers. The controller itself runs as follows:

1. The position control expressions give the ‘commanded’ linear accelerations that are required to drive the system to the desired state.
2. Given the commanded linear accelerations, the necessary total thrust, pitch, and roll are determined.
3. The commanded torques about the three axes of the quadrotor are given by the PD controllers using the commanded yaw, pitch, and roll as angular set points.
4. Given the commanded total thrust and total torques, the motor speeds can be determined.
5. Once the motor speeds are known the system model can be used to obtain the updated state of the system.
6. Repeat steps 1-5.

The end result of this attitude control algorithm is to get the speeds with which to control the motors.

In the case of the PID model Equation 3.21 shows the current position of the quadrotor, $P$, and the desired location of the quadrotor, $P_c$. 
The Equation for the PID controller is shown in Equation 3.22 where \( \ddot{P}_c \) is the vector of commanded accelerations. \( K_p, K_i, \) and \( K_d \) are the proportional, integral, and derivative controllers, respectively, each in the form of a 3 \( \times \) 1 matrix.

\[
\ddot{P}_c = K_p(P_c - P) + K_i \sum_k (P_c - P) + K_d(\dot{P}_c - \dot{P})
\]  
\( (3.22) \)

For the PD controllers, a relation from the linear to the angular coordinate system is used to derive Equations 3.23-3.25, where, \( k_{pa} \) and \( k_{da} \) are the proportional and derivative constants, respectively, while \( a \) is a representation of an angular coordinate. \( \theta_c, \phi_c, \) and \( T_c, \) are the commanded roll, pitch and total thrust, respectively.

\[
\tau_{\phi_c} = \left[ k_{pa}(\phi_c - \phi) + k_{da}(\dot{\phi}_c - \dot{\phi}) \right] I_{x1}
\]  
\( (3.23) \)

\[
\tau_{\theta_c} = \left[ k_{pa}(\theta_c - \theta) + k_{da}(\dot{\theta}_c - \dot{\theta}) \right] I_{y_b}
\]  
\( (3.24) \)

\[
\tau_{\psi_c} = \left[ k_{pa}(\psi_c - \psi) + k_{da}(\dot{\psi}_c - \dot{\psi}) \right] I_{z_z}
\]  
\( (3.25) \)

where,

\[
\theta_c = \arctan\left( \frac{a_x \psi + a_y S_{\psi} + g \psi}{a_x + g} \right)
\]

\[
\phi_c = \arcsin\left( \frac{a_x S_{\psi} - a_y C_{\psi} \sqrt{a_x^2 + a_y^2 + (a_x + g)^2}}{\sqrt{a_x^2 + a_y^2 + (a_x + g)^2}} \right)
\]
The motor output given from the knowledge of steps 1-3 are represented in Equations 3.26-3.29 [12].

\[ T_c = \sqrt{a_x^2 + a_y^2 + (a_z + g)^2} * m \]

\[ w_{1c} = \sqrt{\frac{T_c}{4k} - \frac{\tau_{\theta c}}{2kL} - \frac{\tau_{\psi c}}{4b}} \]  \hspace{1cm} (3.26)

\[ w_{2c} = \sqrt{\frac{T_c}{4k} - \frac{\tau_{\theta c}}{2kL} + \frac{\tau_{\psi c}}{4b}} \]  \hspace{1cm} (3.27)

\[ w_{3c} = \sqrt{\frac{T_c}{4k} + \frac{\tau_{\theta c}}{2kL} - \frac{\tau_{\psi c}}{4b}} \]  \hspace{1cm} (3.28)

\[ w_{4c} = \sqrt{\frac{T_c}{4k} + \frac{\tau_{\theta c}}{2kL} + \frac{\tau_{\psi c}}{4b}} \]  \hspace{1cm} (3.29)

### 3.2.2 Attitude Control method 2

For the attitude control, Hoffman et al.’s [3] used a PID controller such as Equation 22 in low velocity indoor flight. Their results stated that there was around a 2° – 3° margin of error in the yaw. For faster outdoor flight Hoffman et al. [3] used a PD controller as shown in Equation 3.30. Hoffman et al. explained that initially the PD controller is “sufficient enough to bring the” quadrotor “toward the
commanded pitch.” As the speed increased the blade flapping caused the restoring moments to increase creating unrest in the quadrotor flight.

\[
\dot{P}_c = K_p(P_c - P) + K_d(\dot{P}_c - \dot{P}) 
\]  
(3.30)

The motor output is then computed similarly to Equations 3.26-3.29 where the commanded pitch, roll, and total thrust are found from the linear accelerations found in Equation 3.30.

3.2.3 Attitude Control method 3

Dikmen et al. [17] compared several different methods for attitude control, the PD, inverse, backstepping, and sliding mode controller. They concluded that the controller with the best performance is the sliding mode control as it has better performance and responds better to higher initial conditions. “The sliding mode approach is a method which transformed a higher-order system into a first-order system.” “The Lyapunov stability method is applied to keep the nonlinear system under control.” Dikmen et al. [17] define the system in Equation 3.31 and the time derivative in Equation 3.32. The relation to the stable space of the system in shown in Equation 3.33 and the time derivative in 3.34, while the relation of the stable space to the Lyapunov function is in Equation 3.35 and the time derivative in Equation 3.36.

\[
\begin{align*}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= f(x) + g(x)U \\
s &= x_2 + Cx_1
\end{align*}
\] 
(3.31) \hspace{1cm} (3.32) \hspace{1cm} (3.33)
\[ \dot{s} = \dot{x}_2 + C\dot{x}_1 = f(x) + g(x)U + Cx_2 \]  
\[ V = \frac{1}{2} s^2 \]  
\[ V = s\dot{s} = s[f(x) + g(x)U + Cx_2] \]

The stability of the system follows the standards shown in Equation 3.37 where \( s \) is the change of the system. The control law then follows in Equation 3.38.

\[ U = \begin{cases} 
< \beta(x) \text{for} s > 0 \\
- \beta(x) \text{for} s = 0 \\
> \beta(x) \text{for} s < 0 
\end{cases} \]

where,

\[ \beta(x) = -\frac{f(x) + Cx_2}{g(x)} \]

\[ U = \beta(x) - K \text{sgn}(s), \quad K > 0 \]

The application of the sliding mode control equations to the quadrotor system where \( U \) is split into 3 parts to show the total thrust of the motors, the roll movement from motors 2 and 4 and the pitch movement of motors 1 and 3. This is represented as \( U_1, U_2, \) and \( U_3 \) shown in Equations 3.39, 3.40, and 3.41, respectively, where \( \sigma_\phi \) is the representation of the stable space for the roll angle, \( \phi \). Similarly are the representations of the stable space for both the pitch and yaw angles respectively.
The algorithm only focuses on the control of the thrust, roll, and pitch. It does not consider the yaw angle under the attitude control. The algorithm for Dikmen et al.’s sliding method is then as follows:

1. Get the values of the angular positions and their angular rates

\[
U_1 = \beta(x) - K_1 \text{sgn}(\sigma_\phi) \\
U_2 = \beta(y) - K_2 \text{sgn}(\sigma_\theta) \\
U_3 = \beta(z) - K_3 \text{sgn}(\sigma_\psi)
\] 

where,

\[
\sigma_\phi = (\phi_d - \dot{\phi})S_1 + (\phi_d - \dot{\phi}) \\
\sigma_\theta = (\theta_d - \theta)S_3 + (\theta_d - \dot{\theta}) \\
\sigma_\psi = (\psi_d - \dot{\psi})S_5 + (\psi_d - \dot{\psi})
\]

\[
\beta(x) = -\frac{a_1 \ddot{\phi} \psi + a_2 \dot{\Omega}_r + S_2 \dot{\phi}}{b_1} \\
\beta(y) = -\frac{a_3 \dot{\phi} \psi + a_4 \dot{\phi} \Omega_r + S_4 \dot{\theta}}{b_2} \\
\beta(z) = -\frac{a_5 \dot{\theta} \dot{\phi} + S_6 \dot{\psi}}{b_3}
\]
3. Convert the positions to the outputs of the motors.

4. Repeat steps 1-3.

3.3 Obstacle Detection

Obstacle detection is the method in which the quadrotor detects an obstacle through the means of a sensor and then avoids that obstacle.

Several Researchers investigated the problem of obstacle detection, whether it be outdoor or indoor. Prashnath et al. used a visual based Optical Flow [13] technique for obstacle avoidance. Wagster et al. [14] and Gageick et al. [15] used an assortment of ultrasonic sensors to detect and avoid obstacles. In our research we focus on the use of ultrasonic sensors as they perform better than visual based obstacle detection when there are moments of visual hindrance such as fog. Ultrasonic sensors are also more cost effective than the visual based sensors used by Prashnath et al. and does not need as much onboard computation time to react to a potential collision forcing the use of more powerful and more expensive processing hardware.

In this section we describe two methods of obstacle detection improving on the design of Gageick and Wagsters implementation.
3.3.1 Obstacle detection method 1

For obstacle detection Gageick et al.’s [15] used a total of 12 ultrasonic sensors placed in a circle parallel to the horizontal portion of the quadrotor frame.

\[
State = \begin{cases} 
  \text{safe} & \text{where } a + b < m \\
  \text{close} & \text{where } a + b > m \\
  \text{dangerous} & \text{where } a > m 
\end{cases}
\] (3.42)

The collision avoidance uses the data from the ultrasonic sensors and creates zones; safe, close, and dangerous, based off of the distances that the ultrasonic sensors send as shown in Equation 3.42, where \( m \) is the distance received from the ultrasonic sensors, \( a \) is a close range constant and \( b \) is a long range constant.

From these zones, the quadrotor acts as a state machine, where the state describes the movement of the quadrotor. If the zone is a safe zone it is free to move, when it’s a close zone then the machine has limited movement in regards to that close state. The closer the quadrotor is to that close zone the more limited movement it has. If the state is in a danger zone then all movement is limited as it prevents any further approaches to the obstacle causing it to be a danger zone.

We improved on the Obstacle Detection method by Gageick et al. when an obstacle is obstructing the quadrotor’s desired location, the quadrotor moves away from it instead of stopping as happens with Gageick’s method. In our simulator, when the quadrotor’s obstacle
detection is in a *dangerous* state the quadrotor moves away from the object to a *close* state. This ensures that if the quadrotor responds too late to a wind gust from the environment then it will completely avoid the collision. The pseudocode for our obstacle detection method is shown in figures 3.2 - 3.3.

```java
Position calculateSuggestedCommandedPosition()
{
    Position p = new Position();
    number distance = this.getLowestDistance();
    for (i = 0; i < DIMENSION; i++)
    {
        if (POINT_A > distance)
        {
            number difference = calculateDifference(i);
            p.setPoint(i, -1.0 * signum(difference) * (distance + difference) + getLinearPosition().getPoint(i));
        }
        else if (POINT_A + POINT_B > distance)
        {
            number difference = calculateDifference(i);
            p.setPoint(i, (REDUCE_MULTIPLIER * difference) + getLinearPosition().getPoint(i));
        }
        else if (POINT_A + POINT_B < distance)
        {
            p.setPoint(i, getCommandedLinearPosition().getPoint(i));
        }
    }
    return p;
}
```

**Figure 3.2:** Avian’s Obstacle detection and avoidance.

```java
number calculateDifference(number direction)
{
    number commanded = getCommandedLinearPosition().getPoint(direction);
    number current = getLinearPosition().getPoint(direction);
    return commanded - current;
}
```

**Figure 3.3:** Function for calculating the difference of the commanded and current position.
The distance variable is just the lowest distance of all of the running ultrasonic sensors. The difference variable is the current difference of the commanded position and the current linear position. The method focuses on simply modifying the commanded position, this allows us to use this method with any attitude controller. When the state is Dangerous it simply reverses the direction of the current commanded position from the current position of the multicopter. This allows for a slow and smooth movement away from the object. When the state is Close then the commanded position is simply changed by multiplying the difference of the commanded and current position with the REDUCE_MULTIPLIER allowing the quadrotor to slowly move towards the obstacle. If the state is Safe then nothing is done.

3.3.2 Obstacle detection method 2

In this method, Wagster et al. [14] utilize only 6 ultrasonic sensors, on the top, bottom, left, right, forward, and rear of the quadrotor. They use a three level avoidance system to “minimize risk to the” quadrotor and “ensure the extraction of useful information from research in the event the final goal was not reached.”

For the first level it uses a forward facing sensor to “determine presence of and distance to the obstacles in front of the vehicle.” There is then a minimum distance at which the quadrotor and the obstacle can be apart where the quadrotor will not collide with the obstacle ceasing all motion in the process. The algorithm used goes to increase the pitch angle causing the quadrotor to slow down.
The second level uses the left and right sensors of the vehicle to center the quadrotor between two obstacles using a push algorithm. The algorithm takes the difference of both distances from the sensors and converts it to a roll angle that increases as the vehicle gets closer to the obstacle.

The third level uses a modified Rapidly Exploring Random tree algorithm where the quadrotor “not only avoids the obstacle but also plans a path around it.” For Wagster et al. [14], the implementation based its flight path on all of the scores of the available movements. The setup they chose was to limit any form of movement on the quadrotor where it calculates the cost of the waypoint and then chooses the waypoint with the lowest score, thereby the closest target.

Our improvements to Wagsters obstacle detection method is to be able to utilize an unlimited number of ultrasonic sensors, determine the necessary sensors based off of the heading of the quadcopter and the commanded position, and control the movement of the quadrotor using the commanded positions allowing us to implement any attitude controller. Our focus was only on levels 1 and 2 as manipulating the flight path based off of prior knowledge is not apart of obstacle detection but more a flight path problem.

To get the implementation of Wagster to work with an unlimited amount of sensors we decided to create a method that would allow us to get the “front,” “left,” and “right” sensors simply based on the commanded position of the quadrotor, where the pseudocode is shown in figures 3.4 - 3.5.
number getUltrasonicIndex(Position commandedPos) {
    number lowest = NO_VALUE;
    for (number i = 1; i < ULTRASONIC_SENSOR_COUNT; i++) {
        if (getSensorPosDifference(i, commandedPos) < getSensorPosDifference(lowest, commandedPos))
            lowest = i;
    }
    return lowest;
}

Figure 3.4: Pseudocode for getting the sensor index based off of the commanded position.

number getSensorPosDifference(number index, Position commandedPos) {
    if (index == NO_VALUE)
        return MAX_ANGLE;

    number xdiff = getSensor(index).getLinearPos().getPoint(VALUE_X) - currentLinearPos.getPoint(VALUE_X);
    number ydiff = getSensor(index).getLinearPos().getPoint(VALUE_Y) - currentLinearPos.getPoint(VALUE_Y);
    number xdiffDest = commandedPos.getPoint(VALUE_X) - currentLinearPos.getPoint(VALUE_X);
    number ydiffDest = commandedPos.getPoint(VALUE_Y) - currentLinearPos.getPoint(VALUE_Y);

    number degree = atan2(xdiff, ydiff));
    number degreeDest = tan2(xdiffDest, ydiffDest));

    if (degree < 0)
        degree = degree + MAX_ANGLE;
    if (degreeDest < 0)
        degreeDest = degreeDest + MAX_ANGLE;

    return abs(degree - degreeDest) % MAX_ANGLE;
}

Figure 3.5: Pseudocode for getting the difference in angular position of the sensor compared to the angular position of the commanded position.
The implementation of the first level is shown in figure 3.6 where the obstacle
detection method is to determine the distance of an obstacle in front of itself. Here we define
front by the direction of the commanded position. We then get the front facing sensor using
our getUltrasonicIndex method shown in figure 3.4. Once we have the front facing sensor we then
get the value of that sensor and see if it is below the \textit{MINIMUM \_ PHASE \_ ONE} value, if it is then we
set the commanded position to be our current position thereby avoiding potential collisions.

\begin{verbatim}
Position phaseOne(Position p)
{
  int frontSensorIndex = getUltrasonicIndex(getCommandedLinearPos());
  if (frontSensorIndex == NO \_ VALUE)
      return p;
  double frontValue = getSensorValue(frontSensorIndex);
  if (frontValue < MINIMUM \_ PHASE \_ ONE)
  {
    p.setPoint(VALUE \_ X, getCurrentLinearPos().getPoint(VALUE \_ X));
    p.setPoint(VALUE \_ Y, getCurrentLinearPos().getPoint(VALUE \_ Y));
    p.setPoint(VALUE \_ Z, getCurrentLinearPos().getPoint(VALUE \_ Z));
  }
  return p;
}
\end{verbatim}

\textbf{Figure 3.6: Avian’s Phase One pseudocode.}

The implementation of the second level is shown in figure 3.7 where the obstacle
detection method places the multicopter in the center of two obstacles facing each other. In
the case of the second phase the commanded position and rotates the point 90 degrees to the
right to get the right ultrasonic sensor of the multicopter and 90 degrees left to get the left
sensor of the multicopter. It then checks to see if the values of the are not equal, as then the
multicopter would be in the center of the two opposing obstacles, and if the left and right
sensor outputs are below the \textit{MINIMUM \_ PHASE \_ TWO} value. This value is simply the maximum
distance that the ultrasonic sensor can detect. If the three requirements hold true then the new commanded position is calculated to be the center of the two obstacles by first calculating the necessary change of the multicopter based off of the difference in the right and left sensor output, and rotating that difference around the PHASE_TWO_PIVOT which is zero. This method allows the quadrotor to properly avoid the two opposing obstacles on its right and left side depending on the direction the quadrotor is moving in.

```java
Position phaseTwo(Position p)
{
    Position pLeft = new Position();
    pLeft.setPoint(Environment.VALUE_X, (cos(90.0) * getCommandedLinearPos().getPoint(VALUE_X)) - (sin(90.0) * getCommandedLinearPos().getPoint(VALUE_Y)));
    pLeft.setPoint(Environment.VALUE_Y, (sin(90.0) * getCommandedLinearPos().getPoint(VALUE_X)) + (cos(90.0) * getCommandedLinearPos().getPoint(VALUE_Y)));

    Position pRight = new Position();
    pRight.setPoint(Environment.VALUE_X, (cos(-90.0) * getCommandedLinearPos().getPoint(VALUE_X)) - (sin(-90.0) * getCommandedLinearPos().getPoint(VALUE_Y)));
    pRight.setPoint(Environment.VALUE_Y, (sin(-90.0) * getCommandedLinearPos().getPoint(VALUE_X)) + (cos(-90.0) * getCommandedLinearPos().getPoint(VALUE_Y)));

    number leftSensor = getUltrasonicIndex(pLeft);
    number rightSensor = getUltrasonicIndex(pRight);

    if (leftSensor == NO_VALUE || rightSensor == NO_VALUE)
        return p;

    number leftValue = getUltrasonicSensorValue(leftSensor);
    number rightValue = getUltrasonicSensorValue(rightSensor);

    if (leftValue != rightValue
        && leftValue < MINIMUM_PHASE_TWO
        && rightValue < MINIMUM_PHASE_TWO)
    {
        number difference = rightValue - leftValue;
        number xPosD = abs(difference) * signum(getCommandedLinearPos().getPoint(VALUE_X));
        number yPosD = abs(difference) * signum(getCommandedLinearPos().getPoint(VALUE_Y));
        number zPosD = 0.0;
        Position d = new Position(xPosD, yPosD, zPosD);
    }
```

30
number initAngle =
    getUltrasonicSensor(frontSensor).getAngularPos().getPoint(VALUE_PSI);

if (difference < PHASE_TWO_PIVOT)
    number diffAngle = -90.0;
else
    number diffAngle = 90.0;

number finalAngle = initAngle + diffAngle;
number xPos = p.getPoint(VALUE_X) + (cos(finalAngle) * d.getPoint(VALUE_X));
number yPos = p.getPoint(VALUE_Y) + (sin(finalAngle) * d.getPoint(VALUE_X));
p.setPoint(VALUE_X, xPos);
p.setPoint(VALUE_Y, yPos);
}

return p;
}

Figure 3.7: Avian’s Phase Two pseudocode.

3.4 Altitude Control

Altitude control allows the quadrotor to smoothly takeoff and land. This is done by simply getting the height of the quadrotor in the environment and using that value to synchronously change the motors speeds to move the quadrotor up or down to increase or decrease the height of the quadrotor in the environment. To get the height of the environment we took note of how Gageick et al. [19] created an algorithm for calculating the height over the ground by combining the information from several different main and reference sensors using different weights for each sensor.

We then implemented our own algorithm based off of Gageick’s that uses two sensors that change their weights to assign them as a reference or main sensor depending on their
outputs. We also implemented our own algorithms for landing and takeoff that use the existing attitude controller that the multicopter is using to control the height. This has not been done before as Gageick’s implementation uses its own PID controller to control the motors.

3.4.1 Altitude Control Method

To decide if a sensor should be a main or reference sensor the weights of the sensor’s are calculated, where if the weight of the sensor is the highest it is labeled as the main sensor. For our algorithm we simply state that if the ultrasonic sensor reads a value below a certain value then the weight of that ultrasonic sensor becomes greater as the sensor reads a smaller and smaller value. This is helpful as the quadrotor can start its landing and takeoff phase at any height safely.

The weights for the sensors are variable depending on their value, shown in equations 3.43-3.45. $Alt$ is the altitude of the system, $O_u$ and $O_p$ are the outputs of the ultrasonic and pressure sensors respectively, $W_u$ and $W_p$ are the weights for the output of the ultrasonic and pressure sensors respectively, $M_u$ is the maximum output of the ultrasonic sensor, and $R_u$ is the offset for the ultrasonic sensor, this is to give the ultrasonic sensor greater weight even when the ultrasonic sensor detected a value at a relatively high point. In the simulator the $R_u$ value was set to 1.

\[
Alt = (W_u \cdot O_u) + (W_p \cdot O_p) \tag{3.43}
\]

\[
W_u = \frac{\sqrt{M_u^2 - (O_u + R_u)^2}}{M_u} \tag{3.44}
\]
This helps by getting the current height of the system when it is relevant to the system. In the case of the altitude of the quadrotor the pressure is only relevant when the ultrasonic sensors output is too high so that the quadrotor has an understanding of the relative height in the environment, only when the ultrasonic sensor becomes lower does it become more important as this means the quadrotor is closer to a landing sight.

3.4.1.1 Takeoff Method

The aim of the takeoff method is to have the quadrotor safely lift off and hover at 5 meter until further instructions are given. The takeoff method is done in 3 phases.

1. The quadrotor starts from the bottom of the floor, spooling its motors up and giving a large instantaneous thrust to lift off of the floor by simply setting the desired height of the quadrotor to 10 meters above its current position.
2. The quadrotor decreases its motor output to reach a steady rate of ascent by setting the desired height to only 5 meters above its current position.
3. Once 5 meter is reached the quadrotor hovers at that position until further instructions are given by simply setting the desired position to its current position.
3.4.1.2 Landing Method

Similar to the takeoff, the landing is performed in phases based on the height at which the quadrotor is currently located at. We have improved Gageik’s el al [19] altitude controller by using a buffer time to decrease the rate of descent. This allows for a faster rate of descent at higher altitudes and a much smoother and safer descent at lower altitudes. It also allows the quadrotor to land quickly and safely at any height, compared to only being able to land at a height of less than 60 cm.

Avian’s landing is performed according to the following 4 phases:

1. The quadrotor starts to descend at a steady rate from the starting position until it reaches a meter in height by simply setting the desired height to 4 meters below the current height.

2. The quadrotor slowly starts to reduce the speed of the motors based off of the height, the closer the quadrotor by simply setting the desired height to 2 meters below the current position. This is at 0.5 meters high.

3. The quadrotor quickly decreases the rate of descent to prevent sudden collision, acting as a buffer, and allowing the quadrotor to slowly position itself above the landing zone by setting the desired height to 0.1 meters below the current height of the quadrotor. This is at 3 meters high.

4. The quadrotor turns off all motors to prevent any movement as it securely lands. This is at 0.1 meters.
3.5 Mathematical Flocking Model

In regards to the actual behaviors that the hub backend will dictate to each quadrotor, there are several suggestions. Among these are Reynold’s Flocking Rules [20].

Regardless of the behavior, though, a common need is to be able to discern the center position of the group. This can be achieved by taking the average position of all of the members of the flock within the neighborhood, denoted by:

\[
C_N = \frac{1}{N} \sum_{i=1}^{N} \{x_i, y_i, z_i\}
\]  

(3.46)

where \(C_N\) is the average center of a neighborhood within a flock with \(N\) members in the neighborhood [21].

In order to keep the quadrotors from accelerating too quickly away from each other and their target position, a maximum velocity constant was created to limit each rule from being too large. This keeps the flock together and flying safely by only allowing minor position corrections, preventing over corrections.
3.5.1 Flocking Model 1

Reynold’s flocking rules detail the set of mathematical rules that a flock should follow for optimum group flocking, the first of which is separation. Separation keeps the members of the flock from being too close to one another as seen in figure 3.9.

\[
V_{si}(k + 1)_{x,y,z} = \left[ s \sum_{n \in N} \left( \frac{1}{Dx_{i,n}(k)}, \frac{1}{Dy_{i,n}(k)}, \frac{1}{Dz_{i,n}(k)} \right) \right]^{-1} \tag{3.47}
\]

\[
V_{si}(k + 1)_{x,y,z} \tag{3.48}
\]

where,

\[
Dx_{i,n}(k) = P_{ix}(k) - P_{nx}(k) \]

\[
Dy_{i,n}(k) = P_{iy}(k) - P_{ny}(k) \]

\[
Dz_{i,n}(k) = P_{iz}(k) - P_{nz}(k) \]

For \( P_i(k) \neq P_n(k) \forall n \in N \)

Where Equation 3.48 describes the velocity of the current member of the flock, \( N \) is the number of members counted within the local neighborhood, \( Dx_{i,n}(k) \) is the distance in the \( x \) direction, \( Dy_{i,n}(k) \) is the distance in the \( y \) direction, and \( Dz_{i,n}(k) \) is the distance in the \( z \) direction where \( P_i(k) \) is the position of the current member of the flock and \( P_n(k) \) is the position of a neighbor from the set \( N \) and \( s \) is the separation strength.
The second flocking rule is alignment, which ensures that each member of the flock generally matches the velocities of members near it given by the following Equation:

\[
V_{a_i}(k + 1)_{x,y,z} = \left[ \alpha \times \left( \sum_{n \in N} \frac{V_n(k)}{N} \right) \right]
\]  \hspace{1cm} (3.49)

Where \( V_{a_i}(k + 1) \) is the alignment velocity of the current member of the flock, \( V_n(k) \) is the velocity of the neighboring member of the flock from the set of neighbors \( N \) and \( \alpha \) is the alignment strength.
The third Reynold’s Rule of flocking is cohesion which ensures that each member of the flock steers to move towards the average position of its neighbors (figure 3.11). This is achieved by applying Equation 3.50.

\[
V_{ci}(k + 1)_{x,y,z} = \left[ c \ast \left( \frac{\sum_{n \in N} P_{n}(k)}{N} \right) - P_{i}(k) \right]
\]  \hspace{1cm} (3.50)

where,

\[ P_{i}(k) \rightarrow [P_{ix}(k), P_{iy}(k), P_{iz}(k)] \]

\[ P_{n}(k) \rightarrow [P_{nx}(k), P_{ny}(k), P_{nz}(k)] \]

Noting that Equation 3.50 is the cohesion velocity of the current member of the flock, \( P_{i}(k) \) is the position of the current member of the flock and \( P_{n}(k) \) is the position of a neighbor and \( c \) is the cohesion strength.

\[
V_{ti}(k + 1)_{x,y,z} = [t \ast (P_{T} - P_{i}(k))] \hspace{1cm} (3.51)
\]

\[ P_{T} \rightarrow [P_{Tx}(k), P_{Ty}(k), P_{Tz}(k)] \]  \hspace{1cm} (3.52)
The fourth flocking behavior is in place to steer each member of the flock to a specific location in space which helps to keep the flock moving collectively towards a given target. This is achieved by creating a target velocity parameter (Equation 3.51) where $V_{ti}(k+1)_{x,y,z}$ is the targeting velocity of the current member of the flock, Equation 3.52 is the position of the target and $t$ is the targeting strength.

![Figure 3.13: Obstacle Avoidance](image)

The fifth and final rule outlined by Reynold’s rules of flocking is avoidance. This rule forces each member of the flock to avoid a particular point in space. This is accomplished by applying Equation 3.53 where $G$ is the obstacle avoidance function and $sgn$ is the signum function.

$$V_{ri}(k+1)_{x,y,z} = [r \ast G \ast (sgn(P_i(k)) - sgn(P_R))]$$  \hspace{1cm} (3.53)

where,

$$G = \left\{ \frac{1}{\sqrt{(P_{ix}(k) - P_{Rx})^2 + (P_{iy}(k) - P_{Ry})^2 + (P_{iz}(k) - P_{Rz})^2}} \right\}$$

$$P_R \rightarrow [P_{Rx}, P_{Ry}, P_{Rz}]$$
\[ sgn(P_i, R) \rightarrow [sgn(P_x), sgn(P_y), sgn(P_z)]_{i,R} \quad (3.54) \]

where,

\[
sgn(x) = \begin{cases} 
-1 & \text{for } x < 0 \\
0 & \text{for } x = 0 \\
1 & \text{for } x > 0.
\end{cases}
\]

Equation 3.53 is the avoidance velocity of the current member of the flock, \( P_R \) is the position of the obstacle to be avoided (the point in space that will be globally repulsive to each member of the flock), and \( r \) is the avoiding strength.

With all of the velocities calculated from each of the five rules, Oweis et al. [8] calculates the flock member’s new velocity parameter with the following:

\[
V = [V_{si}(k + 1)_{x,y,z} + V_{ai}(k + 1)_{x,y,z} + V_{ci}(k + 1)_{x,y,z} \\
+ V_{li}(k + 1)_{x,y,z} + V_{ri}(k + 1)_{x,y,z}] \quad (3.55)
\]

Where the resultant velocity \( V \) cannot exceed some constant \( V_{limit} \). Using the new velocity, the next position of the flock member can be calculated using the position formula, Equation 3.56 noting that \( P_i(k) \) is the current position of the flock member.

\[
P_i(k + 1) = [P_i(k) + V_i(k + 1)] \quad (3.56)
\]

Another useful parameter that we can calculate using the velocity is the acceleration. This is accomplished using the acceleration formula, given by Equation 3.57 [8].

\[
A_i(k + 1) = [V - V_i(k + 1)] \quad (3.57)
\]
\( A_i(k + 1) \rightarrow [A_{ix}, A_{iy}, A_{iz}] \)  

(3.58)
4. Simulation

We developed a concurrent simulator in Java to test our proposed system. The simulator has two main components, the multicopter component and the hub component.

The multicopter component operates by placing Multicopter objects into a virtual environment that is simulated in the Environment class with many natural forces and aerodynamic effects. Each multicopter is designed to utilize an attitude, altitude, and collision avoidance technique. Each object in the virtual environment is first classified by its type (Movable vs Immovable vs Force objects) that have relevant properties. For instance a Movable object must keep track of Velocity, Acceleration, and Inertia, a Force object must keep track of the changing or unchanging forces that it needs to apply, and an Immovable object doesn’t need that information for the purposes of the simulation.

The hub component works through several key classes. First and foremost is the Hub object, which simulates a base station. It starts up when the application is opened and operates by accepting multicopter connections. Each connection is passed to a HiveMember object, which is the class that keeps track of all important details of the simulation such as the current position, velocity, and acceleration of that specific member. The HiveMember object is also responsible for keeping track of as well as ultimately applying each of the behaviors mentioned in section 4.13 and some others as can be seen in the behavior class diagram in figure 4.19. Once the HiveMember is constructed, it is assigned to the next available HiveInstance. Each HiveInstance can accommodate a set number of HiveMembers, defined
by a constant $Member\ limit$, once a HiveInstance reaches that number of members the Hub launches another instance to handle the next set of members.

4.1 Distributing the Simulation

One of the main features of the simulation is the fact that it is distributed by having the multicopter component and the hub component able to be run on two different networked machines. Please refer to the multicopter architecture in Figure 4.6 and the hub architecture in Figure 4.18.

For the simulation, there are two main methods that can be invoked. The primary application entry point is found in the main controller class and it runs the user interface for interacting with the simulation. The second entry point is found in the hive controller and is used to start the hub and all of the cloud instances that run concurrently on it.

The benefit of running each component on its own machine is that it allows for the large number of concurrent threads in each component to run without having to slow down the simulation. Take, for instance, a multicopter’s array of ultrasonic sensors. In practice, these run by sending out pings to the environment in regular intervals (ping rate) and the multicopter checks against them at its leisure. In the simulation this is done by running each of the sensors as their own thread so that even in the simulation the multicopter is always able to get up to date information about its surroundings without having to pause or slow down other parts of the simulation. This ultimately results in a much more fluid simulation execution.
4.2 Environment

![Environmental Class Diagram](image)

The environment contains a collection of objects, all of which are either of type movable, immovable, or force. These objects all extend the environmental object class that gives the objects a shape, linear position, angular position, name, and color. The environment is set in a three dimensional Euclidian space where the coordinate system is cartesian, split into $x$, $y$, and $z$ standard. The $x$ direction states the length, the $y$ direction states the width, and the $z$ direction states the height.
The purpose of the immovable object type is to act as an obstacle in the environment. There is not a single difference between the Environmental Object and the Immovable object.

The purpose of the movable object type is that they can move freely inside the entire environment, that means that the object must now follow Newtonian mechanics, meaning that the movable object must now be concerned with a system of forces acting on it. The movable object contains more than just a position in a 3 dimensional space, it also contains the linear and angular velocities and accelerations. The movable object has a set of all of the forces that are acting on it, torques that are acting on the system, mass of the movable object, and an inertia that follow that shape of the movable object. These attributes allow the environment to calculate out the changes in the linear and angular positions, velocities, and accelerations allow for a fully movable object.

The force object is a system of forces as it simulates the forces of wind and other movement. The force object is there to act as a disturbance to the movable object and can add to the set of forces as well as torques of the movable object. The force object only affects the movable object when the movable object is inside of the force object. When this happens the environment adds the force of the force object on to the system of forces of the movable object. If the movable object is only halfway inside of the force object then, just as if wind was blowing on half of a quadrotor, there would be a torque on the system such that the force of the force object is affecting the movable object in Equation 4.1 where \( \mathbf{f} \) is the force from the force object, \( r \) is the radius from the center of the acting force, and \( \theta \) is the angle at which the force is acting on the surface of the movable object.
To calculate the torques acting on the movable object, a method was created to check and see if the movable object was inside of a force object. The method goes as follows:

1. Define the center of the movable object as the center of the Euclidean space.

2. Get the maximum and minimum of all 3 dimensions of the movable object.
   The maximum of dimensions $x$, $y$, and $z$ are shown in Equation 4.2. The minimum of dimensions $x$, $y$, and $z$ are shown in Equation 4.3 respectively.

3. Get the maximum and minimum of all 3 dimensions of the force object. The maximum of dimensions $x$, $y$, and $z$ are shown in Equations 4.4. The minimum of dimensions $x$, $y$, and $z$ are shown in Equations 4.5.

4. Compare the maximums and minimums of the 3 dimensions of the movable and force object.
   a. If the maximum position of the movable object is less than the maximum position of the force object, and the minimum position of the movable object is greater than the minimum of the force object then the movable object is inside of the force object, meaning that there are no torques on the movable object.
   b. If the maximum position of the force object is greater than the maximum position of the movable object, and the maximum position of the movable object is greater than the minimum position of the force object then the force is acting on the positive side of the movable object where the exact position of the acting force is shown in Equation 4.6.
c. If the minimum position of the movable object is greater than the minimum position of the force object, and the maximum position of the force object is greater than the minimum position of the movable object then the force is acting on the negative side of the movable object where the exact position of the acting force is shown in Equation 4.7.

5. Calculate the Torque from the force using the calculated distance and the angle of the movable object in that dimension.

\[
Max_{\text{movable}} = \left[ x_{\text{movable}} + \frac{\text{length}_{\text{movable}}}{2} \sin(\phi) \right] \\
y_{\text{movable}} + \frac{\text{width}_{\text{movable}}}{2} \sin(\theta) \\
z_{\text{movable}} + \frac{\text{height}_{\text{movable}}}{2} \sin(\psi) \tag{4.2}
\]

\[
Min_{\text{movable}} = \left[ x_{\text{movable}} - \frac{\text{length}_{\text{movable}}}{2} \sin(\phi) \right] \\
y_{\text{movable}} - \frac{\text{width}_{\text{movable}}}{2} \sin(\theta) \\
z_{\text{movable}} - \frac{\text{height}_{\text{movable}}}{2} \sin(\psi) \tag{4.3}
\]

\[
Max_{\text{force}} = \left[ x_{\text{force}} + \frac{\text{length}_{\text{force}}}{2} \sin(\phi) \right] \\
y_{\text{force}} + \frac{\text{width}_{\text{force}}}{2} \sin(\theta) \\
z_{\text{force}} + \frac{\text{height}_{\text{force}}}{2} \sin(\psi) \tag{4.4}
\]

\[
Min_{\text{force}} = \left[ x_{\text{force}} - \frac{\text{length}_{\text{force}}}{2} \sin(\phi) \right] \\
y_{\text{force}} - \frac{\text{width}_{\text{force}}}{2} \sin(\theta) \\
z_{\text{force}} - \frac{\text{height}_{\text{force}}}{2} \sin(\psi) \tag{4.5}
\]

\[
p = \frac{s}{2} - (Max_{\text{movable}} - Min_{\text{force}}) \tag{4.6}
\]

\[
p = -\frac{s}{2} - (Max_{\text{force}} - Min_{\text{movable}}) \tag{4.7}
\]

\[
s = \left[ \begin{array}{c} \text{length}_{\text{movable}} \\ \text{width}_{\text{movable}} \\ \text{height}_{\text{movable}} \end{array} \right] \tag{4.8}
\]
Once the forces and torques are calculated and added to the system of forces and torques acting on the movable object, the environment then calculates the linear and angular positions, velocities, and accelerations from the forces and torques.

4.3 Multicopter

![Multicopter Class Diagram]

Figure 4.6: Multicopter Class Diagram
The multicopter is a class to represent an autonomous quadrotor that receives inputs from the cloud through the Network Socket class as to which position it needs to go all while taking into account its own surroundings and discerning where it can and cannot go. This is where the quadrotor takes action based off of the attitude control, altitude control, and the obstacle avoidance methods are decided upon runtime. The class diagram for the multicopter is shown in figure 4.6.

4.3.1 Multicopter Cycle

At the beginning of each cycle the Multicopter follows a sequence of steps to control its aerial movement autonomously. The sequence of steps is as follows:

1. Update the calculated position of the multicopter based off of the values from the Inertial Measurement Unit’s accelerometer and gyroscope.
2. If currently in an process of takeoff or landing update the desired position based off of the requirements set by the takeoff and landing procedure.
3. Update the commanded position, velocity, and acceleration based off of the desired position, velocity, and acceleration. The desired position, velocity, and acceleration are usually given by the cloud through the network socket or by the altitude controller.
4. Recalculate the commanded position based off of the collision detection method.
5. Set the suggested motor speeds using the attitude controller.
6. Send the suggested motor speeds to the motor controller.

The Environment class then updates the multicopter based off the the motor controllers motor speeds using the Multicopter System Block that contains the Mathematical model of the quadrotor.

### 4.3.2 Mathematical Constants

The Multicopter constants for the mathematical model discussed in section 3.1 are based off the propeller and body structure of the DJI FlameWheel F330, a commercially available quadrotor design. The constants for the quadrotor model are shown in table 4.1.

<table>
<thead>
<tr>
<th>Multicopter Constants</th>
<th>Symbols</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solidity Ratio</td>
<td>$\sigma$</td>
<td>0.008839</td>
</tr>
<tr>
<td>Lift Slope</td>
<td>$a$</td>
<td>4.5</td>
</tr>
<tr>
<td>Air Density</td>
<td>$\rho$</td>
<td>1.1843</td>
</tr>
<tr>
<td>Twist Pitch</td>
<td>$\theta_{tu}$</td>
<td>4.45</td>
</tr>
<tr>
<td>Mass</td>
<td>$m$</td>
<td>0.5</td>
</tr>
<tr>
<td>Horizontal Motor Length from Center of Gravity</td>
<td>$l$</td>
<td>1.8</td>
</tr>
</tbody>
</table>
Table 4.1: Mathematical Multicopter Constants

| Vertical Motor Length from Center of Gravity | $h$   | 0.4 |

4.4 Sensors

The sensors are there to give the multicopter a reference to the environment and its current surroundings, the sensors themselves run on their own thread and are controlled via the sensor controller, giving the multicopter a constant feed on its location and surroundings.

Just like the multicopter each sensor has its respective cycle and noise level. The cycle rate simply defines when the sensor is to update its current perceived value detected based off its position in its environment and the surrounding objects. The cycle rate and noise level is defined by the sensor, mimicking a real life example of that sensor.

4.4.1 Pressure Sensor

The pressure sensor used gives information on the current atmospheric pressure depending on the location of the quadrotor in the environment. The equation to solve for the atmospheric pressure detected via the sensor that the height of the positioned sensor we use the barometric formula shown in equation 4.8, where $P_h$ is the atmospheric pressure at the height of the pressure sensor relative to the environment, $P_o$ is the atmospheric pressure at at sea level, $m$ is the mass of one molecule, $g$ is the acceleration of gravity, $h$ is the height of the
pressure sensor relative to the environment, $L$ is the temperature lapse rate, $R$ is the universal
gas constant, and $T$ is the average temperature at sea level.

$$P_h = P_o \cdot \left(1 - \frac{L h}{T} \right)^{\frac{R m}{R T}} \quad (4.8)$$

Once we have the pressure from the pressure sensor we then need to calculate the
height of the pressure sensor from the current perceived pressure, shown in equation 4.9.

$$h = \frac{T \cdot \left(\frac{P_h}{P_o}\right)^{\frac{1}{R m}}}{L} \quad (4.9)$$

The Pressure Sensor was designed around the BMP180 digital pressure sensor, which
has a cycle rate of one second, and a maximum noise level of 0.06. [37]

4.4.2 Ultrasonic Sensor

As mentioned in section 4.1, the ultrasonic sensor emits a ping and calculates the
distance by the amount of time it takes for that ping to return. To simulate an ultrasonic
sensor, we specify the range of the ping. So say that the range of the ping is specified to be
equal to 30 degrees, and the position of the ultrasonic sensor is placed at a 45 degree angle on
the quadrotor from the front of the quadrotor then the full range of the ping in the unit circle is
from 30 degrees to 60 degrees from the front of the quadrotor where 30 degrees is the min and 60 degrees is the max.

In our simulator, first we position the ultrasonic sensor relative to the quadrotor. To do so, we solve for the $\psi$ angle of the ultrasonic sensor relative to the quadrotor using the total amount of ultrasonic sensors (count) and the index of the ultrasonic sensor (i) as shown in Equation 4.10. This also determines the angle that the ultrasonic sensor is facing towards its environment so that it detects objects.

$$\psi = \left( \left( \frac{2\pi}{\text{count}} \right) i \right) + \left( \frac{2\pi}{\text{count} \cdot 2} \right)$$  \hspace{1cm} (4.10)

From the $\psi$ value we then get the $x$ and $y$ positions of the ultrasonic sensor relative to the quadrotors body as shown in Equations 4.11 and 4.12, where $l$ is the horizontal length of the position of the ultrasonic sensor relative to the center of the quadrotor. Figure 4.7 shows how the ultrasonic sensors would be displayed if we had 8 ultrasonic sensors on 1 quadrotor.

Figure 4.7: The Ultrasonic Sensors
\[ x = l \cdot \sin(\psi) \]  \hspace{1cm} (4.11)

\[ y = l \cdot \cos(\psi) \]  \hspace{1cm} (4.12)

Once we have the initial position of the ultrasonic sensor we then decide how to create the rays from the initial position of the Ultrasonic sensor. To do this we first decide how to vertically and horizontally create rays that fill in the range of the ultrasonic sensor, as shown in figure 4.8. For ray creation we have 2 steps, the creation of the endpoint of the ray shown in figure 4.9, as the origin of the ray is always in the center of the ultrasonic sensor, and the rotation of the endpoint relative to the ultrasonic sensors position to the environment as shown in figure 4.10.

```java
number start = - PI - (ANGLE / 2.0);
number increment = ANGLE / RAYCAST_COUNT;
if (increment == Double.NaN || increment == Double.POSITIVE_INFINITY)
{
    increment = 0.0;
}
for (i = 0; i <= RAYCAST_COUNT; i++)
{
    number lattIncrement = start + (i * increment);
    for (j = 0; j <= RAYCAST_COUNT; j++)
    {
        number heightIncrement = - (ANGLE / 2.0) + (j * increment);
        endPointList.add(rotateEndPoint(createEndPoint(lattIncrement,
            heightIncrement,
            ultrasonicAngularPosition,
            ultrasonicLinearPosition));
    }
}
```

Figure 4.8: Setting all vertical and horizontal rays for the Ultrasonic sensor.
number[] createEndPoint(number psi, number epsilon)
{
    number endPoint[];
    endPoint[VALUE_X] = (DETECT_DIST + DISTANCE_FROM_CENTER) * sin(psi);
    endPoint[VALUE_Y] = (DETECT_DIST + DISTANCE_FROM_CENTER) * cos(psi);
    endPoint[VALUE_Z] = (DETECT_DIST + DISTANCE_FROM_CENTER) * sin(epsilon);
    return endPoint;
}

Figure 4.9: Creating an end point.

number[] rotateEndPoint(number endPoint[], Position angularPos, Position multiPos)
{
    // rotate around the z axis
    number xTemp = endPoint[VALUE_X] * cos(angularPos.getPoint(VALUE_PSI))
    - endPoint[VALUE_Y] * sin(angularPos.getPoint(VALUE_PSI));
    number yTemp = endPoint[VALUE_X] * sin(angularPos.getPoint(VALUE_PSI))
    + endPoint[VALUE_Y] * cos(angularPos.getPoint(VALUE_PSI));
    number zTemp = endPoint[VALUE_Z];
    endPoint[VALUE_X] = xTemp;
    endPoint[VALUE_Y] = yTemp;
    endPoint[VALUE_Z] = zTemp;

    // rotate around the y axis
    xTemp = endPoint[VALUE_X] * cos(-angularPos.getPoint(VALUE_THETA))
    + endPoint[VALUE_Z] * sin(-angularPos.getPoint(VALUE_THETA));
    yTemp = endPoint[VALUE_Y];
    zTemp = -endPoint[VALUE_X] * sin(-angularPos.getPoint(VALUE_THETA))
    + endPoint[VALUE_Z] * cos(-angularPos.getPoint(VALUE_THETA));
    endPoint[VALUE_X] = xTemp;
    endPoint[VALUE_Y] = yTemp;
    endPoint[VALUE_Z] = zTemp;

    // rotate around the x axis
    xTemp = endPoint[VALUE_X];
    yTemp = endPoint[VALUE_Y] * cos(-angularPos.getPoint(VALUE_PHI))
    - endPoint[VALUE_Z] * sin(-angularPos.getPoint(VALUE_PHI));
    zTemp = endPoint[VALUE_Y] * sin(-angularPos.getPoint(VALUE_PHI))
    + endPoint[VALUE_Z] * cos(-angularPos.getPoint(VALUE_PHI));
    endPoint[VALUE_X] = xTemp + multicopterPos.getPoint(VALUE_X);
    endPoint[VALUE_Y] = yTemp + multicopterPos.getPoint(VALUE_Y);
    endPoint[VALUE_Z] = zTemp + multicopterPos.getPoint(VALUE_Z);
Figure 4.10: Rotating the end point.

The final result is shown in figure 4.11 showing all of the rays created by the methods above.

Figure 4.11: The rays of the Ultrasonic sensor.

We use the ray casting method proposed by Toms Möller et. al [33] to calculate the simulated distance perceived via the ultrasonic sensor.

The Ultrasonic Sensor was designed around the Navo Ultrasonic Sensor, that did not have a datasheet containing the cycle rate and the maximum noise level. [29] In this case we assumed that the cycle rate is 500 milliseconds and the maximum noise level is 0.05. The
maximum distance the ultrasonic sensor can detect is 4.5 meters, the minimum distance is 0.2 centimeters.

4.4.3 Inertial Measurement Unit

The Inertial Measurement Unit, IMU, is a measuring unit that houses a 3 axis gyroscope, accelerometer, and magnetometer. In our case we only care about the gyroscope and accelerometer.

The Sensor is designed after the MinIMU-9 v3 where the cycle rate is 1 millisecond and the maximum noise level is 0.011. [28]

4.4.3.1 Gyroscope

The gyroscope sensor is there to detect the angular acceleration that the sensor is currently undergoing. For the simulation the gyroscope was designed to first get the rate of change in angular position from its current angular position to get the angular velocity. To calculate the angular acceleration the rate of change in angular velocity is calculated. The angular acceleration that the gyroscope receives is shown in equation 4.13 and shows the
perceived angular acceleration, where $\omega_0$ is the initial angular velocity, $\Omega$ is the angular acceleration, $t$ is the refresh rate, in our case this was equal to 0.030 - 0.035, and $\lambda_0$ was the initial angular position relative to its body in the environment. The amount of time that has passed is the sensors refresh rate.

$$\Omega = \begin{bmatrix} \frac{\lambda_\phi - \lambda_{0\phi}}{t} - w_{0\phi} \\ \frac{\lambda_\theta - \lambda_{0\theta}}{t} - w_{0\theta} \\ \frac{\lambda_\psi - \lambda_{0\psi}}{t} - w_{0\psi} \end{bmatrix}$$  \hspace{1cm} (4.13)

4.4.3.2 Accelerometer

The accelerometer is there to detect the linear acceleration relative to its body position. To calculate its linear acceleration the we first calculated the change in position and then the change in velocity giving us the perceived linear acceleration relative to the accelerometer shown in equation 4.14, where $v_0$ is the initial linear velocity, $a$ is the linear acceleration, $t$ is the refresh rate, in our case this was equal to 0.030 - 0.035, and $P_0$ was the initial linear position in the environment. Just like the gyroscope, the accelerometers time that has passed is its refresh rate.

$$a = \begin{bmatrix} \frac{p_x - p_{0x}}{t} - v_{0x} \\ \frac{p_y - p_{0y}}{t} - v_{0y} \\ \frac{p_z - p_{0z}}{t} - v_{0z} \end{bmatrix}$$  \hspace{1cm} (4.14)
Once we have the linear acceleration of the accelerometer in its environment calculate the linear acceleration relative to its body frame shown in equation 4.15. This is the perceived acceleration of the accelerometer, where $\phi$, $\theta$, and $\psi$ are the angular positions of the accelerometer relative to the environment.

$$a_p = \begin{bmatrix} a_x \cos(\phi) + a_y \sin(\phi) + a_z \sin(\phi) \\ a_y \cos(\theta) + a_x \sin(\theta) + a_z \sin(\theta) \\ a_z \cos(\psi) + a_x \sin(\psi) + a_y \sin(\psi) \end{bmatrix}$$  (4.15)

### 4.4.5 White Noise

Each and every sensor generates some level of white noise. In our simulation we simply recreate the white by randomly checking if the sensor should have white noise, randomly deciding if the sensor should have white noise be added to or subtracted from the current sensor value, and then creating a random value between zero and the maximum noise value of the sensor. The pseudocode for this is shown in figure 4.12.

```java
if (floor((random() * WHITE_NOISE_RANDOM)) == WHITE_NOISE_RANDOM_EQUAL)
{
    if ((floor((random() * 2) % 2) == 1))
    {
        sensorValue = sensorValue + (random() * MAXIMUM_NOISE_LEVEL);
    }
    else
    {
        sensorValue = sensorValue - (random() * MAXIMUM_NOISE_LEVEL);
    }
}
```

*Figure 4.12: Pseudocode for generating white noise for sensor value.*
In the case of the simulation WHITE_NOISE_RANDOM was set to a value of 3, WHITE_NOISE_RANDOM_EQUAL was set to a value of 1, and MAXIMUM_NOISE_LEVEL was the maximum noise level of the sensor.

### 4.5 Relative Positioning

In the simulation relative positioning tells the quadrotor what linear and angular position its at in the environment. This is done by using the output of the perceived angular acceleration from the gyroscope, and the perceived linear accelerations from the accelerometer. The accelerometer and gyroscope are both placed inside of the IMU which is positioned right above the multicopters body shown in figure 4.13.

![Figure 4.13: Multicopter with an IMU](image)

The relative angular position is first calculated via getting the angular velocity using the kinematic equation shown in equation 4.16 where $w_{0\phi}$, $w_{0\theta}$, and $w_{0\psi}$ are the previous relative angular velocities in the phi theta and psi directions, $\Omega_{0\phi}$, $\Omega_{0\theta}$, and $\Omega_{0\psi}$ are the undergone angular accelerations, and $t$ the amount of time that has passed since the last velocity was calculated. Relative angular position is then calculated using the kinematic equation shown in equation 4.17 where $\lambda_{0\phi}$, $\lambda_{0\theta}$, and $\lambda_{0\psi}$ are the previous relative angular positions in the phi, theta, and psi directions.
The relative linear position is calculated by converting the perceived acceleration from the accelerometer to the relative acceleration in the environment shown in equations 4.18-4.20, where \( \dot{\phi}, \theta, \) and \( \psi \) are the relative angular positions calculated in equation 4.17, and \( a_{px}, a_{py}, \) and \( a_{pz} \) are the perceived values from the accelerometer in the x, y and z direction respectively.

\[
\lambda = \begin{bmatrix} 
\lambda_{0\dot{\phi}} + w_{\dot{\phi}} \cdot t + \frac{(\Omega_{\phi})^2}{2} \\
\lambda_{0\theta} + w_{\theta} \cdot t + \frac{(\Omega_{\theta})^2}{2} \\
\lambda_{0\psi} + w_{\psi} \cdot t + \frac{(\Omega_{\psi})^2}{2} 
\end{bmatrix}
\]  
(4.17)

\[
w = \begin{bmatrix} 
w_{0\dot{\phi}} + \Omega_{\phi} \cdot t \\
w_{0\theta} + \Omega_{\theta} \cdot t \\
w_{0\psi} + \Omega_{\psi} \cdot t 
\end{bmatrix}
\]  
(4.16)

\[
a_{dz} = \frac{a_{px} \cos(\psi) - a_{pz} \tan(\psi)}{\cos(\phi)} + \frac{a_{py} \tan(\psi) \left( \tan(\phi) \left( \tan(\phi) - 1 \right) \right)}{\cos(\phi) \left( 1 - \tan(\phi) \tan(\theta) \right)} - \frac{a_{pz} \tan(\theta) \left( \tan(\psi) \left( \tan(\phi) - 1 \right) \right)}{\cos(\phi) \left( 1 - \tan(\phi) \tan(\theta) \right)}
\]

\[
a_{dy}(\theta) = \frac{a_{py} \tan(\theta)}{\cos(\phi)} - a_{dz} \tan(\theta) \left( \tan(\phi) - 1 \right)
\]

\[
a_{dx}(\phi) = \frac{a_{px}}{\cos(\phi)} - a_{dy} \sin(\phi) - a_{dz} \sin(\phi)
\]

(4.18)  
(4.19)  
(4.20)
From this we calculate then calculate the relative linear velocity using the kinematic equation shown in equation 4.21 where $v_{0x}$, $v_{0y}$, and $v_{0z}$ are the previous relative linear velocities in the x, y, and z direction. The relative positioning is then calculated using the kinematic equation shown in equation 4.22 where $p_{0x}$, $p_{0y}$, and $p_{0z}$ are the previous relative positions.

$$ v = \begin{bmatrix} v_{0x} + a_{dx} \cdot t \\ v_{0y} + a_{dy} \cdot t \\ v_{0z} + a_{dz} \cdot t \end{bmatrix} $$

(4.21)

$$ p = \begin{bmatrix} p_{0x} + v_x \cdot t + \frac{(a_{dx})^2}{2} \\ p_{0y} + v_y \cdot t + \frac{(a_{dy})^2}{2} \\ p_{0z} + v_z \cdot t + \frac{(a_{dz})^2}{2} \end{bmatrix} $$

(4.22)

From this we have the relative positions of the quadrotor in its environment based solely on the perceived accelerations from the gyroscope and accelerometer.

4.6 Attitude Controller

The attitude controller of the quadrotor is there to allow the quadrotor to control its angular position to tilt from one point in the environment to another as discussed in section 3.2.

In the simulation we implemented the attitude controllers as discussed in their respective sections. After implementation the attitude controllers were tuned to the environment specifically. This entailed changing the constants set in each controller until the
attitude controller did not over or under react. The Dikmen Sliding mode controller was not tuned, the constants were acquired from Dikmen’s implementation. [17]

### 4.6.1 Attitude Controller Method

For the attitude controller we designed a basic algorithm to move the quadrotor from one position to another based off of the commanded linear position given. The design was based off of the suggested attitude control method given by Kemper in section 3.2.1. The attitude controller method occurs in 6 steps:

1. Calculate the necessary linear acceleration to drive the system to its desired state. Check that the linear accelerations are not outside of the maximum and minimum range.
2. Calculate the necessary angular position to achieve that desired linear acceleration.
3. Calculate the necessary angular velocity to achieve the desired angular position.
4. Calculate the commanded torques about the three axes of the quadrotor given the linear acceleration, angular position and velocity.
5. Calculate the necessary motor speeds to operate the angular position.
6. Give the quadrotor the motor speeds as suggested motor speeds for the quadrotor to then send to the Motor controller class.
This method opened the simulator to easily swap out different attitude controllers. When calculating the necessary linear accelerations each attitude controller has their own maximum and minimum accelerations shown in table 4.2.

<table>
<thead>
<tr>
<th>Acceleration Limit</th>
<th>Kemper</th>
<th>Hoffman</th>
<th>Dikmen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum X and Y</td>
<td>50</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>Maximum Z</td>
<td>70</td>
<td>50</td>
<td>90</td>
</tr>
<tr>
<td>Minimum Z</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.2: Maximum and Minimum Linear Accelerations for Attitude Controllers.

### 4.6.2 Tuning PID Controller

For proper flight reactions the PID constants need to be tuned. Tuning a PID controller is a three step process, first to set up the tuning method we must set the proportional, integral, and derivative constants to zero. Then set a desired position for the PID controller to move towards. This gives us a reaction point in which we see how the quadrotor positions itself around the reaction point.
1. Slowly increase the proportional constant until the quadrotor starts to quickly oscillate around the reaction point. Set the proportional constant to half its value and move to setting up the derivative.

2. Slowly increase the derivative constant till the quadrotor oscillates around the reaction point. Set the derivative constant to half of its value and start setting up the integral constant.

3. Slowly increase the integral constant till the quadrotor oscillates around the reaction point, and then halve the current value. This tune-up should be adequate.

The constants for the PID / PD controller for the necessary linear acceleration are shown in table 4.3. The constants for the PD controllers used in both the Hoffman and Kemper angular position calculations are shown in table 4.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Kemper</th>
<th>Hoffman</th>
<th>Dikmen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportional X</td>
<td>7.3</td>
<td>8.5</td>
<td>10.3</td>
</tr>
<tr>
<td>Proportional Y</td>
<td>7.3</td>
<td>8.5</td>
<td>10.3</td>
</tr>
<tr>
<td>Proportional Z</td>
<td>10.0</td>
<td>11.2</td>
<td>10.0</td>
</tr>
<tr>
<td>Derivative X</td>
<td>25.0</td>
<td>22.0</td>
<td>25.0</td>
</tr>
<tr>
<td>Derivative Y</td>
<td>25.0</td>
<td>22.0</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Table 4.3: PID / PD constants for calculating the necessary linear acceleration.

<table>
<thead>
<tr>
<th>Derivative Z</th>
<th>30.0</th>
<th>27.0</th>
<th>20.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral X</td>
<td>0.5</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>Integral Y</td>
<td>0.5</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>Integral Z</td>
<td>0.3</td>
<td>0.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 4.4: PD constants for angular positions

<table>
<thead>
<tr>
<th>Constant</th>
<th>Kemper</th>
<th>Hoffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportional Phi</td>
<td>15.0</td>
<td>13.0</td>
</tr>
<tr>
<td>Proportional Theta</td>
<td>15.0</td>
<td>13.0</td>
</tr>
<tr>
<td>Proportional Psi</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Derivative Phi</td>
<td>50.0</td>
<td>45.0</td>
</tr>
<tr>
<td>Derivative Theta</td>
<td>50.0</td>
<td>45.0</td>
</tr>
<tr>
<td>Derivative Psi</td>
<td>5.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Table 4.4: PD constants for angular positions

4.6.3 Calculating Motor Speeds
To calculate the necessary motor speeds we used the same equations used to calculate the motor speeds in Kempers attitude control section 3.2.1. The constants for calculating the motor speeds are shown in table 4.5.

<table>
<thead>
<tr>
<th>Constants</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag Coefficient</td>
<td>( b )</td>
<td>0.15</td>
</tr>
<tr>
<td>Thrust Coefficient</td>
<td>( k_c )</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 4.5: Motor speed calculation constants

4.7 Motor Control

In the case of the multicopter we designed the motor controller to simply get the suggested values from the attitude controller and check them against the maximum and minimum values of what the motors can run at. In the case of the simulation we set the minimum value of the motors to run at 150 rpm, and the maximum at 3000 rpm.

4.8 Obstacle Detection

For the obstacle detection we implemented both Gageick and Wagster methods discussed in section 3.3. In the simulation both obstacle detection methods used the same
setup, 8 ultrasonic sensors positioned around the multicopter body. The pseudocode for the positioning of the ultrasonic sensors evenly around the multicopter is shown in figure 4.14.

```java
for (number i = 0; i < SENSOR_COUNT; i++)
{
    // Initialize each UltrasonicSensor and start its thread.
    Position angularPos = new Position();
    Position linearPos = new Position();
    angularPos.setPoint(VALUE_PSI,
        ((TOTAL_DEGREES/(SENSOR_COUNT)) * i)
        + ((TOTAL_DEGREES)/(SENSOR_COUNT) * 2.0));
    linearPos.setPoint(VALUE_Y,
        DISTANCE_FROM_CENTER * cos(angularPos.getPoint(VALUE_PSI)));
    linearPos.setPoint(VALUE_X,
        DISTANCE_FROM_CENTER * sin(angularPos.getPoint(VALUE_PSI)));
    UltrasonicSensor us = new UltrasonicSensor(angularPos, linearPos);
    // Add it to our array of sensors.
    myUltrasonicSensorList.add(us)
}
```

Figure 4.14: Pseudocode for placing the Ultrasonic Sensors around the Multicopter.

In the case of the simulation DISTANCE_FROM_CENTER is the distance of the ultrasonic sensor from the center of the multicopter, which is equal to the horizontal motor length from the center of gravity defined in section 4.3.2. TOTAL_DEGREES is the total amount of degrees in a circle which is 360, and SENSOR_COUNT is just the amount of ultrasonic sensors placed around the multicopter which is set to 8 shown in figure 4.15.
4.9 Altitude Controller

In our simulation we implement the takeoff and landing methods discussed in section 3.4. For the sensors we used a pressure sensor positioned five centimeters below the multicopters body and placed an ultrasonic sensor 15 centimeters below the quadrotors body facing the ground as shown in figure 4.16.
4.10 Communication

The network layer of the simulation is done using the Java networking packages and classes to simplify network operations between the multicopter objects and the hub. Each object that needs networking support simply needs a NetworkSocket property which can be used to accept or make connections (represented in the simulation as the NetworkConnection object). This allows each network object to then communicate as if it was its own network entity, though in reality each socket opens a port on the local machine where that fragment of the simulation is being run. Each group of network objects is assigned a starting port and is then incremented from there.

For example, as there are multiple multicopter objects that are connected to the network, the first one starts at the multicopter base port and each subsequent multicopter will then operate on the next available port as determined by the multicopter’s ID (which is
determined upon object generation). So if the multicopter base port is 4000 then the first multicopter’s port will be 4000 + 0 as multicopter ID’s are zero based indexes. This same structure is used to determine the port that each network-enabled object runs on.

Therefore, the port of any given network-enabled object can be denoted as 
\[ \text{port} = \text{base} + i \]
where \text{base} is the base port for that network object and \( i \) is the zero-based ID of that network object. This way the simulation can show the effects of networking operations and latency on how the system of multicopters will perform in a real environment while minimizing on the hardware requirements of our simulation.

For actual data transmission to take place, a Packet object was created so it could be extended by other, more specialized types of data. This can be seen in the class diagram shown in Figure 4.17 below. Among the four Packet subclasses are ConnectionPacket, UpdatePacket, BehaviorPacket, and StatusPacket. The ConnectionPacket object is the first packet sent by a peer, specifically in our case the multicopter connecting to the hub, which transmits the name of the peer (for identification and logging purposes) as well as a Position object describing its current position. The UpdatePacket contains the current Position, Acceleration, and Velocity of the multicopter. Alternatively, it can also be used by the cloud or multicopter to request an update from the other. The BehaviorPacket allows the multicopter to modify its current behavior state in the hub with batch operations. By populating a list of behavior identification numbers and communicating it to the hub a multicopter can alter its own behavior state. Finally, the StatusPacket just contains an enumerated value that describes
the ready status of the multicopter such as READY or PAUSE which notifies the hub as to whether or not to update that multicopter.

![Network Layer Class Diagram](image)

**Figure 4.17: Network Layer Class Diagram**

To read and write fully constructed Java objects to the network, object streams were used as opposed to traditional primitive data streams to reduce the complexity of the network layer. In order to correctly utilize this special type of data stream, special care had to taken in implementation. This was done by first ensuring that no object caching took place, otherwise the object stream will attempt to optimize its speeds by caching objects that had been written. If the object type was the same but the data was not, forgoing this measure would cause old data to be re-sent.
Once the data is sent, in lieu of queueing traffic on arrival an event-driven networking scheme was adopted. Powered by having each NetworkConnection run in its own thread, communication can happen in near real-time so each packet can be handled and responded to, if necessary, immediately on arrival. This is done through the PacketListener interface that allows implementing classes to have a method define how to handle each type of incoming network traffic. Due to the concurrent nature of the networking environment, however, thread synchronization was necessary to ensure network-updated variables were handled safely.

4.11 Hub Control System

The server-side application running the hub handles several features of quadrotor control. Firstly, it handles a number of aspects of the telemetry of the quadrotors. In this sense, it takes care of quadrotor positioning by applying different behaviors that calculate its position. In the hub application, each quadrotor is represented by an object which also contains a property that keeps track of the connection information for that specific quadrotor. Because the hub runs in parallel with several threads to distribute the network update procedure, all quadrotor objects are stored in the main hub instance and then assigned to
threads that are dynamically created to distribute load. From this central storage point, references are created and passed down the control structure to give implemented behaviors and other hub control mechanisms access to data on the entire set of quadrotors.

In order to ensure that each multicopter is updated at the same time interval, a global timer is utilized in the Hub class which each of the HiveInstance objects register with in order to be notified when it is time to update their respective members. When the update event is triggered, each HiveInstance directly requests fresh position, velocity, and acceleration information from each multicopter it is assigned so when behaviors are applied the data is up-to-date. Once new values are calculated, they are then sent directly to each member so they can be applied.

4.12 Multi-Behavior System

![Figure 4.19: Behavior class structure](image-url)
In order to allow for much more precise control over how each individual quadrotor behaves at any given moment, a multi-behavior system was integrated into the hub. To allow for such a system, the hub must keep track of each individual quadrotors currently applied behaviors at each update tick. When the update event is triggered, the hub signals each multicopter to update its environment information and apply its behaviors according to a method inherited from the Behavior abstract class.

To allow for multiple behaviors to collectively modify the state of the multicopter, an UpdateContext object was created. This object contains a Velocity object to be modified by each behavior to define how the multicopter should adjust its position. The UpdateContext object also stores a list of neighboring multicopters that is populated at each update interval (figure 4.20) immediately before any behaviors are applied. This is a key feature as it allows us to only need to populate the neighborhood once for every HiveMember every update rather than once for every behavior. This operation is one of the most computationally expensive operations of the Hub (especially at large group sizes).

```java
Vector<HiveMember> getNeighborhood(HiveMember ref)
{
    Vector<HiveMember> n = new Vector<HiveMember>();
    for (number i = 0; i < this.myGroup.size(); i++)
        if (ref.getID() != this.myGroup.get(i).getID() &&
            getDistanceBetween(ref, this.myGroup.get(i)) <= NEIGHBORHOOD)
            n.add(this.myGroup.get(i));

    return n;
}
```

Figure 4.20: Neighborhood population function

Once each behavior has been applied to the UpdateContext, a maximum velocity parameter is enforced upon the resulting context to prevent overly-rapid acceleration.
Afterwards, the velocity is used to calculate the next position of the multicopter as well as the next acceleration value and each property is updated internally before updating the multicopter over the network. Pseudocode for this process is outlined in figure 4.20.

```java
void applyBehaviors()
{
    // Create the context and set the neighborhood so we only calculate it once.
    UpdateContext context = new UpdateContext();
    context.setNeighborhood(this.myGroup.getNeighborhood(this));

    // Apply all of the behaviors
    for (number i = 0; i < this.myBehaviors.size(); i++)
        this.myBehaviors.get(i).apply(context);

    // Ensure the directional velocities are within the limit
    for (number dir = VALUE_X; dir <= DIMENSION - 1; dir++)
    {
        number v = context.getVelocity().getPoint(dir);
        if (v < 0 && Math.abs(v) > MAX_VELOCITY)
            context.getVelocity().setPoint(dir, MAX_VELOCITY * -1);
        else if (Math.abs(v) > MAX_VELOCITY)
            context.getVelocity().setPoint(dir, MAX_VELOCITY);

        if (isNaN(v))
            context.getVelocity().setPoint(dir, 0.0);
    }

    this.myPosition = this.getNextPosition(context);
    this.myAcceleration = this.getNextAcceleration(context);
    this.myVelocity = context.getVelocity();
}
```

Figure 4.21: Function to apply each enabled behavior

By allowing each member to have its own combination of behaviors and parameters, many possibilities unfold. One such possibility is organizing the multicopters in different patterns or formations. This can be accomplished by applying the targeting behavior to each drone and picking a unique position for each member. In order to prevent the drones from moving towards each other, all that need be done is simply disable the cohesion behavior among any others that move the multicopters. This can be seen operating in figure 4.22 below.
4.13 Simulation Flocking Behaviors

The individual flocking behaviors that the simulator implemented are based heavily on Reynold’s and Oweis et al.’s methods, with some modifications as well as other changes in order to make each parameter work within a concurrent multi-behavioral ecosystem.

To apply each of the flocking behaviors outlined in sections 3.5.1 - 3 each rule must be applied to a single quadrotor at a time, as each object within the hub applies its own behaviors to itself according to each behaviors apply method. Each rule is defined in figures 4.23 - 4.26.

With regards to the rule of separation (figure 4.22), an example of a distributed take on the rule can be seen. This is driven by the fact that each flocking rule takes in the aforementioned flock member ID and the direction and calculates the result of applying that rule given the current state of the flock. There were some issues with infinite values being calculated in some circumstances, so the method had to account for that.

```java
number getSeparationVelocity(number coord, Vector<HiveMember> neighborhood)
{
    if (neighborhood.isEmpty())
        return 0.0;

    number d = 0.0;
```
for (number n = 0; n < neighborhood.size(); n++)
    d += (1 / getDistanceBetween(this.myMember, neighborhood.get(n), coord));

number ret = this.getSeparationStrength() * d;
if (isInfinite(ret))
    return 0.0;

return ret;

Figure 4.23: Separation velocity behavior method
	number getAlignmentVelocity(int coord, Vector<HiveMember> neighborhood) {
    if (neighborhood.isEmpty())
        return 0.0;

    number vn = 0.0;
    for (number n = 0; n < neighborhood.size(); n++)
        vn += neighborhood.get(n).getVelocity(coord);

    return this.getAlignmentStrength() * (vn / neighborhood.size());
}

Figure 4.24: Alignment velocity behavior method
	number getCohesionVelocity(int coord, Vector<HiveMember> neighborhood) {
    if (neighborhood.isEmpty())
        return 0.0;

    number pn = 0.0;
    for (number n = 0; n < neighborhood.size(); n++)
        pn += neighborhood.get(n).getPosition(coord);

    return this.getCohesionStrength() * (pn / neighborhood.size()) -
           this.myMember.getPosition(coord);
}

Figure 4.25: Cohesion velocity behavior method

public double getTargetingVelocity(int coord) {
    if (this.myTarget == null)
        return 0.0;

    return this.getTargetingStrength() *
           (this.myTarget.getPoint(coord) - this.myMember.getPosition(coord));
}

Figure 4.26: Target velocity behavior method
number getAvoidingVelocity(number memberId, number obstacleId, number direction)
{
    double[] P_me = this.myGroup.get(memberId).getPosition().toArray();
    double[] P_obstacle = this.myObstacles.get(obstacleId).toArray();

    number G = 1 / sqrt(
        pow(P_me[X] - P_obstacle[X], 2) +
        pow(P_me[Y] - P_obstacle[Y], 2) +
        pow(P_me[Z] - P_obstacle[Z], 2)
    );

    return (getAvoidStrength() * G *
            (signum(P_i[coord]) - signum(P_R[coord])));
}

Figure 4.27: Avoiding velocity behavior method
5. Simulation Results

The simulation results display the movement of the Quadrotor as it flies through its environment. For our results we state that a smooth motion is positive effect and a jittery motion is a negative effect. Smooth motion will allow the quadrotor to have better metrics for calculating the relative position and therefore will allow the quadrotor to send more accurate positioning information to the cloud.

We determine smooth motion by using the locally weighted scatterplot smoothing, LOESS, non-parametric regression function in the R language to calculate the standard error from the returned value. [36] The lower the standard error the smoother the line is.

5.1 Attitude Control

5.1.1 Dikmen vs. Kemper vs. Hoffman

The graphs show how the attitude controller moves from an initial position of (0,0,0) to its set desired position (1,1,1). In the case of the attitude controller we wanted the smoothest motion in the angular movement.
In figure 5.1 all of the Attitude controllers properly go to the desired location smoothly and accurately with the fastest attitude controller being Dikmen which is a sliding-mode controller arriving at 1 meter in the x direction in 27.42 seconds. Kemper’s attitude controller reached 1 meter in 36.48 seconds. Hoffman’s attitude controller arrived at 1 meter in 46.86 seconds.

![Figure 5.1: X Comparison](image)
Similar behavior is shown in Figure 5.2 for the Y direction. Dikmen’s attitude controller reached 1 meter in the y direction in 32.79 seconds, Kempers attitude controller arrived at 1 meter in 44.16 seconds, and Hoffmans attitude controller arrived in 46.08 seconds.

Figure 5.2: Y Comparison
Figure 5.3 shows how each individual Attitude Controller method for the Quadrotor comes from a distance of 0 in the z direction to a distance of the desired location at point 1. All of the attitude controllers properly go to the desired location. Both Kemper and Hoffman automatically move upwards from their starting position converging at 1 meter in 35.07 seconds and 39.33 seconds respectively. Dikmen’s attitude controller drops to -0.5657 meters in the first 1.68 seconds and then converges at 1 meter in 28.02 seconds. In a real world implementation the dikmen attitude controller would not be accurate enough as a sudden drop in altitude could result in a collision.
Figure 5.4 and 5.5 show how each individual attitude controller changes their degrees according to how close they are to their desired location. The Dikmen attitude
controller has the highest spikes and the greatest amount of jitters, meaning that it rapidly changes its angle based on its rapid change in movement, the spikes are greatest when the change in direction is greatest. The standard error for the phi and theta values are 5.7 and 5.26 respectively. Hoffman and Kemper attitude controllers are almost identical to each other except for the fact that Kemper converges in 30.78 seconds for both the phi and theta values, Hoffman’s converges in 38.46 seconds. The standard errors for Kemper in the phi and theta direction are 5.22 and 4.91 respectively. The standard errors for Hoffman in the phi and theta direction are 6.89 and 9.67 respectively.

Figure 5.6: Psi Comparison

Figure 5.6 shows how each individual attitude controller stays at its desired psi position which is 0, Kemper reacts best as it does not change its psi position at all. The standard error for Kemper’s attitude controller in the psi direction is 4.13. Dikmen jitters from
0-5 and 20-28 where it also jitters in the phi and theta positions, it eventually stabilizes itself after the 30 millisecond mark. The standard error for Dikmen’s attitude controller is 11.69. Hoffman’s attitude controller does not focus on the psi position as such it spikes whenever the altitude of the quadrotor changes dramatically which is at the 0 second mark and the 30 millisecond mark. The standard error for Hoffman’s attitude controller is 3.68.

Among the three attitude control methods that were compared in the simulator by Kemper, Hoffman, and Dikmen respectively, it was concluded that Hoffman’s PD controller method was the smoothest as its average standard error was 4.69290308. Kemper’s average standard error was 5.4937965 and Dikmen’s was 6.32369. All of the Standard errors for the attitude controllers are shown in table 5.1.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Kemper</th>
<th>Hoffman</th>
<th>Dikmen</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>4.384743</td>
<td>1.601325</td>
<td>4.177628</td>
</tr>
<tr>
<td>Y</td>
<td>3.631218</td>
<td>1.438495</td>
<td>3.571708</td>
</tr>
<tr>
<td>Z</td>
<td>10.68106</td>
<td>4.881332</td>
<td>7.573814</td>
</tr>
<tr>
<td>Phi</td>
<td>5.21786</td>
<td>6.888175</td>
<td>5.700442</td>
</tr>
<tr>
<td>Theta</td>
<td>4.917646</td>
<td>9.668774</td>
<td>5.259321</td>
</tr>
<tr>
<td>Phi</td>
<td>4.130222</td>
<td>3.679341</td>
<td>11.65922</td>
</tr>
</tbody>
</table>
Overall the final average of the attitude controllers shows that their smoothness is only best at certain angular and linear directions, and the difference in smoothness is less than one. This is not an appropriate comparison so we tested the attitude controllers performance by adding in white noise to the sensors.

5.1.2 With Sensor Noise

The graphs below show a comparison of the attitude controller with and without noise. The noise variance used in the attitude controller comparison graphs is 0.05.

5.1.2.1 Dikmen With Sensor Noise
The standard error in the X direction with noise is 4.622.
The standard error in the Y direction with noise is 3.785.

![Figure 5.9: Dikmen’s Z comparison with and without noise](image)

The standard error in the Z direction with noise is 7.995.

![Figure 5.10: Dikmen’s Phi comparison with and without noise](image)
The standard error in the phi direction with noise is 11.12.

![Time vs. Theta Position](image)

**Figure 5.11: Dikmen’s Theta comparison with and without noise**

The standard error in the theta direction with noise is 10.51.
The standard error in the theta direction with noise is 11.89.

5.1.2.2 Kemper With Sensor Noise
Figure 5.13: Kemper’s X comparison with and without noise

The standard error in the x direction with noise is 4.338.
The standard error in the y direction with noise is 3.51.

The standard error in the z direction with noise is 10.176.
Figure 5.16: Kemper’s Phi comparison with and without noise

The standard error in the phi direction with noise is 5.867.
The standard error in the theta direction with noise is 5.791.

The standard error in the psi direction with noise is 11.877.
5.1.2.3 Hoffman With Sensor Noise

![Time vs. X Position](image)

**Figure 5.19: Hoffman’s X comparison with and without noise**

The standard error in the x direction with noise is 5.644.
Figure 5.20: Hoffman’s Y comparison with and without noise

The standard error in the y direction with noise is 2.135.
The standard error in the z direction with noise is 5.032.
The standard error in the phi direction with noise is 13.837.
Figure 5.23: Hoffman’s Theta comparison with and without noise

The standard error in the theta direction with noise is 12.308.
The standard error in the psi direction with noise is 11.82.

5.1.2.4 Attitude Controller Noise Final

The attitude controller that has the smoothest motion with noise is Kemper with the average standard error being 5.94701. Dikmen’s average standard error is 8.319959 and Hoffmans is 9.880259. Kemper has better performance as the difference between it and Dikmen’s is greater than 2. All of the Standard errors for the attitude controllers with noise are shown in table 5.2.
### Table 5.2: Standard Errors For Attitude Controllers with noise.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Kemper W/ Noise</th>
<th>Hoffman W/ Noise</th>
<th>Dikmen W/ Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>3.099703</td>
<td>5.685</td>
<td>4.622426</td>
</tr>
<tr>
<td>Y</td>
<td>2.152327</td>
<td>4.784024</td>
<td>3.785161</td>
</tr>
<tr>
<td>Z</td>
<td>10.2911</td>
<td>7.357921</td>
<td>7.995629</td>
</tr>
<tr>
<td>Phi</td>
<td>4.871248</td>
<td>14.01324</td>
<td>11.12085</td>
</tr>
<tr>
<td>Theta</td>
<td>4.855147</td>
<td>12.34499</td>
<td>10.50764</td>
</tr>
<tr>
<td>Phi</td>
<td>10.41253</td>
<td>15.09638</td>
<td>11.88806</td>
</tr>
<tr>
<td>Overall</td>
<td>5.94701</td>
<td>9.880259</td>
<td>8.319959</td>
</tr>
</tbody>
</table>

5.2 Obstacle Detection

The graphs show how the obstacle detection methods handle flying through a simple hallway to reach a point at the end of the hallway. The quadrotor starts on a platform positioned at (0,0,0), this is the takeoff pad. There are 3 different hallways that the quadrotor must fly between, the first set is a hallways of a width of 7 meters, this hallway is only 2 meters long and its center is at position (3,0,0). The second hallway is 6 meters in width, 2 meters long and its center is at (5,0,0). The third hallway is only 5 meters in width, 6 meters
long and its center is at (7,0,0). The quadrotor’s goal is at position (12,0,5) in the environment, this is located 2 meters away from the end of the third hallway.

An obstacle detection method is stated as performing well if its movement is smooth when moving through our obstacle course. The obstacle avoidance method should force the quadrotor to not collide with obstacles at all cost.

Figure 5.25: X Comparison

Figure 5.25 shows the quadrotor starting at 0 meters in the X direction and moving to position 12 in the environment. The Gageick obstacle avoidance method arrived at position 12 in 123.43 seconds, Wagsters reached that point in 150.12 seconds.
Figure 5.26 shows the quadrotor starting at 0 meters and changing according to the position the quadrotor is in the hallway. Gageick is in the first hallway at time markers 18.06 to 35.91, Wagster is in the first hallway at time markers 19.35 to 42.09. The second hallway’s time markers are 35.91 to 51.57 and 42.09 to 64.74 seconds for Gageick and Wagster respectively. The third hallway’s time marker’s are 51.57 to 95.00 and 64.74 to 120.93 seconds for Gageick and Wagster respectively.

The maximum value for Gageick along the Y axis is 0 meters while the minimum value is -0.72 meters. Wagster’s maximum value is -0.179 meters and the minimum is 0.21 meters.

The reason why the difference in Y position for Wagster is smaller than the Y position for Gageick is that Wagster takes into account the left and right ultrasonic sensor values and attempts to position itself in the center of those values. Gageick’s obstacle avoidance method
does not take into account the values of the wall, except for when the values are too close for collisions to occur, this causes spikes as shown on time markers 29.52 and 89.34 seconds when Gageick’s method responds to the quadrotor being too close to the hallway’s right wall.

![Figure 5.27: Z Comparison](image)

Figure 5.27 shows the quadrotor starting at 0 meters, taking off, and moving to an altitude of 5 meters. Both Gageick and Wagster’s method have a constant heading with small changes in altitude.
Figure 5.28: Phi Comparison

Figure 5.29: Theta Comparison
Figures 5.28, 5.29, and 5.30 shows the quadrotor’s phi, theta, and psi degrees according to their time marker denoting their position in the multiple hallways.

In the phi direction the standard error for Gageick was 36.44 while the standard error for Wagster was 34.74. In the theta direction the standard error for Gageick was 39.48 while Wagster was 38.23. In the psi direction the standard error for Gageick was 39.53 while the standard error for Wagster was 18.57.

It was concluded then that Wagster’s obstacle detection method was the smoothest as its average standard error was 28.00426 while Gageick’s average was 33.4292. Wagster was also better at avoiding the walls during the flight through the hallways. All standard errors are shown in table 5.3.
<table>
<thead>
<tr>
<th>Comparison</th>
<th>Wagster</th>
<th>Gageick</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1.156503</td>
<td>6.224846</td>
</tr>
<tr>
<td>Y</td>
<td>35.09231</td>
<td>37.9196</td>
</tr>
<tr>
<td>Z</td>
<td>40.22925</td>
<td>40.98503</td>
</tr>
<tr>
<td>Phi</td>
<td>34.7439</td>
<td>36.44013</td>
</tr>
<tr>
<td>Theta</td>
<td>38.23012</td>
<td>39.47894</td>
</tr>
<tr>
<td>Phi</td>
<td>18.57348</td>
<td>39.52665</td>
</tr>
<tr>
<td>Overall</td>
<td>28.00426</td>
<td>33.4292</td>
</tr>
</tbody>
</table>

Table 5.3: Standard Errors For Obstacle Detection through multiple hallways.

5.3 Altitude Controller Results

These graphs show how the altitude controller uses the takeoff and landing methods using the implementation described in section 3.4. There is a platform in the environment positioned at (0,0,0) this is where all of the quadrotors takeoff and land. The attitude controller used is the kemper attitude controller.
Figure 5.31: Takeoff method in Altitude Controller.

Figure 5.31 shows the quadrotor starting on the platform at 1.46 meters since that is the height of the quadrotor with an ultrasonic sensor affixed to it on the bottom. The quadrotor then goes into takeoff mode from the initial phase starting at 0.24 seconds, then the second phase starting at 3.11 meters at 0.84 seconds. The final phase occurred at 1.92 seconds and hovered at a steady 5.63 meters.
Figure 5.32: Height Comparison for Landing method in Altitude Controller.

Figure 5.32 shows the quadrotor starting at a specific height, going into landing mode, and then landing from that height on to the platform. The landing height is positioned at 1.46 meters since the ultrasonic sensor is still affixed to the bottom. In the graph all landings have a slight curve before they land, this shows a smooth landing. At ten meters the quadrotor smooths its height at 3.07 meters at 5.1 seconds and lands at 7.8 seconds. At twenty meters the quadrotor smooths its height at 3.19 meters at 9.03 seconds and lands at 13.38 seconds. At thirty meters the quadrotor smooths its height at 3.12 meters at 12.63 seconds and lands at 14.82 seconds.

The greatest standard error is lower than 1 showing that the landing method does indeed ensure a smooth landing. All of the standard errors and times are shown in table 5.4.
<table>
<thead>
<tr>
<th>Starting Height</th>
<th>Smooth Time</th>
<th>Landing Time</th>
<th>Standard Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Meters</td>
<td>5.1</td>
<td>7.8</td>
<td>0.8512206</td>
</tr>
<tr>
<td>20 Meters</td>
<td>9.03</td>
<td>13.38</td>
<td>0.6584615</td>
</tr>
<tr>
<td>30 Meters</td>
<td>12.63</td>
<td>14.82</td>
<td>0.3058578</td>
</tr>
</tbody>
</table>

Table 5.4: Standard Errors For Attitude Controllers with noise.

5.4 Hub Results

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment Strength</td>
<td>1.0</td>
</tr>
<tr>
<td>Cohesion Strength</td>
<td>2.0</td>
</tr>
<tr>
<td>Separation Strength</td>
<td>2.0</td>
</tr>
<tr>
<td>Targeting Strength</td>
<td>2.0</td>
</tr>
<tr>
<td>Neighborhood Size</td>
<td>5.0</td>
</tr>
<tr>
<td>Starting Position</td>
<td>Random</td>
</tr>
<tr>
<td>Target Point</td>
<td>{ 5.0, 5.0, 5.0 }</td>
</tr>
<tr>
<td>Hub Update Time</td>
<td>300 milliseconds</td>
</tr>
<tr>
<td>Maximum Velocity</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Table 5.5: Constants used in this experiment

To show the results of applying the flocking method outlined in section, nine graphs were generated showing the X, Y, and Z position, velocity, and acceleration of each group member vs. time. Constants used for each flocking behavior in this experiment can be seen in
table 5.5. For this experiment, all flocking behaviors were enabled. The results can be seen in figures 5.33-5.38 which show the simulation running with eight flock members.

Figure 5.33: Time vs. X Position

Figure 5.33 shows the X position of each group member over time during the execution of the simulation. This shows that each of the flocking behaviors is being applied correctly.
As can be seen from Figure 5.34, Y Position over time looks strikingly similar to the X Position over time graph from Figure 5.33 with differences in the order (from largest value to smallest over time) of group members. This is because though members may be close in one dimension, they are not in another in order to accommodate. Another interesting point to note is that the Z position of the target point was 5 and in the graph it can be clearly seen that each group member is tending to that point.
Figure 5.35: Time vs. Z Position
Figure 5.36: Time vs. X Velocity

The X velocity was by far the most jittery out of all of the velocities. It is mainly caused by the fact that each member is pushing and pulling (with the separation and cohesion behaviors). Still, the range of X velocities never goes below -0.3 or above 0.3.
The cone shape that is visible in 5.37 from the 2000 millisecond mark to the 8000 millisecond mark shows the gradual taper of each units Y velocity to be ±0.2. This is expected behavior especially for groups that start in random locations since they have to group up before the velocities get within ±0.15.
What can be seen in Figure 5.38 is the velocity of each group member in the Z direction. Though sporadic at times, it shows many dispersed velocities for each member slowly join together by the 8000 millisecond mark. Referring to Figure 5.35, it can be seen that between the 6000 and 8000 millisecond mark Unit6 is underneath Unit0. When Unit6 attempted to ascend, the separation behavior caused it to descend to avoid collision.
The acceleration in the X direction was fairly constant for most of the execution of the simulation. Some units varied more than other, such as Unit5 near the 2000 millisecond mark as well as around the 600 millisecond mark. Other group members, such as Unit6 and Unit7 had far more consistent accelerations.
Figure 5.40: Time vs. Y Acceleration

Figure 5.40 looks a lot like 5.39 though the variation after the 4000 millisecond mark is far less than in the X acceleration graph. This is reflective of the very low update time which allows for much smoother flight once each member settles into its position within the group.
Finally, in figure 5.41 it can be seen that the Z acceleration was very sporadic though it largely was between 5 and -5. These outliers are caused by sudden stops by subgroups of members turning around to move towards the target point.

For the hub backend behavior parameters, experimentation was used to determine optimal constants for operation. Specifically, these include the separation, alignment, cohesion, targeting, and avoiding strength constants that Oweis et al. use as well as a maximum velocity parameter and update time.
These constants apply in situations where the flock size is greater than one, otherwise the constants will alter expected results. For example, sending a flock with a single quadrotor to position \( \{x : 1, y : 1, z : 1\} \) with a targeting strength of \( t = 2 \) will cause the quadrotor to go to \( \{x : 2, y : 2, z : 2\} \). This is because a single quadrotor doesn’t need to worry about some of the 2 other three basic flocking rules of separation, alignment, and cohesion since there aren’t any other flock members. The other strength constants are also irrelevant for flocks of size one as the other rules are not applied without any other members. This can be seen in

For flocks of size greater than one, these constants control the behaviors that are prioritized. A higher targeting constant causes the entire flock (as a group) to tend to move towards the target point. The same goes for the other strength constants, though making the cohesion strength the same value as the separation strength has no effect. This is because the cohesion rule is keeping the flock together and the separation rule is making sure they’re also apart.

Another important constant is the update time (in milliseconds). This constant affects how frequently the cloud instances update the flock with new position corrections. Several values were tested and it was found that longer update times would cause the flock to “stutter”. This isn’t as fluid as normal flocking behaviors that can be observed in nature, so a smaller update value is preferable.

The maximum velocity parameter is important as it defines how far from a quadrotors current position it can travel each update. This constant must be modified in relation with the
update time in order to achieve fluid motion. A good rule of thumb for this constant is to have it as $V_{max} = \frac{U}{100}$ since it typically takes the quadrotor slightly less than a second to travel one meter within the environment.
6. Verification

6.1 Relative Positioning

Relative positioning is produced by using the outputs of the linear acceleration of the accelerometer and the angular acceleration of the gyroscope to output a linear and an angular position in the environment. For our Verification we assumed that we knew our angular position from the output of the gyroscope and we knew how our output from our accelerometer was being calculated shown in equation 4.15.

From this information our known values were our perceived acceleration in the x, y and z direction, $a_{px}$, $a_{py}$, and $a_{pz}$ respectively. From the relative positioning of our gyroscope our known values were are angular positions phi, theta, and psi, $\lambda_\phi$, $\lambda_\theta$, and $\lambda_\psi$ respectively. Our unknown values that we were solving for are the relative acceleration in the x, y and z direction, $a_{dx}$, $a_{dy}$, and $a_{dz}$ respectively.

We first solve for our relative acceleration in the x direction in the steps below:

\begin{align*}
    a_{px} &= a_{dz} \cdot \cos(\lambda_\phi) + a_{dy} \cdot \sin(\lambda_\phi) + a_{dz} \cdot \sin(\lambda_\phi) \quad (6.1) \\
    a_{px} - a_{dy} \cdot \sin(\lambda_\phi) - a_{dz} \cdot \sin(\lambda_\phi) &= a_{dx} \cdot \cos(\lambda_\phi) \quad (6.2) \\
    a_{dx} &= \frac{a_{px} - a_{dy} \cdot \sin(\lambda_\phi) - a_{dz} \cdot \sin(\lambda_\phi)}{\cos(\lambda_\phi)} \quad (6.3) \\
    a_{dx} &= \frac{u_{px} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)}{\cos(\lambda_\phi)} \quad (6.4)
\end{align*}
Solve for the relative acceleration in the y direction using the solved value of the relative acceleration in the x direction:

\[ a_{py} = a_{dx} \cdot \cos(\lambda_\theta) + a_{dy} \cdot \cos(\lambda_\theta) \]

\[ a_{py} = a_{dx} \cdot \sin(\lambda_\theta) - a_{dy} \cdot \sin(\lambda_\theta) = a_{dy} \cdot \cos(\lambda_\theta) \]

\[ a_{dy} = \frac{a_{py} - a_{dx} \cdot \sin(\lambda_\theta) - a_{dy} \cdot \sin(\lambda_\theta)}{\cos(\lambda_\theta)} \]

\[ a_{dy} = \frac{a_{py} - a_{dx} \cdot \tan(\lambda_\theta) - a_{dy} \cdot \tan(\lambda_\theta)}{\cos(\lambda_\theta)} \]

\[ a_{dy} = \frac{a_{py}}{\cos(\lambda_\theta)} - \left( \frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi) \right) \cdot \tan(\lambda_\theta) + a_{dz} \cdot \tan(\lambda_\theta) \]

\[ a_{dy} = \frac{a_{py}}{\cos(\lambda_\theta)} - \frac{a_{px} \cdot \tan(\lambda_\theta)}{\cos(\lambda_\phi)} + \left( a_{dy} \cdot \tan(\lambda_\phi) \cdot \tan(\lambda_\theta) \right) - a_{dz} \cdot \tan(\lambda_\theta) \]

\[ a_{dy} = a_{dy} \cdot \left( 1 - \left( \tan(\lambda_\phi) \cdot \tan(\lambda_\theta) \right) \right) = \frac{a_{py}}{\cos(\lambda_\theta)} - \frac{a_{px} \cdot \tan(\lambda_\theta)}{\cos(\lambda_\phi)} \]

\[ a_{dy} = a_{dy} \cdot \left( 1 - \left( \tan(\lambda_\phi) \cdot \tan(\lambda_\theta) \right) \right) = \frac{a_{py}}{\cos(\lambda_\theta)} - \frac{a_{px} \cdot \tan(\lambda_\theta)}{\cos(\lambda_\phi)} \]

\[ a_{dy} = \frac{a_{py}}{\cos(\lambda_\phi)} - \frac{a_{px} \cdot \tan(\lambda_\theta)}{\cos(\lambda_\phi)} + \left( a_{dz} \cdot \tan(\lambda_\phi) \cdot \tan(\lambda_\theta) \right) - a_{dz} \cdot \tan(\lambda_\theta) \]
Solve for the relative acceleration in the z direction using the solved value of the relative acceleration in the x and y direction:

\[
a_{pz} = a_{dx} \cdot \cos(\lambda_\psi) + a_{dy} \cdot \sin(\lambda_\psi) + a_{dz} \cdot \sin(\lambda_\psi)
\]

\[
a_{pz} = a_{dx} \cdot \sin(\lambda_\psi) - a_{dy} \cdot \sin(\lambda_\psi) = a_{dz} \cdot \cos(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz} - a_{dx} \cdot \sin(\lambda_\psi) - a_{dy} \cdot \sin(\lambda_\psi)}{\cos(\lambda_\psi)}
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - a_{dx} \cdot \tan(\lambda_\psi) - a_{dy} \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
a_{dz} = \frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
\frac{a_{pz}}{\cos(\lambda_\psi)} - \left(\frac{a_{px}}{\cos(\lambda_\phi)} - a_{dy} \cdot \tan(\lambda_\phi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
\left(\frac{a_{pz}}{\cos(\lambda_\psi)} - \frac{a_{px} \cdot \tan(\lambda_\phi)}{\cos(\lambda_\phi)} + \left(a_{dz} \cdot \tan(\lambda_\phi) \cdot \tan(\lambda_\psi) - a_{dz} \cdot \tan(\lambda_\phi)\right) \cdot \tan(\lambda_\psi)
\]

\[
\cdot \tan(\lambda_\psi) \cdot \left(\tan(\lambda_\phi) - 1\right)
\]

\[
125
\]
From the calculation of the relative acceleration in the z direction we can calculate the relative acceleration in the y direction. Then we can calculate the relative acceleration in the x direction from the results of the relative acceleration in the y and z direction.
6.2 Relative Position Results

These graphs display how similar the positions are from the perceived position from the relative position calculation to the real position of the quadrotor in the environment. There is a platform in the environment positioned at (0,0,0) this is where all of the quadrotors takeoff. This is to initialize the quadrotors gyroscope and accelerometer. The attitude controller used is the kemper attitude controller. Once the takeoff procedure is successful the attitude flight path is positioned at (2,2,5). The average standard error of the phi, theta, and psi with relative positioning is 10.448. The maximum difference of the simulated vs. calculated position is 1.47 in the Z direction. This is caused by the altitude of the quadrotor moving rapidly.

![Figure 6.1: Comparison of Real and Relative X position.](image)
Figure 6.1 shows the X position in the environment vs. the relative X position calculated by the quadrotor. The standard error for the relative position is 1.649. The greatest difference of the real vs. relative position is 0.406 meters.

![Time vs. Y Position](image)

**Figure 6.2: Comparison of Real and Relative Y position.**

Figure 6.2 shows the Y position of the quadrotor in the environment vs. the calculated relative position in the environment. The standard error in the y direction is 1.364 meters, and the maximum difference is 0.399.
Figure 6.3 shows the Z position of the quadrotor in the environment vs. the calculated relative position in the environment. The standard error in the z direction is 11.358, and the maximum difference is 1.475 meters. This maximum difference only occurred during the peak of the takeoff method.
Figure 6.4 shows the phi position of the quadrotor in the environment vs. the calculated relative position in the environment. The standard error in the y direction is 10.29, and the maximum difference is 0.00149 meters.
Figure 6.5 shows the theta position of the quadrotor in the environment vs. the calculated relative position in the environment. The standard error in the y direction is 10.30, and the maximum difference is 0.00149 meters.
Figure 6.6: Comparison of Real and Relative Psi position.

Figure 6.6 shows the psi position of the quadrotor in the environment vs. the calculated relative position in the environment. The standard error in the psi direction is 10.70, and the maximum difference is 4.56e-05 meters.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Standard Error</th>
<th>Standard Error</th>
<th>Overall Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simulated</td>
<td>Calculated</td>
<td>Difference</td>
</tr>
<tr>
<td>X</td>
<td>2.910365</td>
<td>1.648658</td>
<td>0.4068883</td>
</tr>
<tr>
<td>Y</td>
<td>2.398179</td>
<td>1.363661</td>
<td>0.3996819</td>
</tr>
<tr>
<td>Z</td>
<td>11.23941</td>
<td>11.35762</td>
<td>1.474556</td>
</tr>
<tr>
<td></td>
<td>Simulated</td>
<td>Calculated</td>
<td>Difference</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>Phi</td>
<td>10.21809</td>
<td>10.26872</td>
<td>0.001490871</td>
</tr>
<tr>
<td>Theta</td>
<td>10.25515</td>
<td>10.29908</td>
<td>0.001490871</td>
</tr>
<tr>
<td>Phi</td>
<td>10.74641</td>
<td>10.69983</td>
<td>4.5587e-05</td>
</tr>
<tr>
<td>Overall</td>
<td>7.961267</td>
<td>7.606261</td>
<td>0.3806922</td>
</tr>
</tbody>
</table>

Table 6.1: Standard Errors and Overall Maximum Difference of Simulated and Calculated position.

From the data it shows that the relative positioning is pretty accurate with Kempers attitude controller. The difference in the overall standard error for the simulated position compared to the calculated position with the relative positioning method is less than .4, and the overall maximum difference is 0.38. This shows that the relative positioning method works.
7. Conclusion

In this paper we demonstrated a complete mathematical model for flocking multiple autonomous quadrotors using a distributed multi-behavioral control system. We detailed the complete software design to construct the quadrotors and laid out the hardware requirements and structure of the cloud backend. Additionally, this paper compared and created different techniques for local and remote obstacle avoidance and detection, quadrotor attitude and altitude control as well as flocking. The simulation covered in this paper showed that we can achieve a scalable hub system to calculate and control each quadrotor’s velocities and positions with respect to its enabled behaviors. It also demonstrated the success of altitude and attitude control as well as obstacle avoidance algorithms computed on each quadrotor.

The results of this research can be applied to many different issues such as rapidly mapping indoor and outdoor areas, scalable search and rescue in many different emergency situations (like a forest fire or a mine collapsing) that does not depend on GPS. It could also be utilized to control multiple groups of different kinds of members such as land or sea drones as well.

Additional work for this paper would be to include the ability to as well as larger scale indoor and outdoor mapping. Also, adding the use of wireless cluster communication through quadrotors in order to reduce the number of flock members that need access to the hub backend.
References


[37] Internet: http://www.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf [April 21, 2015]