



Modeling and simulation of autonomous quadrotor systems and their onboard sensors

Simulation: Transactions of the Society for Modeling and Simulation International
2021, Vol. 97(3) 195–214
© The Author(s) 2020
DOI: 10.1177/0037549720974753
journals.sagepub.com/home/sim



Christian Micklish , Nathan Hilliard and Hala ElAarag

Abstract

Simulators are valuable tools to evaluate autonomous systems and methods of their control in a cost-effective and safe fashion. There are several implementations of simulators for autonomous systems in the literature. However, they provide limited capabilities in terms of the simulation of specific autonomous methods, for example control, flight, or sensor dynamics. In this paper, we present our design of a simulator for an environment that handles multiple quadrotors with their own set of sensors. We simulate three main sensors to get spatial and environmental information: an Inertial Measurement System that contains a three-axis gyroscope and an accelerometer; a pressure sensor; and ultrasonic sensors. To provide a realistic environment, we implement the thrust force, hub force, total torque, rolling moments, and equations of motion. We provide several attitude controllers, altitude controllers, obstacle detection methods, positioning methods, and flocking methods. We then present a comparison of their performance and behavior. Until the writing of this paper, there were no other simulators that provide this number of various autonomous methods. Moreover, the object-oriented design of our simulator makes it easy to support the addition of any other controller or method. Our simulator provides a comprehensive testbed for current and future methods for autonomous quadrotors.

Keywords

Autonomous systems, quadrotor, sensor simulation, attitude control, altitude control, obstacle detection, flocking

1. Introduction

Safe quadrotor control is always important for cost-conscious experiments with the high need for repeatability. Safety could be attained by either designing a mechanism around the quadrotor to keep the quadrotor and its environment unharmed or utilizing a simulator to create a true to life virtual environment.

The safety mechanism in most cases is a strong cage around the quadrotor that prevents damage in flight or the entrance of large objects to the rotor blades. Mulgaonkar et al.¹ designed a strong carbon fiber cage around a 25 gram pico-quadrotor. The light weight of the quadrotor, small size, and enveloping cage made it nearly harmless in collisions. Briod et al.² encapsulated a quadrotor with a cage on a three-axis gimbal to allow the cage to take the impact of the collision and rotate accordingly without a change in the angular orientation of the quadrotors. The prototype described was successfully flown autonomously through a dense forest, incurring multiple collisions with trees without damaging the quadrotor or the trees.

Quadrotor simulators allow for a more cost-effective and safer implementation of autonomous control without

physical limitations. Dong et al.³ developed a quadrotor simulator testbed for flight controls and trajectory generation. The algorithms are then implemented into a real system using the fine-tuned parameters obtained from the simulator. The results of the real system closely matched the simulator's results.

Our simulator has a modular, objected-oriented, and distributed design for a system of autonomous quadrotors and their onboard sensors. The simulated sensors are designed to have the same output as their real counterparts. Several autonomous methods, such as takeoff, landing, and obstacle detection, are implemented along with multiple attitude controllers, altitude controllers, and flocking methods. Our simulator creates a cost-effective and safe environment for testing the performance of different algorithms and fine-tuning their parameters.

Stetson University, Deland, FL, USA

Corresponding author:

Christian Micklish, Stetson University, 421 Woodland Blvd, Deland, FL 32723, USA.

Email: cmicklis@stetson.edu

2. Related work

Several researchers used or implemented simulators for the quadrotors and sensors necessary to provide autonomous flight. MATLAB and Simulink are commonly used for attitude and stabilization methods. Li and Li⁴ created a Proportional Integral Derivative (PID) controller inside Simulink. They defined all of the PID constants through simulation runs and ported the PID controller to a quadrotor testbed. Sellali and Allali⁵ used a Neuro-Fuzzy method and PID controller combination to optimize a quadrotor's PID constants during flight. The initial implementation and verification were done using MATLAB/Simulink. Once it was tested using Simulink, it was implemented on an F450 quadrotor. Rosca et al.⁶ combined a PID controlled quadrotor with a Brain Control Interface (BCI). The mathematical model and numerical simulations were initially done in Simulink and then were transferred to a Parrot AR Drone. As a simulation tool, Simulink is effective in the initial verification process of control mechanisms for quadrotors, but other simulators supply more detailed sensor information and environmental control.

The Robot Operating System (ROS) and Gazebo, a robotic simulation platform used for autonomous quadrotor research and verification, provides granular control of the quadrotor environment, physics engine, and sensor signals. Meyer et al.⁷ adapted the quadrotor model of the ROS and Gazebo simulators to accurately simulate complex in-flight aerodynamic effects from wind tunnel data. They updated the Inertial Measurement System (IMU), barometric sensor, ultrasonic sensor, magnetic field sensor, and Global Positioning System (GPS) Receiver. All sensors used a first-order Gauss–Markov error model. Curi et al.⁸ used ROS as a simulation platform to validate a sensor fusion algorithm with a Proportional Derivative (PD) controller. Santamaria-Navarro et al.⁹ created a model quadrotor with a dynamic robotic arm manipulator holding a visual observer at the end of the arm.

Advances in sensor simulation provide a purview into outputs produced by a sensor's perception of the environment. McAnanama and Marsden¹⁰ merged an IMU simulator from representative flight dynamics data with an open source Flight Dynamics Model (FDM). Holder et al.¹¹ and Durst et al.¹² compared Lidar data points from a real-life and similar simulated environment to determine the necessary accuracy of simulated Lidar used to provide spatial information to autonomous vehicles. Holder et al.¹¹ determined that simple geometric shapes in simulated Lidar data points proved suitable for many autonomous scenarios. Durst et al.¹² showed that discrepancies occur with vegetation, triggering the need for more accurate Lidar sensor signals for complex objects such as vegetation and rough terrain. Durst and Goodin¹³ altered the process of simulated microelectromechanical system (MEMS) sensors and GPS receivers by producing their signals from

first principle effects. The noise of the MEMS sensor signal is dictated by an equation using the internal and external temperature of the sensor and its total execution time. This closely mimics the noise produced by a MEMS accelerometer and gyroscope. The GPS receives positional information from simulated satellite positions and produces receiver noise from simulated clouds that alter positional information. This produces more realistic positional changes received by the GPS. Kranz¹⁴ collected ultrasonic sensor measurements on three different surface shapes in both a simulated and a real environment. The simulated sensors caused a completely different effect and were altered to match the data produced by the ultrasonic sensors. These advances in simulating sensors aid in providing more realistic and necessary environmental information from the sensor inputs to the outputs used by autonomous quadrotors.

The structures of these simulators are dependent on the research element investigated and do not have the capability of comparison to other research elements. Unlike other research, in this paper we present the implementation of a simulator that allows for multiple elements to be running at the same time in order to provide a more detailed image of the sensor usage, inertial structure, and operation during an autonomous flight. In addition, our simulator provides the capability of a more detailed exploration of control methods and their usage of sensory information.

3. The simulator

We developed a simulator in Java with two main components, the quadrotor component and the hub component. The implementation of the hub component is presented by Micklisch et al.,¹⁵ while a portion of the environment and quadrotor components is presented by Micklisch et al.¹⁶ Our simulator is a discrete-event simulator; that is, it advances the environment clock every few milliseconds. This allows for easily changing the granularity of the simulation. The simulator's sensors provide constant feedback about the simulated environment akin to their real-world counterparts.

3.1. Architecture

The simulator's quadrotor component and hub component are runnable as separate threads. The multithreading feature of our simulator allows for the simulation and management of a large number of agents with ease. The overall architecture of the simulator is illustrated in Figure 1.

The simulator needs to be invoked in two locations, the Main and the Hub. The QuadrotorController controls the Quadrotor and Sensors. The Model controls the Environment. It is invoked by the main controller. The Hub is invoked through the HiveController and initializes the Hub and all of the HiveInstances that run concurrently.

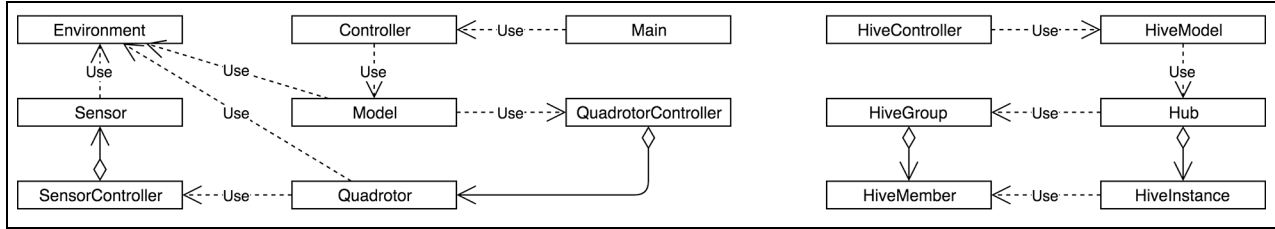


Figure 1. Entry point class diagram.

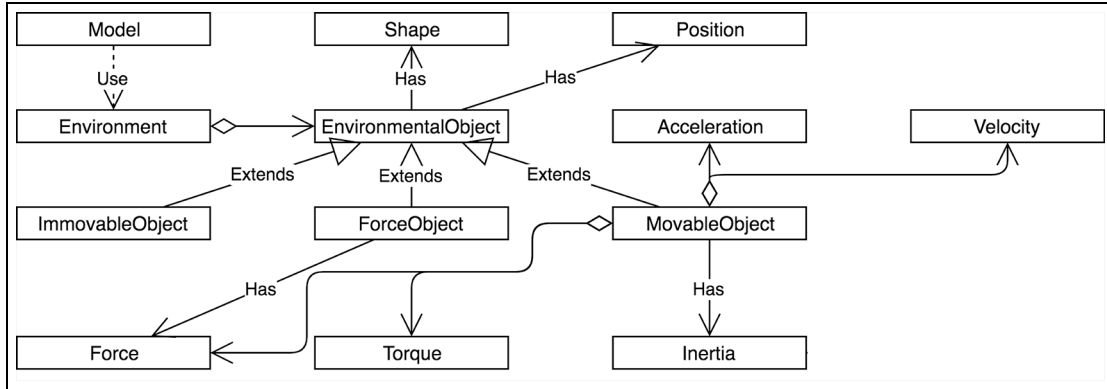


Figure 2. Environmental class diagram.

This design enables a large number of concurrent threads in each component to run without slowing down the simulation. Take, for instance, a quadrotor array of ultrasonic sensors. In practice, these run by sending out pings to the environment in regular intervals (ping rate) and the quadrotor checks against them at its leisure. In our simulator, this is done by running each of the sensors as their own thread so that even in the simulation the quadrotor 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.

3.2. Environment

The environment is a three-dimensional Euclidian space with an x , y , and z Cartesian coordinate system. Here, x is the length-wise direction, y is the width-wise direction, and z is the height-wise direction. The environment contains three different objects, Force, Movable, and Immovable. Shown in Figure 2, these objects extend the EnvironmentalObject, which provides a shape and position.

The ImmovableObject acts as an obstacle in the environment. There is no difference between the EnvironmentalObject and the ImmovableObject except its name.

The MovableObject moves freely inside the environment, adherent to Newtonian mechanics. It must be concerned with a system of forces acting on it. The

MovableObject contains all of the forces and torques applied to it, a set of all velocities and accelerations, and inertia that follows its shape and mass. These attributes enable the environment to calculate changes in the linear and angular positions, velocities, and accelerations.

The ForceObject simulates the forces of wind and gravity. The ForceObject is there to disturb the MovableObject and adds to its force and torque. It only affects the MovableObject when it is inside the ForceObject. If the MovableObject is partially inside of the ForceObject then a torque τ would be applied to the MovableObject from the force of the ForceObject, as shown in Equation (1), where F is the force from the ForceObject, r is the radius from the center of the acting force, and θ is the angle at which the force is acting on the surface of the movable object:

$$\tau = Fr \sin(\theta) \quad (1)$$

To calculate the torques acting on the movable object, a method was created to check if the movable object was inside a force object, as follows.

1. Define the center of the movable object as the center of the Euclidean space.
2. Get the maximum and minimum of the x , y , and z dimensions of the MovableObject, shown in Equations (2) and (3), respectively.

3. Get the maximum and minimum of the x , y , and z dimensions of the ForceObject, shown in Equations (4) and (5), respectively.
4. Compare the maximums and minimums of the x , y , and z dimensions of the MovableObject and ForceObject.
 - a. If the maximum position of the Movable Object is less than the maximum position of the ForceObject, and the minimum position of the MovableObject is greater than the minimum of the ForceObject, then the MovableObject is inside the ForceObject, meaning that there are no torques on the MovableObject.
 - b. If the maximum position of the ForceObject is greater than the maximum position of the MovableObject, and the maximum position of the MovableObject is greater than the minimum position of the ForceObject, then the force is acting on the positive side of the MovableObject, where the exact position of the acting force is shown in Equation (6).
 - c. If the minimum position of the Movable Object is greater than the minimum position of the ForceObject, and the maximum position of the ForceObject is greater than the minimum position of the MovableObject, then the force is acting on the negative side of the MovableObject, where the exact position of the acting force is shown in Equation (7).
5. Calculate the torque from the force using the calculated distance and the angle of the MovableObject in that dimension using Equation (1) where the radius, r , is set to the calculated distance, p :

$$Max_{movable} = \begin{bmatrix} x_{movable} + \frac{length_{movable}}{2} \sin(\phi) \\ y_{movable} + \frac{width_{movable}}{2} \sin(\theta) \\ z_{movable} + \frac{height_{movable}}{2} \sin(\psi) \end{bmatrix} \quad (2)$$

$$Min_{movable} = \begin{bmatrix} x_{movable} - \frac{length_{movable}}{2} \sin(\phi) \\ y_{movable} - \frac{width_{movable}}{2} \sin(\theta) \\ z_{movable} - \frac{height_{movable}}{2} \sin(\psi) \end{bmatrix} \quad (3)$$

$$Max_{force} = \begin{bmatrix} x_{force} + \frac{length_{force}}{2} \sin(\phi) \\ y_{force} + \frac{width_{force}}{2} \sin(\theta) \\ z_{force} + \frac{height_{force}}{2} \sin(\psi) \end{bmatrix} \quad (4)$$

$$Min_{force} = \begin{bmatrix} x_{force} - \frac{length_{force}}{2} \sin(\phi) \\ y_{force} - \frac{width_{force}}{2} \sin(\theta) \\ z_{force} - \frac{height_{force}}{2} \sin(\psi) \end{bmatrix} \quad (5)$$

$$p = \frac{s}{2} - (Max_{movable} - Min_{force}) \quad (6)$$

$$p = -\frac{s}{2} - (Max_{force} - Min_{movable}) \quad (7)$$

$$\text{where } s = \begin{bmatrix} length_{movable} \\ width_{movable} \\ height_{movable} \end{bmatrix} \quad (8)$$

Once the forces and torques are calculated and added to the set of forces and torques applied to the MovableObject, the environment then calculates the linear and angular positions, velocities, and accelerations from the forces and torques.

3.3. Quadrotor

The quadrotor class represents an autonomous quadrotor that receives inputs from the Network Socket class to define its next desired position while taking in its surroundings. It then determines where it can and cannot go and takes action based on the attitude control, altitude control, and the obstacle avoidance methods.

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

1. Update the calculated position of the quadrotor from the IMU's accelerometer and gyroscope.
2. If in a process of takeoff or landing update, the desired position is based on the requirements set by the takeoff and landing procedure.
3. Update the commanded position, velocity, and acceleration from the desired position, velocity, and acceleration. The desired position, velocity, and acceleration are usually given by the HiveInstance through the network socket or by the altitude controller.
4. Recalculate the commanded position based on the collision detection method.
5. Set the desired motor speeds with the attitude controller.
6. Send the desired motor speeds to the motor controller.

The Environment class then updates the quadrotor based on the motor controller's speeds using the Quadrotor System Block that contains the mathematical model of the quadrotor.

3.4. Hub

The communication layer to the Hub is performed with the NetworkSocket. Shown in Figure 3, the NetworkSocket operates in parallel to connect to the hub via the NetworkConnection object. The NetworkConnection operates as its own network entity, mimicking the process of opening a connection to a remote base station. This is

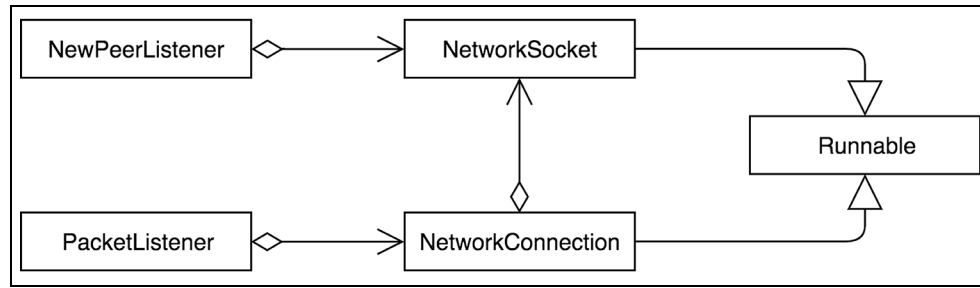


Figure 3. Network class diagram.

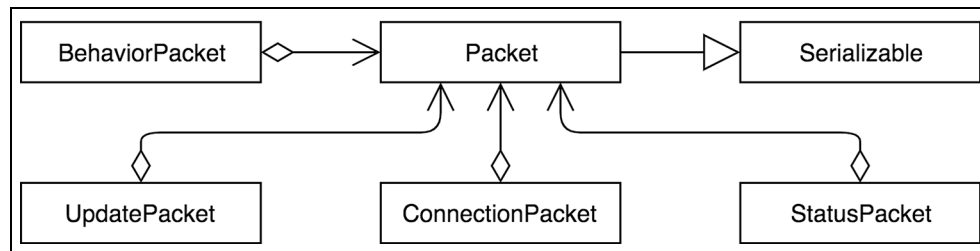


Figure 4. Packet class diagram.

simulated by opening a local port on the machine where the quadrotor component of the simulator is operating. The quadrotor components running in the simulation are treated as a network group, where the group is assigned a starting port. Each quadrotor of the group is assigned a port from its respective network group and the order in which it was added to the group, where $port = base + i$. Here, $port$ is the port number of the NetworkSocket, $base$ is the starting port of the network group the quadcopter is assigned to, and i is the assigned order of the quadcopter in the network group. In this way, the simulation can show the effects in a real environment while minimizing hardware requirements.

For data transmission to take place, a Packet object, illustrated in Figure 4, was created so it could extend specialized data types, such as the quadrotors' spatial and rotational positions. There are four subclasses of Packet, namely ConnectionPacket, UpdatePacket, StatusPacket, and BehaviorPacket. The ConnectionPacket is the first packet sent by a quadrotor or peer, which transmits the name of the peer as well as the current Position. The UpdatePacket contains the current Position, Velocity, and Acceleration of the quadrotor. It lets the quadrotor notify the Hub of its current whereabouts and enables the Hub to provide the quadrotor's next position. The BehaviorPacket allows the quadrotor 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 quadrotor can alter its own behavior state. The

StatusPacket contains an enumerated value that describes the current status of the quadrotor. There are two statuses for a quadrotor, a READY and a PAUSE status. A READY status is when the quadrotor is prepared to get an update from the Hub after connecting to the network. A PAUSE status is when the quadrotor has disconnected from the network group or halted operation. This may occur because it cannot move to a new location or must suspend its operation because of a simulated hardware failure.

To read and write a fully constructed Java object to the network, object streams were used as opposed to traditional primitive data streams to reduce the complexity of the network layer. This ensured that no object caching occurred. If the same object type was sent with updated data then caching might resend old data.

An event-driven network scheme handles incoming data. Each NetworkConnection is run in its own thread, and communication can happen in near real-time so each packet can be handled and responded to immediately on arrival. This is done through the PacketListener interface that allows classes to have a method define how to handle each type of incoming network traffic. Due to the concurrent nature of networking environments, thread synchronization was necessary to ensure network-updated variables were handled safely in the Hub.

The Hub operates in parallel with several threads to distribute the network update process. As shown in Figure 5, the Hub represents a collection of HiveGroups acting as a

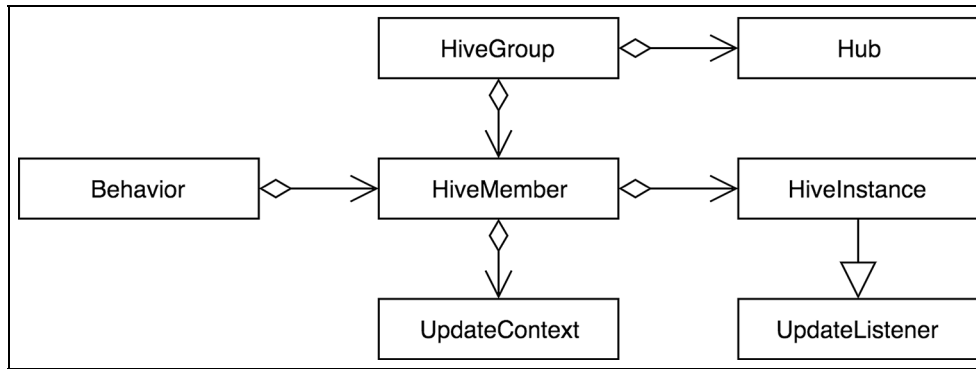


Figure 5. Hub class diagram.

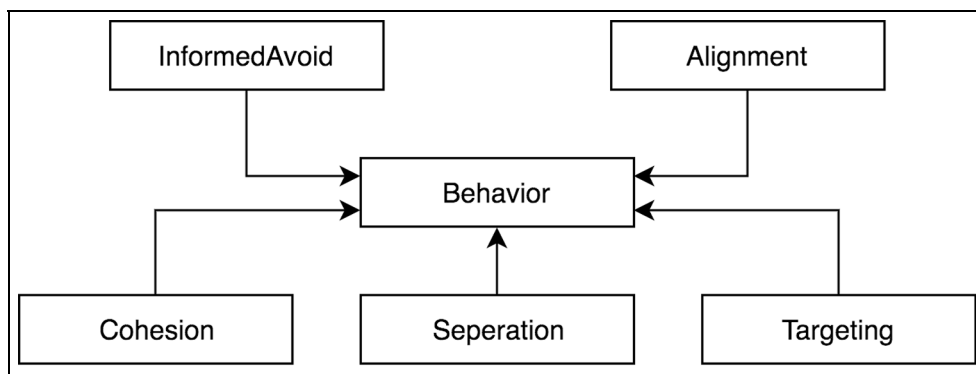


Figure 6. Hub class diagram.

network group of quadrotors: the HiveMembers. Each HiveMember creates a reference of itself, HiveInstance, to synchronize with its other HiveMembers and the Hub. This ensures that each quadrotor is updated at the same time interval, controlled by a global timer in the Hub class, to notify their referenced HiveMembers on their updated location. When the update event is triggered, each HiveInstance requests Position, Velocity, and Acceleration information from its respective quadrotor so Behaviors have data to apply to the new commanded location.

Behaviors offer precise control over how each individual quadrotor acts at any given moment. Shown in Figure 6, a multi-behavior system builds the actions of the hub. The Hub keeps track of a quadrotor's applied Behaviors. To allow for multiple behaviors to modify the state of the quadrotor, an UpdateContext object was created on the HiveMember. It contains a modifiable Velocity for each behavior to define how the quadrotor should adjust its position. The UpdateContext object also stores a set of neighbor quadrotors, determined by Equation (9). Here, n denotes the neighbor and i denotes the current quadrotor; x , y , and z represent the neighbor's and quadrotor's three-dimensional position; R is the maximum radius the neighbor can have from the quadrotor considered¹⁷:

$$\sqrt{(x_n - x_i)^2 - (y_n - y_i)^2 - (z_n - z_i)^2} < R \quad (9)$$

The Behaviors shown in Figure 6 are described as flocking methods in Section 8. Separation, Alignment, Cohesion, Targeting, and InformedAvoid Behaviors are applied individually to each HiveMember.

3.5. Quadrotor mathematical model

The quadrotor is a rigid frame with its center of gravity at its center. The control points four motors equidistant from each other in a square-like format. When placed on the frame, the adjacent motor rotates opposite to the other and both sets form a “+” shape, as shown in Figure 7.

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

In relation to the environment, the quadrotor will be able to move linearly and angularly in three-dimensional space. This means that it will need to move in a linear direction through the simulated environment and at an angular direction relative to its linear position. The linear directions are defined by our Cartesian coordinate system

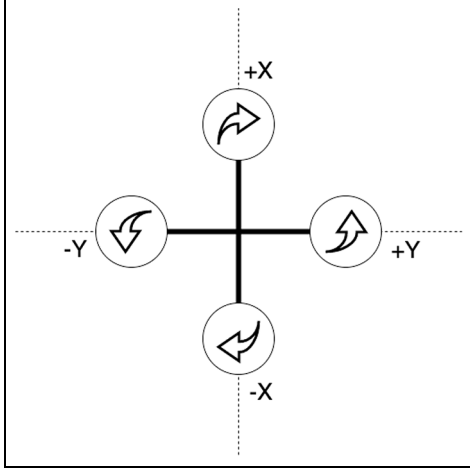


Figure 7. Rotational quadrotor movement.

in which we will have three axes named x , y , and z . The x -axis defines positions of length, the y -axis is the width, and the z -axis defines positions of height. The angular directions are defined as follows: ϕ is the roll angle around the x -axis, θ is the pitch angle around the y -axis, and ψ is the yaw angle around the z -axis. Thus, our coordinate system, q , is then defined as shown in Equation (10), where η is the linear coordinate system and ξ is the angular coordinate system:

$$q = \begin{bmatrix} \eta \\ \xi \end{bmatrix}, \eta = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \xi = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \quad (10)$$

3.5.1. Forces and moments. The quadrotor has several aerodynamic forces acting on itself. These forces are produced via a spinning rotor blade. Fay¹⁸ discussed aerodynamic forces in the Mesicopter project. He derived several forces for quadrotors from the aerodynamic effects caused by helicopter and blade movement through flight. Bouabdallah¹⁹ implemented four forces in the design and control of autonomous quadrotors. In this research, we implement the thrust force, hub force, drag moment or total torque, rolling moment, and equations of motion. We base our implementation on Fay¹⁸ and Bouabdallah.¹⁹

3.5.2. Thrust force. We calculate the thrust force by integrating the vertical forces applied to all blade elements, as shown in Equations (10)–(13). In Equation (11), T is the resulting thrust force, C_T is the coefficient of the thrust force, shown in Equation (12), ρ is the air density of the surrounding air, A is the disk area of the propeller, Ω is the angular rate of the propeller, and R_{rad} is the rotor radius. In Equation (12), θ_{tw} is the twist pitch of the propeller, θ_0 is the pitch of incidence of the propeller, λ is the inflow ratio, shown in Equation (13), μ is the rotor advance ratio,

shown in Equation (14), σ is the solidity ratio of the propeller, and a is the lift slope of the propeller:

$$T = C_T \rho A (\Omega R_{rad})^2 \quad (11)$$

$$\frac{C_T}{\sigma a} = \left(\frac{1}{6} + \frac{1}{4} \mu^2 \right) \theta_0 - (1 - \mu^2) \frac{\theta_{tw}}{8} - \frac{1}{4} \lambda \quad (12)$$

In Equation (13), v_1 is the inflow velocity, \dot{z} is the linear velocity in the z direction, S_p in this case is the speed of the propeller, and R is the radius of the propeller. In relation to the inflow velocity, W is the weight of the quadrotor in relation to the thrust, where d is the diameter of the propeller, P_c is the power constant, and f is the power factor:

$$\lambda = \frac{v_1 - \dot{z}}{S_p R} \quad (13)$$

where:

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

$$W = \left(\frac{\pi}{2} d^2 A (P_c S_p^f)\right)^{\frac{1}{3}}$$

In Equation (14), 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:

$$\mu = \frac{V}{\Omega R} \quad (14)$$

where:

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

3.5.2.1. Hub force. The hub force is calculated by integrating all of the horizontal forces applied to all blade elements, shown in Equation (15). Here, H is the total hub force acting on the system, C_H is the coefficient of the hub force, shown in Equation (16), and $\overline{C_d}$ is the drag coefficient of the blades at 70% radial station:

$$H = C_H \rho A (\Omega R_{rad})^2 \quad (15)$$

$$\frac{C_H}{\sigma a} = \frac{1}{4a} \mu \overline{C_d} - \frac{1}{4} \lambda \mu \left(\theta_0 + \frac{\theta_{tw}}{2} \right) \quad (16)$$

3.5.2.2. Drag moment. The drag moment, or total torque, of the quadrotor as all of the aerodynamic forces around the center of the rotor is shown in Equation (17). This determines the amount of power required to spin the motor. Here, Q is the total torque or drag moment and C_Q is the drag moment coefficient, described by Equation (18):

$$Q = C_Q \rho A (\Omega R_{rad})^2 R_{rad} \quad (17)$$

$$\frac{C_Q}{\sigma a} = \frac{1}{8a} (1 + \mu^2) \overline{C_d} + \lambda \left(\frac{1}{6} \theta_{rw} - \frac{1}{8} \theta_{rw} - \frac{1}{4} \lambda \right) \quad (18)$$

3.5.2.3. Rolling moment. The rolling moment occurs in forward flight when the “advancing blade is producing more lift than the retreating blade,” as shown in Equation (19). Here, R_m is the rolling moment, shown in in Equation (20), and C_{R_m} is the rolling moment coefficient:

$$R_m = C_{R_m} \rho A (\Omega R_{rad})^2 R_{rad} \quad (19)$$

$$\frac{C_{R_m}}{\sigma a} = -\mu \left(\frac{1}{6} \theta_{rw} - \frac{1}{8} \theta_{rw} - \frac{1}{4} \lambda \right) \quad (20)$$

3.5.3. 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 described by Kemper²⁰ is shown in Equation (21), 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 (22), 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, I , is a combination of I_r and I_c , and is shown in Equation (23):

$$I_r = \begin{bmatrix} \frac{1}{24} m_r l^2 & 0 & 0 \\ 0 & \frac{1}{24} m_r l^2 & 0 \\ 0 & 0 & \frac{1}{12} m_r l^2 \end{bmatrix} \quad (21)$$

$$I_c = \begin{bmatrix} \frac{1}{12} m_c (w^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12} m_c (l^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{12} m_c (w^2 + l^2) \end{bmatrix} \quad (22)$$

$$I = I_r + I_c \quad (23)$$

3.5.4. Equations of motion. The equations of motion are defined by the equations for acceleration given by Equations (24)–(29). Here, A_c is the surface area of the quadrotor, C_x and C_y are the friction constants in the x and y directions, respectively, 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:

$$\ddot{x} = \frac{(\sin(\psi) \sin(\phi) + \cos(\psi) \sin(\theta) \cos(\phi)) \sum_{i=1}^4 T_i - \sum_{i=1}^4 H_i - \frac{1}{2} C_x A_c \rho \dot{x}}{m} \quad (24)$$

$$\ddot{y} = \frac{(-\cos(\psi) \sin(\phi) + \sin(\psi) \sin(\theta) \cos(\phi)) \sum_{i=1}^4 T_i - \sum_{i=1}^4 H_i - \frac{1}{2} C_y A_c \rho \dot{y}}{m} \quad (25)$$

$$\ddot{z} = \frac{mg - \sum_{i=1}^4 T_i}{m} \quad (26)$$

$$\ddot{\phi} = \frac{\dot{\theta} \dot{\psi} (I_{yy} - I_{zz}) + J_r \dot{\theta} \Omega_r + l(-T_2 + T_4) - h \left(\sum_{i=1}^4 H_{yi} \right) + \left(\sum_{i=1}^4 (-1)^{i+1} R_{mxi} \right)}{m} \quad (27)$$

$$\ddot{\theta} = \frac{\dot{\phi} \dot{\psi} (I_{zz} - I_{xx}) - J_r \dot{\phi} \Omega_r + l(T_1 - T_3) + h \left(\sum_{i=1}^4 H_{yi} \right) + \left(\sum_{i=1}^4 (-1)^{i+1} R_{myi} \right)}{m} \quad (28)$$

$$\ddot{\psi} = \frac{\dot{\theta} \dot{\psi} (I_{xx} - I_{yy}) + J_r \Omega_r + \left(\sum_{i=1}^4 (-1)^{i+1} Q_i \right) + l(H_{x2} - H_{x4}) + l(-H_{y1} + H_{y3})}{m} \quad (29)$$

3.6. Sensors

Three main sensors are used to get spatial and environmental information the IMU, pressure sensor, and ultrasonic sensors. Each sensor has a refresh rate, when it collects information from its environment, and a noise level, with random maximum deviation from the true value. The IMU is based on a MinIMU-9 v3 with a 100 Hz cycle and a noise level of 0.011.²¹ The pressure sensor is based on a BMP180 with a 200 Hz cycle and a maximum noise level of 0.06.²² The ultrasonic sensors are designed around the Navo ultrasonic sensor with a 20 Hz cycle and a maximum noise level of 0.005.²³

3.6.1. Inertial Measurement Unit. The IMU contains a three-axis gyroscope, accelerometer, and magnetometer. We will only be using the gyroscope and accelerometer data from the IMU. In our simulator, the gyroscope provides the current angular position to determine the angular velocity and angular acceleration. The angular acceleration, Ω , is described in Equation (30). Here, λ is the current angular position, λ_0 is the previous angular position, ω_0 is the previous angular velocity, and t is the time that has passed since the last cycle:

$$\Omega = \begin{bmatrix} \frac{\lambda_{\phi} - \lambda_{0\phi}}{t} - \omega_{0\phi} \\ \frac{\lambda_{\theta} - \lambda_{0\theta}}{t} - \omega_{0\theta} \\ \frac{\lambda_{\psi} - \lambda_{0\psi}}{t} - \omega_{0\psi} \end{bmatrix} \quad (30)$$

The accelerometer provides the linear acceleration relative to its current linear and angular position, shown in Equation (31). The acceleration relative to the current linear position, a , is shown in Equation (32). Here, p is the current linear position, p_0 is the previous linear position, and v_0 is the previous linear velocity; ϕ , θ , and ψ are the respective current pitch, roll, and yaw angles of the accelerometer relative to its environment:

$$ar = \begin{bmatrix} a_x \cdot \cos(\phi) + a_y \cdot \sin(\phi) + a_z \cdot \sin(\phi) \\ a_y \cdot \cos(\theta) + a_x \cdot \sin(\theta) + a_z \cdot \sin(\theta) \\ a_z \cdot \cos(\psi) + a_x \cdot \sin(\psi) + a_y \cdot \sin(\psi) \end{bmatrix} \quad (31)$$

$$a = \begin{bmatrix} \frac{px - p_{0x}}{t} - v_{0x} \\ \frac{py - p_{0y}}{t} - v_{0y} \\ \frac{pz - p_{0z}}{t} - v_{0z} \end{bmatrix} \quad (32)$$

3.6.2. Pressure sensor. The pressure sensor provides the altitude from the current atmospheric pressure of the sensor in the environment. Equation (33) determines the height, h , of the sensor by its atmospheric pressure, P_h . Here, P_o is the atmospheric pressure at sea level, m is the mass of one molecule, g is the acceleration of gravity, L is the temperature lapse rate, R is the universal gas constant, and T is the average temperature at sea level:

$$h = \frac{T \cdot \left(\frac{P_h}{P_o} \right)^{\frac{1}{\frac{g \cdot m}{R \cdot L}}}}{L} \quad (33)$$

3.6.3. Ultrasonic sensor. The ultrasonic sensor emits a ping and calculates the distance by the amount of time it takes for that ping to return. Each sensor has a horizontal and vertical range with which it can receive the ping return, along with a distance range. If the total ping return range of the quadrotor is 30 degrees and the ultrasonic sensor was placed on a circle at the 45-degree position then the simulator would allow the ultrasonic sensor to receive a ping from the 30–60-degree position relative to the circle.

In our simulator, we position the ultrasonic sensor on a horizontal circular platform around the quadrotor shape. Its angular position, ψ , on the frame is dependent on the number of ultrasonic sensors, shown in Equation (34). Here, c is the total amount of ultrasonic sensors and i is the index of the ultrasonic sensors. The ultrasonic sensor is pointed perpendicular to the horizontal platforms face:

$$\psi = \frac{2\pi}{c}i + \frac{2\pi}{c \cdot 2} \quad (34)$$

From ψ , the x and y positions of the ultrasonic sensor relative to the quadrotor body are determined, where $x = l \cdot \sin(\psi)$ and $y = l \cdot \cos(\psi)$. Here, l is the horizontal

Algorithm 1. Pseudocode for setting all vertical and horizontal rays for the ultrasonic sensor.

```

SET start = - PI - (ANGLE / 2.0)
SET increment = ANGLE / RAYCAST_COUNT
FOR X to RAYCAST_COUNT
  SET lattIncrement = start + (X * increment)
  FOR Y to RAYCAST_COUNT
    SET heightIncrement = - (ANGLE / 2.0) + (Y * increment)
    SET endpoint to CALL createEndPoint with
      lattIncrement,
      heightIncrement
    CALL rotateEndPoint with
      endpoint,
      ultrasonicAngularPosition,
      ultrasonicLinearPosition
    CALL addRay with endpoint
  END FOR
END FOR

```

Algorithm 2. Pseudocode for creating an endpoint for the ray protruding from the ultrasonic sensor.

```

DEFINE createEndPoint with PSI, EPSILON
  SET X = (DETECT_DIST + DIST_CENTER) * sin(PSI);
  SET Y = (DETECT_DIST + DIST_CENTER) * cos(PSI);
  SET Z = (DETECT_DIST + DIST_CENTER) * sin(EPSILON);
  return [X, Y, Z];
END DEFINE

```

length of the position of the ultrasonic sensor relative to the center of the quadrotor.

The ray casting method proposed by Möller and Trumbore²⁴ is used to calculate the distance by the simulated pings received via the ultrasonic sensor. The initial position of the ultrasonic sensor describes the origin of the rays. Algorithm 1 determines the vertical and horizontal creation of the rays that fill in the ping return range of the ultrasonic sensor. Algorithm 2 creates the endpoint of the ray from its vertical and horizontal range. Algorithm 3 rotates the endpoint relative to the ultrasonic sensor's position in the environment.

DETECTION_DIST is the maximum distance at which the ultrasonic sensor can detect an object. DIST_CENTER is the distance from the center of the Quadrotor to the ultrasonic sensor.

In Algorithm 3, E_X is the x position of the end of the ray, E_Y is the y position of the end of the ray, E_Z is the z position of the end of the ray, U_PHI is the ϕ of the ultrasonic sensor, U_THETA is the θ of the ultrasonic sensor, U_PSI is the ψ of the ultrasonic sensor, Q_X is the x position of the quadrotor, Q_Y is the y position of the quadrotor, and Q_Z is the z position of the quadrotor.

4. Attitude control

Attitude control enables the quadrotor to control its angular position to move itself to the desired linear position. There

Algorithm 3. Pseudocode for rotating the endpoint of a ray protruding from the ultrasonic sensor.

```

DEFINE rotateEndPoint with E_X, E_Y, E_Z, U_PHI, U_THETA, U_PSI, Q_X, Q_Y, Q_Z
  // rotate around the z axis
  SET xTemp = E_X * cos(U_PSI) - E_Y * sin(U_PSI);
  SET yTemp = E_X * sin(U_PSI) + E_Y * cos(U_PSI);
  SET zTemp = E_Z;
  E_X = xTemp;
  E_Y = yTemp;
  E_Z = zTemp;
  // rotate around the y axis
  xTemp = E_X * cos(-U_THETA) + E_Z * sin(-U_THETA);
  yTemp = E_Y;
  zTemp = -E_X * sin(-U_THETA) + E_Z * cos(-U_THETA);
  E_X = xTemp;
  E_Y = yTemp;
  E_Z = zTemp;
  // // rotate around the x axis
  xTemp = E_X;
  yTemp = E_Y * cos(-U_PHI) - E_Z * sin(-U_PHI);
  zTemp = E_Y * sin(-U_PHI) + E_Z * cos(-U_PHI);
  E_X = xTemp + Q_X;
  E_Y = yTemp + Q_Y;
  E_Z = zTemp + Q_Z;
  return endPoint;
END DEFINE

```

has been much research to determine the best method of attitude control of the quadrotor. The choice of the attitude control method is key to building a proper controlling mechanism tailored specifically to the quadrotor's design, as each attitude control method has its advantages and disadvantages.

Kemper²⁰ concludes that the best control method is a combination of the PID and PD controllers as its attitude control algorithm. Hoffman et al.²⁵ also utilized PID and PD controllers, but used the PID controller for low-velocity flight and the PD controller for high-velocity flight. Dikmen et al.²⁶ compared several attitude controllers and decided that the sliding mode technique had the best performance, especially on higher initial conditions.

The implementation of Kemper, Hoffman, and Dikmen's attitude controllers to interface with the simulator are covered by Micklisch et al.¹⁶

5. Altitude control

Altitude control allows the quadrotor to smoothly takeoff and land. The determined height of the quadrotor in the environment is used to synchronously change the motor speeds of the quadrotor to increase or decrease the height of the quadrotor in the environment. Our altitude control algorithm is based on Gageik et al.'s technique²⁷ to calculate the height over the ground. It combines information from main and reference sensors with weights for each sensor. Unlike Gageik's implementation, we do not have PID controllers to control the motors. Our algorithm uses

two sensors that change their assigned weights to be a reference or main sensor from their signal outputs. In this section, we present algorithms for landing and takeoff that use the altitude controller of the quadrotor to control the height.

5.1. Altitude control

To decide if a sensor should be a main or reference sensor, the weights of the sensors are calculated. The sensor with the highest weight is designated as the main sensor. The weight is calculated when the output of the sensor goes below a certain threshold. The pressure sensor threshold is at an altitude of 15 meters. The ultrasonic sensor threshold is at a detected ground distance of 5 meters. This gives priority to the ultrasonic sensor, as the landing area is closer than the pressure sensor perceives it to be. This enables the quadrotor to start its landing and takeoff phase at any height safely.

The weights for the sensors are shown in Equations (35)–(37). In the equations, 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. It is used to give the ultrasonic sensor greater weight when it detects a value at a relatively high altitude. In the simulator, we set the value R_u to 1:

$$Alt = W_u O_u + W_p O_p \quad (35)$$

$$W_u = \frac{\sqrt{M_u^2 - (O_u + R_u)^2}}{M_u} \quad (36)$$

$$W_p = 1 - W_u \quad (37)$$

The ultrasonic sensor priority helps get the relevant height of the system for landing or takeoff. In the case of the altitude of the quadrotor, the pressure is only relevant when the ultrasonic sensor's output is too high so that the quadrotor has an understanding of the relative height in the environment. Only when the ultrasonic sensor outputs lower values is it more important, as the quadrotor is closer to a landing sight.

5.2. Takeoff method

The aim of the takeoff method is to have the quadrotor safely lift off and hover at 5 meters until further instructions are given. The takeoff method is performed in three states of operation.

1. If the quadrotor starts from the bottom of the floor, it slowly spools up its motors, and sets the desired height to 10 meters above its current position. This causes the motors to give a large instantaneous thrust to lift off the floor.
2. If the quadrotor is off the ground but is below 5 meters of its original takeoff position then it sets the desired height to 5 meters above its original takeoff position. This causes the motor output to decrease and reach a steady rate of ascent.
3. If the quadrotor is at a position of 5 meters, it keeps its desired position until further instructions are given.

5.3. Landing method

Similar to the takeoff, the landing is performed in phases based on the height at which the quadrotor is currently located. Unlike Gageik's et al. method,²⁷ our landing method allows the quadrotor to land at any height, rather than only being able to land at a height of less than 60 cm. We use a buffer time to decrease the rate of descent to allow for a faster rate of descent at higher altitudes and a smoother descent at lower altitudes. Landing is performed in four states of operation.

1. If the quadrotor is 1 meter above its landing position, it sets its desired position to 4 meters below its current height until it reaches 1 meter in height above a landing position. This causes the quadrotor to descend at a steady rate.

2. If the quadrotor is in between 1 and 0.5 meters in height above its landing position, it sets its desired position to 2 meters below its current height until it reaches a height of 0.5 meters. This reduces the rate of descent.
3. If the quadrotor is between 0.5 and 0.1 meters in height above the landing position, it sets its desired position to 0.1 meters below its current height until it reaches a height of 0.1 meters. This flares the motors to quickly reduce the rate of descent.
4. If the quadrotor is below 0.1 meters in height above the landing position, it turns off all motors to prevent any movement as it securely lands.

6. Obstacle detection

Several researchers have investigated the problem of obstacle detection, whether outdoor or indoor. Prashanth et al.²⁸ used a visual-based optical flow technique for obstacle avoidance. Gageik et al.²⁹ and Wagster et al.³⁰ used an assortment of ultrasonic sensors to detect and avoid obstacles. Our research focuses 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 Prashanth et al.²⁸ and do 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 present two proposals for object detection based on Gageik et al.²⁹ and Wagster et al.³⁰

6.1. Obstacle detection method I

We base this method on the obstacle detection method by Gageik et al.,²⁹ where they used 12 ultrasonic sensors in a circle parallel to the horizontal portion of the quadrotor frame:

$$State = \begin{cases} safe & \text{where } a + b < m \\ close & \text{where } a + b > m \\ dangerous & \text{where } a < m \end{cases} \quad (38)$$

As in Gageik et al.,²⁹ our obstacle detection method uses data from the ultrasonic sensors and creates three states: *safe*, *close*, and *dangerous*, depending on the distances that the ultrasonic sensors send, as shown in Equation (38). Here, m is the distance received from the ultrasonic sensors, a is a close range constant, and b is a long range constant.

From these states, the quadrotor acts as a state machine, where the state describes the movement of the quadrotor. If the state is in a *safe* state, it is free to move. If it is in a *close* state then the machine has limited movement. The closer the quadrotor state is to the *close* state the more

Algorithm 4. Pseudocode for obstacle detection and avoidance.

```

DEFINE calculateSuggestedPosition with lowestDistance
  SET position to [0, 0, 0]
  FOR DIM from [X, Y, Z]
    IF POINT_A > lowestDistance THEN
      SET diff to CALL calculateDifference WITH DIM
      SET position[DIM] to -1 * signum(diff)
      * (lowestDistance + diff) + quadPos[DIM]
    ELSE IF POINT_A + POINT_B > lowestDistance THEN
      SET diff to CALL calculateDifference WITH DIM
      SET position[DIM] to RED_MULTI * diff
      + quadPos[DIM]
    ELSE IF POINT_A + POINT_B < lowestDistance THEN
      SET position[DIM] to commandedQuadPos[DIM]
    END IF
  END FOR
  return position
END DEFINE

```

movement is limited. If the state is in a *dangerous* state, then all movement is limited to prevent any further approaches to the obstacle that triggered the *dangerous* state.

We improved on Gageik et al.'s²⁹ obstacle detection method when an obstacle obstructs the quadrotor's desired location. In Gageik et al.'s implementation, the quadrotor stops and stays in its position. In our implementation, 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 Algorithms 4 and 5. DIM represents the Cartesian coordinates x , y , and z ; quadPos represents the quadrotor position, while commanded QuadPos represents the attitude controls desired quadrotor position. POINT_A and POINT_B represent a and b from Equation (38), respectively. RED_MULTI is the reduction multiplier, used to slow down the quadrotor.

When the state is *dangerous* it simply reverses the direction of the current commanded position from the current position of the quadrotor. This allows for a slow and smooth movement away from the object. When the state is *close* then the commanded position is changed by multiplying the difference between the commanded and current position with the reduction multiplier so the quadrotor slowly moves toward the obstacle. If the state is *safe* then nothing is done.

6.2. Obstacle detection method 2

This method is based on Wagster et al.'s³⁰ obstacle detection method. It uses a three-level avoidance system to

Algorithm 5. Pseudocode for calculating the difference between the commanded and the current position.

```

DEFINE calculateDifference with DIM
  SET commanded to commandedQuadPos[DIM]
  SET current to quadPos[DIM];
  return commanded - current;
END DEFINE

```

reduce potential collisions while retrieving environmental information even if the goal was obstructed.

The first level uses a forward facing sensor to determine the 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. The quadrotor will cease all motion to prevent a collision. The algorithm increases the pitch angle to slow down the quadrotor.

The second level uses the left- and right-hand sensors of the vehicle to center the quadrotor between two obstacles with 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 determines a path around the obstacle to reach its goal. For Wagster et al.,³⁰ 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 Wagster et al.'s³⁰ obstacle detection method is to utilize an unlimited number of ultrasonic sensors, determine the front, left-, and right-hand sensors from the heading of the quadcopter and the commanded position, and control the movement of the quadrotor with the commanded positions to allow us to utilize any attitude controller. We focused on levels 1 and 2, as the flight path based on prior knowledge is not an obstacle detection problem but a flight path problem.

The pseudocode for determining the front, left-, and right-hand sensors is shown in Algorithms 6 and 7. NO_VALUE represent a null or empty value. MAX_ANGLE is the greatest angle in a circle, 360 degrees.

The implementation of the first level is shown in Algorithm 8, where the obstacle detection method is used to determine the distance of an obstacle in front of itself. Here we define front by the direction of the commanded position. We get the front facing sensor using our getUltrasonicIndex method, shown in Algorithm 6. From this, we determine if the front facing sensor value is below the *MINIMUM_PHASE_ONE* value. If it is, the commanded position is set to be the current position, thereby avoiding potential collisions.

Algorithm 6. Pseudocode for getting the sensor index, given the commanded position.

```

DEFINE getUltrasonicIndex with commandedPosition
  SET lowest to NO_VALUE;
  FOR X to ULTRASONIC_SENSOR_COUNT
    SET currentDiff CALL getSensorPosDifference with X, commandedPosition
    SET lowestDiff CALL getSensorPosDifference with lowest, commandedPosition
    IF currentDiff < lowestDiff THEN
      SET lowest to X
    END IF
  END FOR
  return lowest
END DEFINE

```

Algorithm 7. Pseudocode for getting the difference between the angular position of the sensor compared to the angular position of the commanded position.

```

DEFINE getSensorPosDifference
  with sensorPosition, commandedPos, quadPos
  IF index == NO_VALUE THEN
    return MAX_ANGLE;
  END IF
  SET xdiff to CALL sensorPosition[X]- quadPos[X]
  SET ydiff to sensorPosition[Y]- quadPos[Y]
  SET xdiffDest to commandedPos[X] - quadPos.getPoint[X]
  SET ydiffDest to commandedPos[Y] - quadPos.getPoint[Y]
  SET degree to atan2(xdiff, ydiff)
  SET degreeDest to tan2(xdiffDest, ydiffDest))
  IF degree < 0 THEN
    degree = degree + MAX_ANGLE
  END IF
  IF degreeDest < 0 THEN
    degreeDest = degreeDest + MAX_ANGLE
  END IF
  return abs(degree - degreeDest) % MAX_ANGLE
END DEFINE

```

The implementation of the second level is shown in Algorithm 9, where the obstacle detection method places the quadcopter in the center of two obstacles facing each other. In this level, the commanded position is rotated 90 degrees clockwise and counter clockwise to determine the left- and right-hand ultrasonic sensors and gather their values. It then checks for two requirements: if the values of the sensors are equal; and if the left- and right-hand sensor outputs are below the *MINIMUM_PHASE_TWO* value. *MINIMUM_PHASE_TWO* is the maximum distance that the ultrasonic sensor can detect. If the two requirements hold true then the new commanded position is calculated to be the center of the two obstacles. It calculates the necessary change of the quadcopter by the difference in the output of the right- and left-hand sensors and rotates that

Algorithm 8. Pseudocode for first level of obstacle detection method 2.

```

DEFINE phaseOne with position, commandedPos, quadPos
  SET frontSensorIndex to CALL getUltrasonicIndex with
    commandedPos
  IF frontSensorIndex == NO_VALUE THEN
    return position
  END IF
  SET frontValue to CALL getSensorValue with frontSensorIndex
  IF frontValue < MINIMUM_PHASE_ONE THEN
    SET position[X] to quadPos[X]
    SET position[Y] to quadPos[Y]
    SET position[Z] to quadPos[Z]
  ENDIF
  return position;
END DEFINE

```

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-hand sides, depending on the direction the quadrotor is moving in.

7. Relative positioning

The positioning of the quadrotor relative to its environment is determined by the quadrotor once the perceived position is transferred to the overall movement of the quadrotor over time. Positioning methods outside of the object's perceived movement have met with much success. Fernández et al.³¹ proposed an external camera array to locally position a robot. The experiments use six cameras positioned around a structured environment to determine the absolute position with infrared beacons. When two cameras detect the infrared beacons, the maximum position error is 5 centimeters. The maximum position error is 2 centimeters when all cameras detect the beacons.

Algorithm 9. Pseudocode for second level of obstacle detection method 2.

```

DEFINE phaseTwo with position, commandedPos, quadPos
  SET positionLeft to [X, Y]
  SET positionLeft[X] to (cos(90.0) * commandedPos[X]) - (sin(90.0) * commandedPos[Y])
  SET positionLeft[Y] to (sin(90.0) * commandedPos[X]) + (cos(90.0) * commandedPos[Y])
  SET positionRight to [X, Y]
  SET positionRight[X] to (cos(-90.0) * commandedPos[X]) - (sin(-90.0) * commandedPos[Y])
  SET positionRight[Y] to (sin(-90.0) * commandedPos[X]) + (cos(-90.0) * commandedPos[Y])
  SET leftValue to CALL getUltrasonicLeftSensorValue
  SET rightValue to CALL getUltrasonicRightSensorValue
  IF leftValue != rightValue
    && leftValue < MINIMUM_PHASE_TWO
    && rightValue < MINIMUM_PHASE_TWO
  THEN
    SET difference to rightValue - leftValue
    SET xPosD to abs(difference) * signum(commandedPos[X])
    SET initAngle to getUltrasonicFrontAngularPosition[PSI]
  IF difference < PHASE_TWO_PIVOT THEN
    SET diffAngle to -90.0
  ELSE
    SET diffAngle to 90.0
  END IF
  SET finalAngle to initAngle + diffAngle
  SET xPos to position[X] + (cos(finalAngle) * xPosD)
  SET yPos to position[Y] + (sin(finalAngle) * xPosD)
  SET position[X] to xPos
  SET position[Y] to yPos
END IF
return p;
END DEFINE

```

Relative positioning methods designed to use an onboard sensor have also been relatively successful. Kim and Kim³² implemented a position estimation scheme with a smartphone's internal IMU to detect a pedestrian's position indoors. In a set of 10 experiments with a gyro orientation estimation, map matching, and particle-filtering scheme, the average position error is 4.19 meters. In our simulator, our positioning method is implemented with an onboard IMU. The initial perceived position, velocity, and acceleration are set to zero.

The relative angular velocity and position of the quadrotor are calculated from the perceived angular velocity, shown in Equations (39) and (40), respectively. Here, ω_r is the relative angular velocity of the IMU in its environment and ξ_r is the relative angular position of the IMU in its environment. The relative angular velocity and position are initially zero:

$$\omega_r = \begin{bmatrix} \omega_{r\phi}(i-1) + \Omega_{p\phi}(i)t \\ \omega_{r\theta}(i-1) + \Omega_{p\theta}(i)t \\ \omega_{r\psi}(i-1) + \Omega_{p\psi}(i)t \end{bmatrix} \quad (39)$$

$$\xi_r = \begin{bmatrix} \xi_{r\phi}(i-1) - \omega_{r\phi}(i)t + \frac{(\Omega_{p\phi}(i))^2}{2} \\ \xi_{r\theta}(i-1) - \omega_{r\theta}(i)t + \frac{(\Omega_{p\theta}(i))^2}{2} \\ \xi_{r\psi}(i-1) - \omega_{r\psi}(i)t + \frac{(\Omega_{p\psi}(i))^2}{2} \end{bmatrix} \quad (40)$$

The relative linear acceleration is calculated by converting the perceived acceleration from the accelerometer to the relative acceleration in the environment, shown in Equations (41)–(43). Here, ϕ , θ , and ψ are the relative angular positions from ξ_r . The relative linear velocity, v_r , and relative linear position, η_r , are attainable through the kinematic conversions shown in Equations (44) and (45), respectively:

$$a_{rx} = \frac{a_{px} - a_{ry} \sin(\phi) - a_{rz} \sin(\phi)}{\cos(\phi)} \quad (41)$$

$$a_{ry} = \frac{\frac{a_{py}}{\cos(\theta)} - \frac{a_{px} \tan(\theta)}{\cos(\phi)} + a_{rz} \tan(\theta)(\tan(\phi) - 1)}{1 - \tan(\phi) \tan(\theta)} \quad (42)$$

$$a_{rz} = \frac{\frac{a_{pz}}{\cos(\psi)} - \frac{a_{px} \tan(\psi)}{\cos(\phi)} + \frac{a_{py}(\tan(\psi)(\tan(\phi)-1))}{\cos(\theta)(1-\tan(\phi)\tan(\theta))} - \frac{a_{px} \tan(\theta)(\tan(\psi)(\tan(\phi)-1))}{\cos(\phi)(1-\tan(\phi)\tan(\theta))}}{1 - \tan(\phi) \tan(\psi) - \frac{(\tan(\theta)(\tan(\phi)-1))(\tan(\psi)(\tan(\phi)-1))}{1-\tan(\phi)\tan(\theta)}} \quad (43)$$

$$v_r = \begin{bmatrix} v_{rx}(i-1) + a_{rx}(i)t \\ v_{ry}(i-1) + a_{ry}(i)t \\ v_{rz}(i-1) + a_{rz}(i)t \end{bmatrix} \quad (44)$$

$$\eta_r = \begin{bmatrix} \eta_{rx}(i-1) - v_{rx}(i)t + \frac{(a_{rx}(i))^2}{2} \\ \eta_{ry}(i-1) - v_{ry}(i)t + \frac{(a_{ry}(i))^2}{2} \\ \eta_{rz}(i-1) - v_{rz}(i)t + \frac{(a_{rz}(i))^2}{2} \end{bmatrix} \quad (45)$$

8. Flocking methods

Reynolds³³ presented a model for the distributed behavior of flocks, while Su et al.¹⁷ based the positioning of the flock on virtual leader methodology by using the center of mass for all agents, as shown in Equation (46). Here, C_N is the average position, the center, of a neighborhood within a flock with N neighbors in the neighborhood:

$$C_N = \frac{1}{N} \sum_{i=1}^N \{x_i, y_i, z_i\} \quad (46)$$

In order to keep the quadrotors from accelerating too quickly, a maximum velocity is applied to each behavior, preventing over-corrections. The mathematical models of the separation, alignment, cohesion, targeting, and informed avoidance behavior of a flock are presented by Micklisch et al.¹⁵

9. Results

The simulation results display the movement of the quadrotor as it flies through its environment. We consider a smooth motion as a positive effect and a jittery motion as a negative effect. Smooth motion is determined by the locally weighted scatterplot smoothing (LOESS) non-parametric regression function in the R language to calculate the standard error from the returned value. The lower the standard error, the smoother the line. We use the value of 1 for the span and 2 for the degree.

9.1. Attitude control

Each attitude control simulation run sets the start position of the quadrotor at (0,0,0) in the x , y , and z coordinates. The desired position of the attitude controller is set at (1,1,1). Micklisch et al.¹⁶ give detailed analysis of Kemper, Hoffman, and Dikmen attitude controller performance.

We calculated the standard error of the three investigated attitude controllers. The results are presented in Table 1. Based on these results, one can conclude that Hoffman's PD controller method was the smoothest, as its average standard error was 4.69290308 in comparison to

Table 1. Standard errors for attitude controllers.

Comparison	Kemper	Hoffman	Dikmen
x	4.384743	1.601325	4.177628
y	3.631218	1.438495	3.571708
z	10.68106	4.881332	7.573814
ϕ	5.21786	6.888175	5.700442
θ	4.917646	9.668774	5.259321
ψ	4.130222	3.679341	11.65922
Overall	5.4937965	4.69290308	6.32369

Table 2. Standard errors for attitude controllers with noise.

Comparison	Kemper w/ noise	Hoffman w/ noise	Dikmen w/ noise
x	3.099703	5.685	4.622426
y	2.152327	4.784024	3.785161
z	10.2911	7.357921	7.995629
ϕ	4.871248	14.01324	11.12085
θ	4.855147	12.34499	10.50764
ψ	10.41253	15.09638	11.88806
Overall	5.94701	9.880259	8.319959

Kemper's average standard error of 5.4937965 and Dikmen's of 6.32369.

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 1 under an ideal scenario.

9.1.1. Summary of comparison of attitude controllers with noise. We calculated the standard error of the three attitude controllers with a noise scenario. The results are presented in Table 2. Based on these results, the attitude controller that has the smoothest motion with noise is Kemper's, with the average standard error being 5.94701. Dikmen's average standard error is 8.319959 and Hoffman's is 9.880259. Kemper's has better performance, as the difference between it and Dikmen's is greater than 2.

9.2. Altitude control

Figure 8 shows how the altitude controller uses the takeoff method, described in Section 5.2. The attitude controller used in this experiment is the Kemper²⁰ attitude controller. In our simulation environment, we positioned a platform at (0,0,0), where all of the quadrotors takeoff and land.

The quadrotor starts on the platform at 1.46 meters, which is the height of the quadrotor with an ultrasonic sensor affixed to its bottom. As shown in Figure 8, the

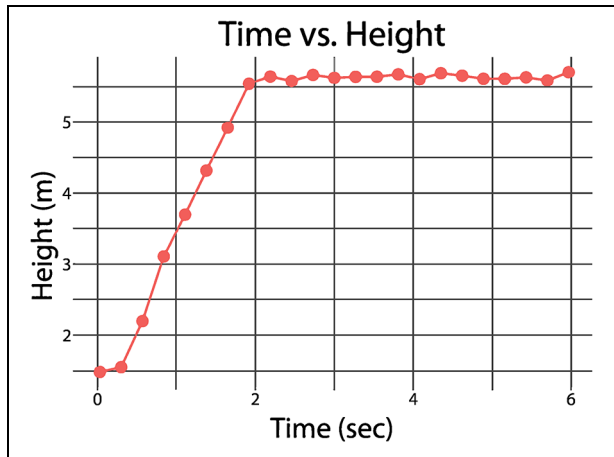


Figure 8. Illustration of the takeoff method.

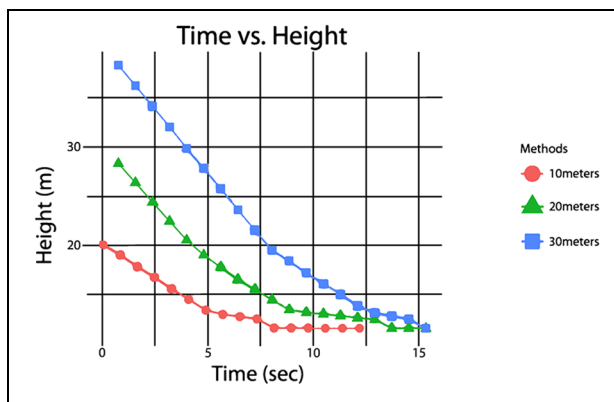


Figure 9. Illustration of the landing method.

quadrotor goes into takeoff mode from the initial phase at 0.24 seconds, the second phase at 3.11 meters at 0.84 seconds, and the final phase at 1.92 seconds, where it hovered at a steady 5.63 meters.

Figure 9 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, as the ultrasonic sensor is still affixed to the bottom. In Figure 9, we present landing from 10, 20, and 30 meters. One can notice that all landings have a slight curve before they land, which illustrates a smooth landing.

The standard errors and times for landing from 10, 20, and 30 meters are shown in Table 3. From the table, we can see that the standard error is lower than 1. This illustrates that the landing method does indeed ensure a smooth landing.

9.3. Obstacle detection

The quadrotor starts on a platform positioned at (0,0,0): the takeoff pad. In this simulation, we have three different

Table 3. Standard errors and timing for landing.

Starting height	Smooth time	Landing time	Standard error
10 meters	5.1	7.8	0.8512206
20 meters	9.03	13.38	0.6584615
30 meters	12.63	14.82	0.3058578

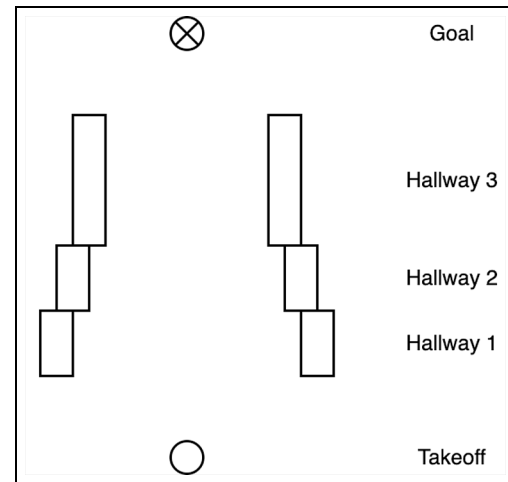


Figure 10. Map of the three hallways and the takeoff and goal locations.

hallways that the quadrotor must fly between. The first is a hallway of 7 meters in width, 2 meters long, and its center is at (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, 4 meters long, and its center is at (8, 0, 0). The quadrotor's goal is at (12, 0, 5), located 2 meters away from the end of the third hallway. Figure 10 shows the map of the hallways, takeoff, and goal locations.

An obstacle detection method is considered to be 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 costs.

Figure 11 shows the quadrotor starting at 0 meters in the x direction and moving to position 12 in the environment. Obstacle avoidance method 1 arrived at the goal in 123.43 seconds, while method 2 reached it in 150.12 seconds.

From Figure 11, one can notice that method 1 is in the first hallway at time markers 18.06–35.91, while method 2 is in the first hallway at time markers 19.35–42.09. The second hallway's time markers are 35.91–51.57 and 42.09–64.74 seconds, while the third hallway's time markers are 51.57–95.00 and 64.74–120.93 seconds for methods 1 and 2, respectively.

The maximum value for method 1 along the y -axis is 0 meters, while the minimum value is -0.72 meters. The

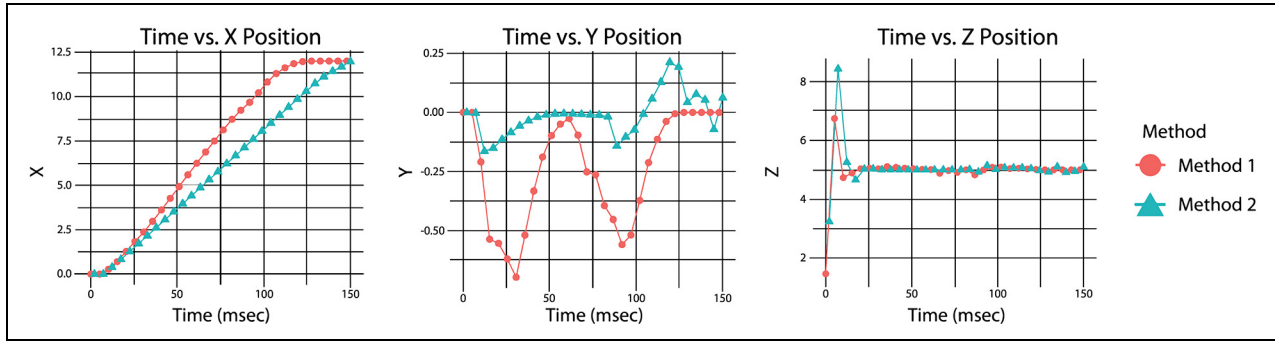


Figure 11. The x , y , and z comparison for obstacle detection methods 1 and 2.

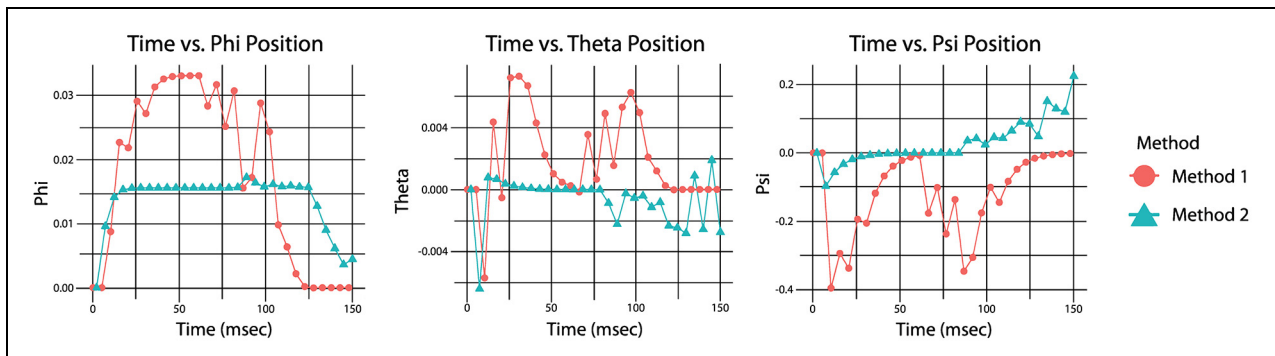


Figure 12. The ϕ , θ , and ψ comparison for obstacle detection methods 1 and 2.

maximum value for method 2 is -0.179 meters and the minimum is 0.21 meters.

The reason why the difference in the y position for method 2 is smaller than the y position for method 1 is that method 2 takes into account the left- and right-hand ultrasonic sensor values and attempts to position itself in the center of those values. Method 1 does not take into account the values of the wall, except for when the values are too close for collisions to occur, which causes spikes, as shown at time markers 29.52 and 89.34 seconds when method 1 responds to the quadrotor being too close to the hallway's right-hand wall.

Figure 12 shows the quadrotor's ϕ , θ , and ψ degrees according to their time marker, denoting their position in the multiple hallways.

Standard errors are shown in Table 4. From Figures 11 and 12 and Table 4, we conclude that obstacle detection method 2 was smoother, as its average standard error was 28.00426 while method 1's average was 33.4292.

9.4. Relative positioning

The graphs shown in Figures 13 and 14 display the comparison between the simulation models' perceived flight and the simulated flight. The simulation models'

Table 4. Standard errors for obstacle detection in multiple hallways.

Comparison	Method 1	Method 2
x	6.224846	1.156503
y	37.9196	25.09231
z	40.98503	40.22925
ϕ	36.44013	34.7439
θ	39.47894	38.23012
ψ	39.52665	18.57348
Overall	33.4292	28.00426

perceived flight is represented by the relative method, while the simulated system flight is represented by the real method in the results below. There is a platform in the environment positioned at $(0, 0, 0)$, the starting position of the quadrotor. This allows the quadrotor to set up its initial positions accurately. The target of the quadrotor is located at $(2, 2, 5)$. The IMU's noise level was set to zero to remove the sensor error noise from the experiment.

Figure 13 shows the real linear positions in the environment versus the relative linear positions calculated by the quadrotor. Figure 14 shows comparisons between the real and relative angular positions calculated by the quadrotor.

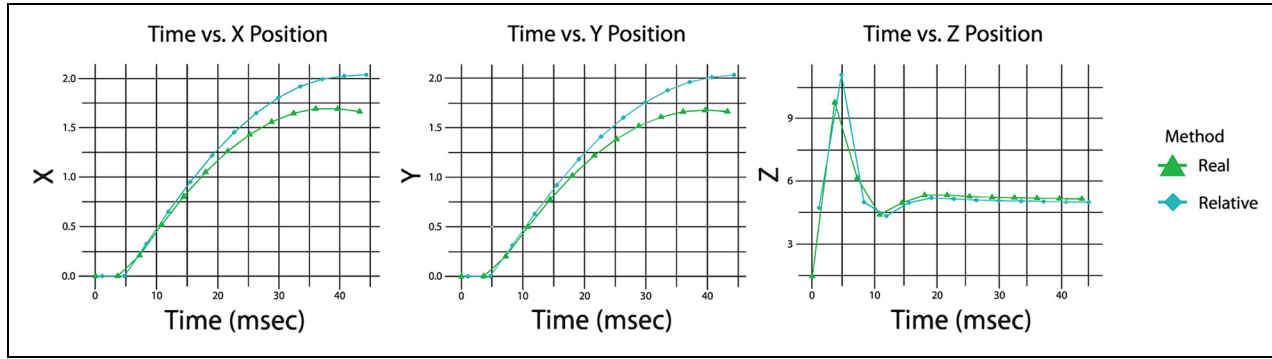


Figure 13. The x , y , and z comparison for real and relative linear positions.

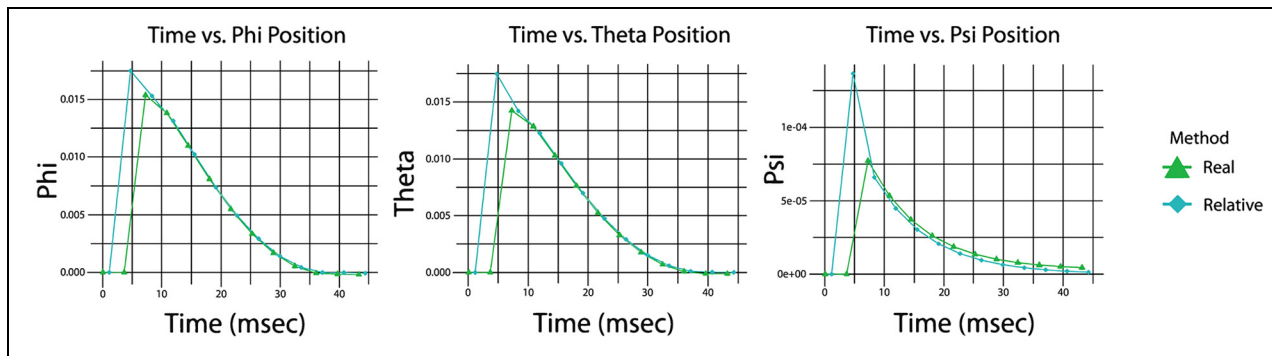


Figure 14. The ϕ , θ , and ψ comparison for real and relative angular positions.

Table 5. Simulation constants.

Comparison	Standard error real	Standard error relative	Overall maximum difference
x	2.910365	1.648658	0.4068883
y	2.398179	1.363661	0.3996819
z	11.23941	11.35762	1.474556
ϕ	10.21809	10.26872	0.001490871
θ	10.25515	10.29908	0.001490871
ψ	10.74641	10.69983	4.5587e-05
Overall	7.961267	7.606261	0.3806922

The standard errors and differences between the relative and real positions are shown in Table 5. The overall difference in the standard error of the relative position and the real position is 0.355006, where the relative position is linearly smoother than the real position.

The relative and real positions differ maximally by 38 centimeters over a distance of 5.74 meters. The path of the relative position closely follows the real path. During the flight, the relative position is greater than the real position. This is most notable in the z position, as the quadrotor rapidly takes off to a height of 11 meters before falling down to the target height of 5 meters in under 10 seconds. The

relative positions are more accurate across the angular movements than the linear movements.

9.5. Flocking positioning

In Micklisch et al.,¹⁵ a group of eight autonomous quadrotors are introduced into a simulated environment and the target position of the central neighborhood is set at (5,5,5). We illustrated that the group of quadrotors successfully positions itself at the target position.

10. Conclusion

In this paper, we presented a simulator for a fully autonomous quadrotor system and the sensors used to gather information in the simulated environment. Quadrotor simulators provide a testbed for control and stabilization mechanisms, the reproduction of flight and sensor dynamics, and a tool for autonomous flight research. We have presented the software design to construct such a testbed. Our simulator is scalable and provides a valuable tool to realistically study a system of quadrotors in different scenarios and environments. In this research, we presented and validated attitude controllers, takeoff, and landing methods, an altitude controller, obstacle detection

methods, and flocking mechanisms. We used our scalable simulator to test the effectiveness of our methods and controllers and provide performance comparisons for each one. We illustrated that we have provided an environment with sufficient sensory information of the quadrotor to allow for the simulation and evaluation of several control methods of an autonomous flight.


11. Future work

For future work, we will more accurately simulate the sensors described in this paper by closely mimicking their error models. This will provide an avenue for implementing and testing digital filters to estimate the output of the sensors utilized. We would like to implement a real-life autonomous quadrotor to compare its performance to the results obtained from our simulator.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

ORCID iD

Christian Micklisch  <https://orcid.org/0000-0002-9675-280X>

References

- Mulgaonkar Y, Cross G and Kumar V. Design of small, safe and robust quadrotor swarms. In: *2015 IEEE international conference on robotics and automation (ICRA)*, Seattle, WA, USA, 26–30 May 2015, pp. 2208–2215. IEEE.
- Briod A, Kornatowski P, Zufferey JC, et al. A collision-resilient flying robot. *J Field Robot* 2013; 31: 496–509.
- Dong W, Gu GY, Zhu X, et al. Development of a quadrotor test bed — modelling, parameter identification, controller design and trajectory generation. *Int J Adv Robot Syst* 2015; 12: 7.
- Li J and Li Y. Dynamic analysis and PID control for a quadrotor. In: *2011 IEEE international conference on mechatronics and automation*, Beijing, China, 7–10 August 2011, pp. 573–578. IEEE.
- Sellali BB and Allali A. Neuro — fuzzy methods coupled to operational PID, to improve the flight parameters of a drone. In: *2017 18th international conference on sciences and techniques of automatic control and computer engineering (STA)*, Monastir, Tunisia, 21–23 December 2017, pp. 314–319. IEEE.
- Rosca S, Leba M, Ionica A, et al. Quadcopter control using a BCI. *IOP Conf Ser Mater Sci Eng* 2018; 294: 12048.
- Meyer J, Sendobry A, Kohlbrecher S, et al. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In: *Simulation, modeling, and programming for autonomous robots*. Berlin Heidelberg: Springer, 2012, pp.400–411.
- Curi S, Mas I and Pena RS. Autonomous flight of a commercial quadrotor. *IEEE Latin Am Trans* 2014; 12: 853–858.
- Santamaria-Navarro A, Lippiello V and Andrade-Cetto J. Task priority control for aerial manipulation. In: *2014 IEEE international symposium on safety, security, and rescue robotics*, Hokkaido, Japan, 27–30 October 2014, pp. 1–6. IEEE.
- McAnanama JG and Marsden G. An open source flight dynamics model and IMU signal simulator. In: *2018 IEEE/ION position, location and navigation symposium (PLANS)*, Monterey, CA, USA, 23–26 April 2018, pp. 874–881. IEEE.
- Holder M, Rosenberger P, Bert F, et al. Data-driven derivation of requirements for a Lidar sensor model. *Graz* 2018; 1–10.
- Durst PJ, Goodin C, Gates BQ, et al. The need for high-fidelity robotics sensor models. *J Robot* 2011; 2011: 1–6.
- Durst PJ and Goodin C. High fidelity modelling and simulation of inertial sensors commonly used by autonomous mobile robots. *World J Model Simulat* 2012; 8: 172–184.
- Kranz AE. *Improving sonar sensor fidelity in a robot simulator ultrasonic sensor*. Dissertation, Prince Gorge, University of Northern British Columbia, 2014.
- Micklisch C, Hilliard N and ElAarag H. Modeling and simulation of scalable flocking with fully autonomous quadrotors. In: *proceedings of the 21st international symposium on performance evaluation of computer and telecommunication systems*, Bordeaux, France, 9–12 July 2018, pp. 1–9. Society for Modeling and Simulation International (SCS).
- Micklisch C, Hilliard N and ElAarag H. Modeling and simulation of fully autonomous quadrotors. In: *proceedings of the communications and networking symposium*, Baltimore, MD, USA, 15 April 2018, paper no. 5, p. 5. San Diego: Society for Computer Simulation International.
- Su H, Wang X and Lin Z. Flocking of multi-agents with a virtual leader. *IEEE Trans Auto Contr* 2009; 54: 293–307.
- Fay G. *Derivation of the aerodynamic forces for the Mesicopter simulation*. Stanford, CA: Stanford University, 2001.
- Bouabdallah S. *Design and control of quadrotors with application to autonomous flying*. Lausanne: Epfl, 2007.
- Kemper E. *Quad-rotor flight path energy optimization*. Doctoral Dissertation, Northern Arizona University.
- Pololu Robotics and Electronics. MinIMU-9 v3 gyro, accelerometer, and compass (L3GD20H and LSM303D carrier), <https://www.pololu.com/product/2468> (accessed 23 September 2014).
- Adafruit Industries. Unique & fun DIY electronics and kits]. *BMP180 Digital pressure sensor*, <https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf> (5 April 2013, accessed 23 April 2015).
- DealeXtreme - Cool Gadgets at the Right Price - DX Free Shipping Worldwide. Navo Ultrasonic Sensor Distance Measuring Module, <https://web.archive.org/web/20201124041146/https://esjdedxylxgadgets.wordpress.com/2015/04/26/navo-ultrasonic-sensor-distance-measuring-module-green-270051/> (accessed 23 November 2020).
- Möller T and Trumbore B. Fast, minimum storage ray/triangle intersection. In: *ACM SIGGRAPH 2005 courses*, 31 July 2005, p.7.
- Hoffmann G, Huang H, Waslander S, et al. Quadrotor helicopter flight dynamics and control: theory and experiment. In: *AIAA guidance, navigation and control conference and exhibit*, 20 August 2007, paper no. 9, p. 6461. Elsevier BV.

26. Dikmen İC, Arisoy A and Temeltas H. Attitude control of a quadrotor. In: *2009 4th international conference on recent advances in space technologies*, Istanbul, Turkey, 11 June 2009, pp.722-727. IEEE.
27. Gageik N, Rothe J and Montenegro S. Data fusion principles for height control and autonomous landing of a quadcopter. In: *proceedings of the UAVweek workshop contribution*, Siegen, Germany, 20–21 November 2012, pp. 1–6. Würzburg: University of Würzburg.
28. Prashanth KR, Shankpal P, Nagaraja B, et al. Real time obstacle avoidance and navigation of a quad-rotor MAV using optical flow algorithms. *Sastech J* 2013; 12: 31-35.
29. Gageik N, Müller T and Montenegro S. Obstacle detection and collision avoidance using ultrasonic distance sensors for an autonomous quadcopter. University of Würzburg, Aerospace Information Technology (Germany), Würzburg, 3–23 September 2012.
30. Wagster J, Rose M, Yeralian H, et al. Obstacle avoidance system for a quadrotor UAV. In: *infotech@ aerospace* Garden Grove, CA, USA, 19–21 June 2012, paper no. 2548.
31. Fernández I, Mazo M, Lázaro JL, et al. Local positioning system (LPS) for indoor environments using a camera array. In: *the 11th international conference on advanced robotics (ICAR 2003)*, Coimbra, Portugal, 30 June–3 July 2003, paper no. 239, Vol. 2, pp.613–618.
32. Kim N and Kim Y. *Indoor positioning system with IMU, map matching and particle filter. recent advances in electrical engineering and control applications*. Berlin: Springer, 2015, pp.41–47.
33. Reynolds CW. Flocks, herds, and schools: a distributed behavioral model. *Comput Graph* 1987; 21: 25–34.

Author biographies

Christian Micklis obtained his bachelor degree in Computer Science from Stetson University in 2015. His research interests include modeling and simulation and performance evaluation. His email address is cmicklis@stetson.edu.

Nathan Hilliard obtained his bachelor degree in Computer Science from Stetson University in 2015. His research interests include modeling and simulation and performance evaluation. His email address is nhilliard@stetson.edu.

Hala ElAarag obtained her PhD in Computer Science from the University of Central Florida in 2001. She has been a member of the Faculty of Computer Science at Stetson University since August 2002. Her research interests include computer networks, computer architecture, modeling and simulation and performance evaluation. Her email address is helaarag@stetson.edu.