12-2019

Multirotor UAS Sense and Avoid with Sensor Fusion

Jonathan Mark Buchholz

MULTIROTOR UAS SENSE AND AVOID WITH SENSOR FUSION

A Thesis

Submitted to the Faculty

of

Embry-Riddle Aeronautical University

by

Jonathan Mark Buchholz

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Unmanned and Autonomous Systems Engineering

December 2019

Embry-Riddle Aeronautical University

Daytona Beach, Florida | Prescott, Arizona

Multirotor UAS Sense and Avoid with Sensor Fusion

by

Jonathan Mark Buchholz

This thesis was prepared under the direction of the candidate's thesis committee chairman, Dr. Sam Siewert, Department of Computer, Electrical, and Software Engineering, and has been approved by the members of his thesis committee. It was submitted to the Electrical, Computer, Software, and Systems Engineering Department and was accepted in partial fulfillment of the requirements for the degree of Masters in Unmanned and Autonomous Systems Engineering.

THESIS COMMITTEE:

12/6/2019

Dr. Sam Siewert
Committee Chairman

Dr. Richard Stansbury
Committee Member

12/6/2019

Dr. Douglas Isenberg
Committee Member

12/6/2019

Dr. Steven Bruder
Committee Member

12/6/2019

Dr. Iacopo Gentilini
Committee Member

Timothy A. Wilson
Department Chair, Electrical, Computer,
Software, and Systems Engineering

Associate Vice President For Academics

ACKNOWLEDGEMENTS

I would like to thank Dr. Sam Siewert for dedicating a great deal of time and patience to advising my progress on this project. To Dr. Richard Stansbury, for his thesis committee membership, and also for ensuring my transition between sister campuses was well received. I am thankful to Dr. Steven Bruder and Dr. Douglas Isenberg for their membership in my thesis committee and for their excellent advice. I would also like to thank Dr. Ken Bordignon for allowing my use of his previous work. I owe thanks to the Embry-Riddle ICARUS Research group for their assistance with testing procedures. Without the constant support of my girlfriend, my parents, my siblings, and my friends old a new, such work would not have been possible. I am forever thankful to all of you. Finally, were it not for Dr. Iacopo Gentilini and his belief in my academic capabilities, I would not have pursued a graduate degree in the first place. Thank you.

Table of Contents

ABBREVIATIONS

ADS-B	Automatic Dependent Surveillance-Broadcast
DOF	Degree of Freedom
EO/IR	Electro-Optical/InfraRed
FAA	Federal Aviation Administration
FOV	Field of View
GA	General Aviation
GPS	Global Positioning System
IMU	Inertial Measurement Unit
IR	InfraRed
LIDAR	Light Detection and Ranging
OODA	Observe Orient Decide Act
RADAR	Radio Detection and Ranging
RGB	Red, Green, Blue (camera/image)
RGBD	RGB Depth
ROS	Robot Operating System
RPM	Rotations per Minute
SAA	Sense and Avoid
ToF	Time of Flight
UAS	Unmanned Aircraft System
UV	UltraViolet
VFH	Vector Field Histogram

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this thesis, the key concepts of independent autonomous Unmanned Aircraft Systems (UAS) are explored including obstacle detection, dynamic obstacle state estimation, and avoidance strategy. This area is explored in pursuit of determining the viability of UAS Sense and Avoid (SAA) in static and dynamic operational environments. This exploration is driven by dynamic simulation and post-processing of real-world data. A sensor suite comprised of a 3D Light Detection and Ranging (LIDAR) sensor, visual camera, and 9 Degree of Freedom (DOF) Inertial Measurement Unit (IMU) was found to be beneficial to autonomous UAS SAA in urban environments. Promising results are based on to the broadening of available information about a dynamic or fixed obstacle via pixel-level LIDAR point cloud fusion and the combination of inertial measurements and LIDAR point clouds for localization purposes. However, there is still a significant amount of development required to optimize a data fusion method and SAA guidance method.

1.0 Introduction

1.1 Problem Definition

Considering potential large-scale implementations of autonomous UAS operations in urban environments, complete reliance on external networks or broadcasts such as Global Positioning System (GPS) or Automatic Dependent Surveillance-Broadcasting (ADS-B) presents concerns for system contingencies should network reliability be compromised. Further, the possible presence of non-cooperative UAS in urban environments threatens the safety of systems which cannot autonomously Sense and Avoid (SAA) aerial dynamic obstacles. For this reason, the methods by which autonomous UAS can continue to operate in these conditions are reliant on independent sensing and processing capabilities. To the author's best knowledge, no current systems which operate independently of external networks such as GPS or ADS-B have simultaneously considered both independent localization (determination of the system's position) and obstacle detection.

1.2 Literature Review

1.2.1 SAA Methods

The Observe Orient Decide Act (OODA) Loop is a conceptual avoidance tool originally used to train US Air Force fighter pilots. Intended for use in adversarial situations involving enemy planes, the OODA Loop encouraged pilots to reacting to assumed enemy actions preemptively. Pilots would observe enemy behavior as well as any environmental factors. With that information, they would reorient themselves with respect to their enemy's position, considering how they would be likely to react. They

would then decide on the best way to confuse the enemy and act out that decision. The OODA Loop also saw use outside of aircraft as a generalized tactical tool for outsmarting one's enemy. This system is not entirely relevant to UAS SAA as the goal for the latter is to avoid rather than offensively confront dynamic obstacles. The general framework is reminiscent of reactive autonomous SAA systems and subsequently has been adapted to a command and control setting [1], [2].

Another GA SAA technique is the Traffic Collision Avoidance System II (TCAS II). TCAS II was developed in 1989 to actively prevent midair collisions between compliant GA aircraft [3]. If the paths of two aircraft would intersect or become uncomfortably close, signals would be sent between them specifying a change in course that would deconflict their trajectories. This results in a separation between the aircraft that can be accounted for miles before the intersection would have taken place.

Due to the complexity of adapting OODA to autonomous UAS and the required compliance of TCAS II, methods for obstacle avoidance that are typically used in mobile robotics applications seemed favorable.

In mobile robotics, the potential field model functions on the principle of artificial attraction and repulsion [4]. When a mobile robot becomes close enough to an obstacle, a repulsive force is applied to control the system away from the obstacle. Conversely, an attractive force is applied to draw the system closer to its desired position. While this may work for certain situations, this algorithm is incapable of handling local minima, which can trap it in a position where the repulsive and attractive forces cancel each other out. This occurs more often in 2D systems than in 3D.

Alternatively, the Vector Field Histogram (VFH) is a robust reactive avoidance system for 2D mobile robots [5]. This algorithm uses ranged data to build a polar histogram that defines directions that the mobile robot can safely travel in. This system, like the potential field algorithm, is susceptible to local minima if implemented without additional modifications. Since both potential field and VFH methods possess the same limitations, the simplicity of the potential field was preferable.

1.2.2 Available Sensors

Autonomy for UAS requires specific sensing capabilities to acquire information about the environment and the UAS itself. The information that is most important to SAA is location of obstacles and self-localization. Without range data, obstacles could not be detected in 3D space. Without self-localization, the UAS has no global reference for where itself or a detected obstacle are positioned. For an independent system, this data must be captured entirely from the on-board sensors [6]. The usefulness of various sensors in terms of their applicability to independent autonomous SAA is assessed in this section.

Considering the following sensors, each can be classified as either passive of active. Passive sensors acquire data by accepting energy already present in the environment. Active sensors require stimulation of the environment and detection of a respective response. Both passive and active sensors have benefits and disadvantages in specific scenarios. Most notably, active sensor stimulus can be detected by external systems, not only by the sensor itself. This makes passive sensors more viable for situations where stealth is of higher importance. For some autonomous UAS applications, this may be worth consideration, but for applications like parcel delivery, stealth is

largely unnecessary. A benefit of active sensing is a lack of dependence on existing energy in the environment. A simple analogy for this is a passive visible camera equipped with a light source. The visible camera alone is most useful during the day, but a light source can be used to stimulate the environment at night.

Active Ranging Sensors

Direct ranging sensors are capable of raw distance measurements. The primary purpose of these sensors is to detect physical structures. The following sensors are classified as active, meaning they emit energy, unlike passive sensors.

Radio Detection and Ranging (RADAR) is a widely used and well-established method of detecting structures such as buildings, planes, and ships based on their reflection of emitted Radio Frequency (RF) signals. The reflections of these signals can be detected as far away as several kilometers. It is also worth noting the emerging technology of passive RADAR. Rather than emitting an RF signal like active RADAR, passive RADAR relies on existing RF signals in the environment.

Light Detection and Ranging (LIDAR) is a distance measurement method which uses infrared lasers in a similar way that RADAR uses RF signals. A beam of light is emitted by the sensor, absorbed and reradiated by an obstruction, and then detected by the sensor. By knowing the speed of light and the timing between emission and detection, the distance that the beam traveled can be determined. LIDAR using a single beam, appropriately named beam LIDAR, provide a single distance reading along one direction. A beam LIDAR moving in a rasterized fashion creates a plane LIDAR, which can read measurements at various angles within a single plane. Rasterizing multiple beam LIDAR at distinct angles creates a 3D LIDAR, which can read measurements at various angles

within multiple planes or conical regions. The Doppler effect can also be used to determine speed of an obstacle by measuring the frequency shift between emitted and returned light waves.

Time of Flight (ToF) cameras use the same principle as LIDAR, but to a different extent. Much like the pixels of a visible camera, ToF cameras organize distance readings in a grid format. This is possible by using an array of light sensors. The result is a dense set of measurement data across a narrow Field of View (FOV).

Ultrasonic sensors operate by timing the reflection of sound waves. These sensors are typically used in UAS for altitude measurement and detection of nearby obstacles. While ultrasonic sensors are simple in design and relatively inexpensive compared to other active sensors, they are limited to detection in a single direction. Arrays of ultrasonic sensors are a potential solution to this, but these are susceptible to "chatter" in which sensors mistake a returned sound wave originating from other sensors as their own.

Infrared ranging sensors emit a pulse of infrared light that returns via reflection to an infrared-sensitive receiver. The intensity of the return is used to calculate the distance to the reflecting object, within an operating range. While these sensors are useful in indoor settings, the natural infrared light coming from the Sun can drastically effect sensor functionality.

RGB Depth (RGBD) sensors are active sensors that use recognizable infrared projections that can evaluate relative distance when objects in the field of view distort that projection. These sensors also come with visual cameras that can provide a corresponding color image. Like infrared ranging, RGBD sensors are typically limited to indoor applications away from direct sunlight.

Passive Ranging Sensors

The emerging technology of passive RADAR is also worth considering for UAS SAA. Rather than emitting an RF signal like active RADAR, passive RADAR relies on existing RF signals in the environment. This allows for covert detection, which as stated before is not necessarily a requirement for autonomous UAS SAA. However, the reliance on external RF signals disqualifies it for consideration in independent UAS SAA.

Passive Imaging Sensors

Imaging sensors, also known as Electro-Optical/InfraRed (EO/IR) cameras, operate via arrays of passive light detectors that are sensitive to specific wavelengths of light ranging from UltraViolet (UV) to InfraRed (IR). These wavelengths can be visible (300 to 700 nm) as in typical RBG cameras, or in various regions of the infrared spectrum (700 nm to 14 μm). The choice of which type of sensor to use is dependent on its use case.

Visual cameras are preferred in situations where analogs for human vision are possible and useful but share similar limitations. Visual cameras are reliant on the presence of external light sources. This makes them useful for daylight or indoor applications but hinders their utility at night.

Infrared cameras can detect wavelengths of light outside the visual spectrum. This can be leveraged for thermal imaging in the case of long-wave infrared (LWIR) or to visually penetrate cloud cover in the case of short-wave infrared (SWIR).

Navigational Sensors

Inertial Measurement Units (IMUs) are the most widely used independent navigational sensors. The components of an IMU are typically a combination of multiple

accelerometers, gyroscopes, and magnetometers aimed in orthogonal directions. This provides up to a three-axis representation of a body's linear motion, angular motion, and orientation. IMUs are typically paired with GPS because without position measurements, the system's localization will experience drift caused by numerical integration of accumulating errors in acceleration measurements.

1.2.3 Existing Systems

Current UAS systems have focused on addressing GPS loss in urban environments, independent sensing, and SAA, but not simultaneously.

An urban UAS navigation system based on LIDAR, GPS, and known maps is described by Chen *et al* [7]. This system is designed to be resilient to losses in GPS by leveraging maps and feature recognition in LIDAR point clouds. However, the system's dependence on map truth models hinders its flexibility to unexpected or truly unknown environments.

Scannapieco *et al* [8] present a proof-of-concept RADAR odometry system for small fixed-wing UAS. This system used RADAR exclusively to receive two-dimensional motion and had potential for real-time operations. Still, they claim that independent localization in urban environments is an open problem.

GPS-denied localization can be possible through downward-facing optical flow, as presented by Pestana *et al* [9]. Their system was proven to work for both indoor and outdoor environments. While this system can effectively handle independent localization, its situational awareness to potential obstacles was not considered.

1.3 Constraints and Assumptions

To clearly state the decided approach for this thesis, constraints and assumptions are made regarding the project's scope.

For constraints, the UAS SAA system will assume no prior knowledge of the environment. The functionality of this SAA must be useful in a general context, and not particular to specific types of dynamic obstacles that appear in urban environments. The SAA system is further constrained by the exclusive use of independent sensing rather than reliance on ground-based systems, external networks, or cooperative UAS.

It is assumed that the environment which the UAS inhabits is primarily static, with potentially a single unknown dynamic obstacle. The nature of this dynamic obstacle will be indifferent such that the presence of the UAS will not have an effect on the path of the obstacle. Finally, the capability of the UAS to operate in real-time was not considered but could be approached in future work. All analysis is applied to post-processed data and simulation.

2.0 Experimental Configuration

2.1 Sensor Suite

The sensor suite can be separated into three elements; sensors, processor, and power supply. The sensors included in this setup are a 3D LIDAR (Velodyne Puck LITE) [10], a 720p optical webcam (Logitech C270) [11], and an IMU/GPS unit (VectorNav VN200) [12]. A wiring diagram for the sensor suite is shown in Figure 2.1 below.

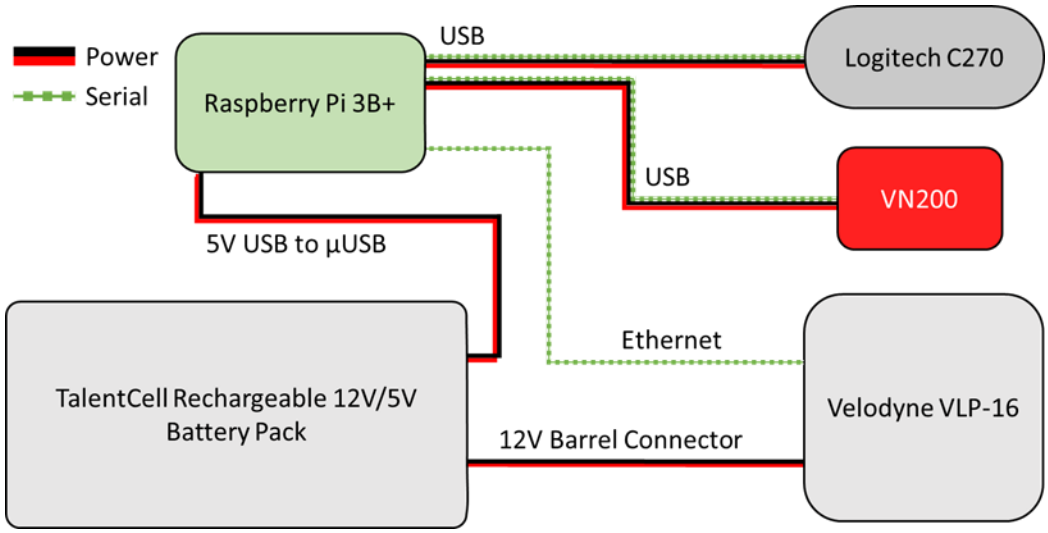

Figure 2.1: Sensor Suite Wiring Diagram

In the wiring diagram, the battery pack supplies power to the Raspberry Pi 3B+ and the LIDAR. Power and serial communication for the visual camera and IMU are provided through USB connection to the Raspberry Pi 3B+. Serial communication with the LIDAR is provided via an ethernet cable. The physical system is shown in Figure 2.2.

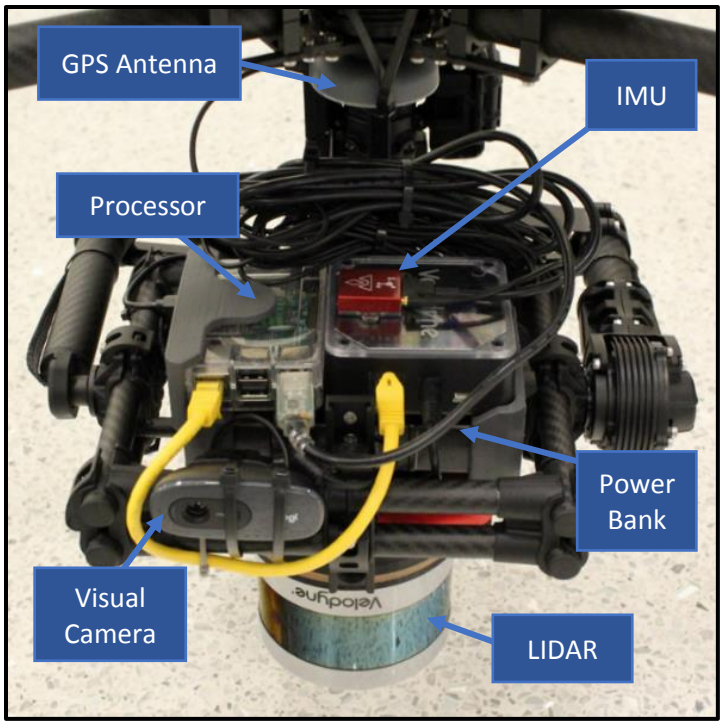

Figure 2.2: Mounted Sensor Suite

Often used in autonomous driving systems, the 3D LIDAR possesses a wide field of view (360°H x 30°V) and an effective measurement range of 1 to 100 meters (accurate to (+/-) 3 centimeters). The vertical field of view is spanned by sixteen (16) emitter and receiver pairs placed every 2° between +15° and -15° from horizontal. With rasterization, the result is a series of measurement bands. The horizontal resolution of this LIDAR is dependent on the speed at which the sensor rasterizes. The tradeoff between horizontal resolution and frequency of data return is dependent on this speed. For this project, a default speed of 600 RPM is used, resulting in a horizontal resolution of 0.01° with an accuracy of (+/-) 0.005° at a framerate of 10 Hz. The RPM range for the LIDAR is between 300 and 1200 RPM, which linearly scales the resolution between 0.005° and 0.02° and the framerate between 20 Hz and 5 Hz, respectively.

The IMU/GPS unit serves two purposes; first by emulating an independent system and second by having a relative truth model for localization. In an independent system, the IMU would be used exclusively in conjunction with other independent sensors for localization purposes. For testing purposes, a fused IMU/GPS localization method can be used as a truth model to compare against independent localization.

The sensor suite processor is a Raspberry Pi 3B+ running ROS (Robot Operating System) [13] on an Ubuntu Linux distribution. ROS is generally used in embedded software as a base architecture for various robotics applications. In this case, ROS is used for synchronized collection of LIDAR and camera data. Existing user-made packages for the LIDAR and visual camera are used for interfacing to ROS. The INS/GPS unit data is recorded via a separate Linux Bash script[1].

2.3 Flight Platform

The system for capture of mid-flight data consists of a manually controlled carrier drone and an isolated sensor suite. The carrier drone is a Freefly ALTA 6, a hexrotor with a span of over 1 meter capable of lifting a payload of 6 kilograms [14]. The Figure 2.3 presents the flight capture system in its entirety.

[1] VN200 Bash script provided by David Stockhouse, ERAU ICARUS Research Group.

Figure 2.3: Experimental Flight Data Capture System

This system was selected in part due to simplicity in adapting sensor suite components to an existing drone platform, and in part due to isolation of the sensor suite dynamic behavior from the carrier drone dynamic behavior.

3.0 Algorithm Overview

3.1 Obstacle Detection

Obstacle detection is dependent on ranged data and is therefore primarily driven by measurements provided by LIDAR. This method of recursive voxelized point cloud segmentation is adapted from work by Vo et al [15]. Initially, points are separated into bins called voxels based on their Cartesian location. A best-fit plane is defined for each voxel based on a least-squares approach as defined using the following equations.

$$\begin{Bmatrix} x_c = \left[p_{1,x}, p_{2,x}, \dots, p_{n,x}\right]^T - \bar{p}_x \\ y_c = \left[p_{1,y}, p_{2,y}, \dots, p_{n,y}\right]^T - \bar{p}_y \\ z_c = \left[p_{1,z}, p_{2,z}, \dots, p_{n,z}\right]^T - \bar{p}_z \end{Bmatrix} \quad \text{Equation 3.1}$$

$$sp(x_c, y_c) = \sum_{i=1}^{n} x_{c,i} * y_{c,i} \quad \text{Equation 3.2}$$

$$n = \begin{bmatrix} sp(y_c, z_c) * sp(x_c, y_c) - sp(x_c, z_c) * sp(y_c, y_c) \\ sp(x_c, y_c) * sp(x_c, z_c) - sp(x_c, x_c) * sp(y_c, z_c) \\ sp(x_c, x_c) * sp(y_c, y_c) - sp(x_c, y_c) * sp(x_c, y_c) \end{bmatrix} \quad \text{Equation 3.3}$$

$$\hat{n} = \frac{n}{\|n\|_2} \quad \text{Equation 3.4}$$

where x_c, y_c, and z_c are the coordinates of the voxel points $p_1$through p_n from their centroids, $\bar{p}_x$, $\bar{p}_y$, and $\bar{p}_z$, respectively. The function $sp(\cdot,\cdot)$ is used as shorthand for the element-wise sum of products. The resultant vector $\hat{n}$ is normal to the best-fit plane passing through the centroid of the voxel. From the normal vector, the residual noise can be characterized through Equations 3.5 and 3.6.

$$d_i = p_i^T - [\bar{p}_x \quad \bar{p}_y \quad \bar{p}_z] * \hat{n} \quad \text{Equation 3.5}$$

$$r = \sqrt{\frac{1}{k}\sum_{i=1}^{k} d_i^2} \quad \text{Equation 3.6}$$

where d_i is the distance of each point in the voxel to the best-fit plane, and r is the residual error of all points in the voxel.

Moving from the voxel preparation stage to the voxel reduction stage, the residual error is checked against a threshold. If this threshold is exceeded, the voxel undergoes reduction, by which it splits into 8 octant voxels. These new voxels will be processed the same way as the initial voxel, until a threshold for residual or minimum voxel size is met.

Once the set of voxels is sufficiently reduced, region growth begins. Starting with the voxel with the least residual error, neighboring voxels will be considered for region growth. If that voxel's residual is sufficiently low and its normal vector is sufficiently aligned with the seed normal, then the voxels are joined as a region. Region growth continues until there is no valid seed voxel to consider.

Applying this method to obstacle detection, regions from separate LIDAR point clouds can be compared and motion can be extrapolated from regions that otherwise inexplicably moved between point clouds.

3.2 Obstacle Avoidance

The path which the UAS takes is defined by a series of points, globally prescribing the position which the UAS must reach and the velocity it must maintain when reaching the position. The inclusion of velocity allows for preemptive course correction to aim the UAS toward subsequent waypoints with manageable overshoot.

When a dynamic obstacle is detected, its location is compared to the current position of the UAS. If the two are within a distance threshold of each other, a fictitious force is applied to the control law of the UAS, proportional to the inverse square of the distance between the bodies.

This method is functional, but not optimal for smooth avoidance paths. As the UAS is pushed by this fictitious force, the error from the path following command is accumulated. After avoidance, when the fictitious force is no longer in effect, the compiled error from path following results in an abrupt return to the path.

3.3 Point Cloud Image Fusion

The LIDAR and camera fusion method explored in this project is primarily reliant on coordinate transformations. Initially, point clouds are captured in the LIDAR coordinate frame and designated by a vector in that frame. The location of each point can be defined as $p_{i,L}$ for each i point in the complete point cloud. The location and orientation of the camera frame is assumed to be known relative to the LIDAR frame. The transformation of point $p_{i,L}$ in the LIDAR frame to $p_{i,C}$ in the camera frame is given in Equation 3.1.

$$p_{i,C} = (R_C^L)^{-1} * (p_{i,L} - o_L^C) \quad \text{Equation 3.1}$$

where R_C^L is the rotation matrix to orient the LIDAR frame with the camera frame, and o_L^C is the location of the camera in the LIDAR frame. With each point transformed to the camera frame, the field of view of the camera can be modeled as a region in a spherical coordinate system. By converting each camera frame point into spherical coordinates, the points which lie within the field of view of the camera can be isolated. Once these visible points are isolated, their relative position within the field of view can be used to associate those points with pixels in a camera image. Assuming a similar coordinate frame to that of the LIDAR, the bounds of the field of view of the camera can be defined by the following equations.

$$\varphi_{min} = -\frac{HFOV}{2}, \varphi_{max} = \frac{HFOV}{2} \quad \text{Equation 3.2}$$

$$\theta_{min} = 90° - \frac{VFOV}{2}, \theta_{max} = 90° + \frac{VFOV}{2} \quad \text{Equation 3.3}$$

where φ is defined as a right-hand rotation about z, starting at x, and θ is defined as a downward rotation from z toward the xy plane. HFOV and VFOV represent the camera's horizontal and vertical field of view (in degrees). The associated pixel for a point within the field of view is given by Equation 3.4 and 3.5.

$$row_i = ceil\left(h * \frac{\theta_{i,C} - \theta_{min}}{VFOV}\right) \quad \text{Equation 3.4}$$

$$column_i = ceil\left(w * \frac{\varphi_{max} - \varphi_{i,C}}{HFOV}\right) \quad \text{Equation 3.5}$$

where *h* and *w* represent the height and width of the reference image in pixels, and $\varphi_{i,C}$ and $\theta_{i,C}$ are the angular spherical coordinates of point $p_{i,C}$. Use of the ceiling function should only be done if indexing at 1; floor can be used for languages indexing at 0. Since the index *i* has not changed for each point, the color of the pixel located in the image at $(column_i, row_i)$ is associated to the point in the LIDAR point cloud. Figure 3.1 shows an example of this fusion.

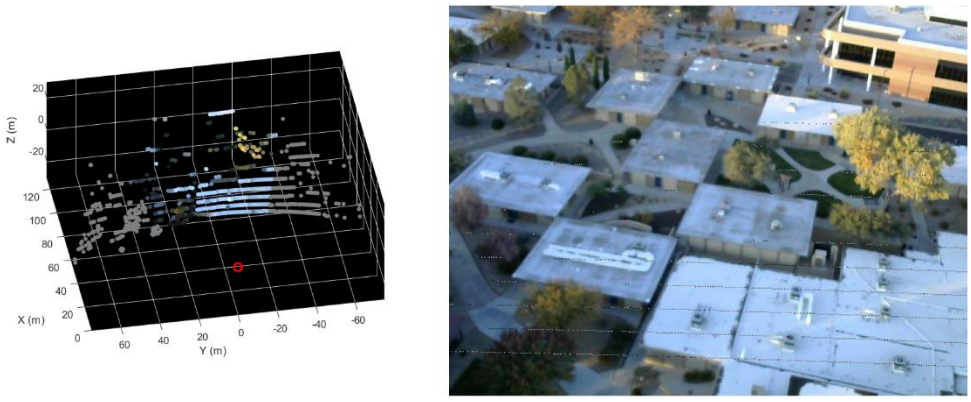

Figure 3.1: Example Point Cloud Image Fusion

The accuracy of this fused data set is dependent on several factors including accuracy of known coordinate transformations, camera distortion properties, camera resolution, and redundant overlap between LIDAR and image field of view.

4.0 Simulated Environment

4.1 Multirotor Simulation

Simulation is an essential tool when developing autonomous systems, especially for aerial platforms. However, a simulation is only as useful as its model is accurate. For that reason, an existing hexrotor dynamic simulation was created in Simulink and provided by Dr. Ken Bordignon and Dr. Iacopo Gentilini from their work in optimal UAS path planning [16] specifically for adaptation into this thesis. Their dynamic model and closed-loop control system provided the basis for the following simulation, which was significantly modified. The following section describes original experimentation in path definition and visual presentation.

The path prescribed to the hexrotor is defined by a series of waypoints, each prescribing position, velocity and acceleration in 3 dimensions. The path through these points is generated via cubic spline, following the boundary conditions. The trajectory planning system considers two consecutive points at a time [17]. Between these points, we specify a trajectory with continuous, differentiable position and velocity using the following cubic equation.

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \qquad \text{Equation 4.1}$$

This cubic equation is specified for *x* but can be expanded to each dimension independently. Using this equation between just two points will not allow for boundary conditions aside from position to be met. Instead, two additional intermediate waypoints must be specified. These waypoints do not have boundary conditions, instead they maintain continuity between multiple spline segments. Since there are effectively three consecutive pairs of waypoints, there are not three separate cubic spline equations that

contribute to a continuous piecewise trajectory. Consolidating these equations across all dimensions into a single diagonalized matrix system of equations results in the following equation.

$$Ax = b \qquad \text{Equation 4.2}$$

where A is a 12x12 matrix built from the components of *t*, *x* is a 12x1 vector containing each coefficient a_0 through a_3, and *b* is a 12x1 vector of boundary conditions. The number of columns of A is determined by the number of splines needed to span each pair of waypoints, in this case, four points require three lines. For each of these lines, four constants (a_0 through a_3) are required to constrain them. The number of rows of A is determined by the number of boundary conditions needed to define spline intersections and endpoints. The multiplication of Ax provides the system of equations for each of the splines; equal to each respective boundary condition. Since A is square and upper triangular, it has an inverse, provided *t* is increasing between each spline segment. Inverting A and pre-multiplying each side isolates the vector *x*, containing the constants for each spline segment.

To command the defined set of splines, at any time *t* between two waypoints, the desired position, velocity, and acceleration can be gathered from the derivatives of Equation 4.1, substituting the appropriate coefficients. Typically, the time at which each waypoint or intermediate waypoint is to be crossed is determined ahead of time. However, the time to completion can be estimated based on an average desired velocity and a distance between waypoints. The times for intermediate waypoints can be any distinct times between and not including the start and end times. The desired position,

velocity, and acceleration is used to create an error signal when subtracted from the current state which drives the input to the dynamic model.

The second modification of this simulation is a visual reference for the hexrotor's trajectory and attitude. This model utilizes basic surface geometry in MATLAB to create a wireframe representation of a hexrotor, as shown in Figure 4.1.

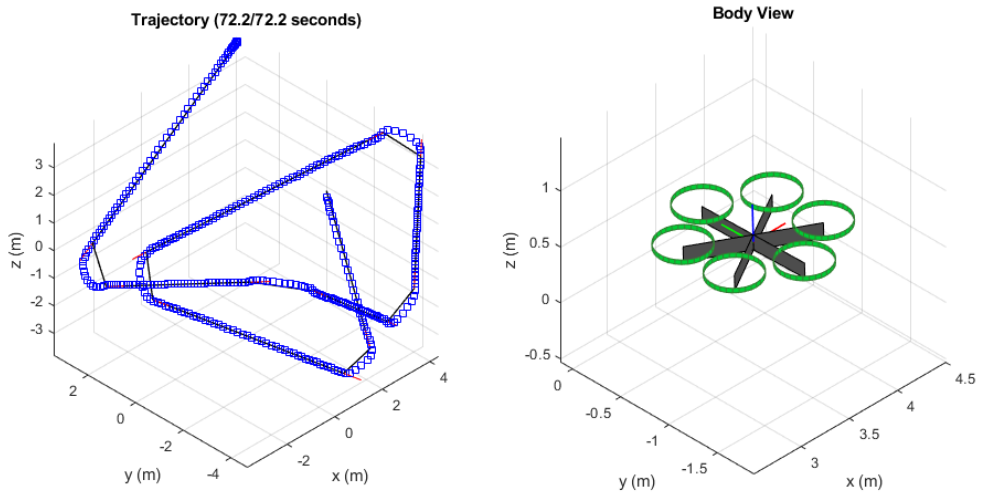

Figure 4.1: Hexrotor Trajectory and Orientation Visualization

On the left of Figure 4.1, the trajectory of the UAS is plotted in blue squares. The vertices of the underlying black line are the desired set of waypoints which have a red vector showing the direction of the desired velocity at that waypoint. Looking the path that this simulated hexrotor took around sharp corners, the smoothness of the cubic spline command can be seen. It is worth noting that the corners visible in the trajectory plot are cropped from their original position and are intentionally overshot. On the right, the orientation of the hexrotor is illustrated. The coordinate frame of the hexrotor is presented in red, green, and blue representing forward, left, and up, respectively. The trajectory and orientation are separated to avoid overloading the information within a single plot.

4.2 Obstacle Simulation

In addition to modifying the path definition and visual representation of the provided hexrotor simulation, a dynamic obstacle model has also been introduced. The trajectory truth model for simulating a dynamic obstacle is based on the recorded trajectory of the same multirotor model, but previously simulated. Dynamic obstacle detection is implemented via zero-mean Gaussian noise added to truth model. This emulates the noise in the point cloud segmentation detection caused by reducing an inconsistent region of points to their centroid.

Since the measurement received from the point cloud is a position, a state estimator is needed to estimate the future motion of the obstacle. To estimate this motion and reduce the noise on the position reading, a discretized Kalan Filter is used. "Discretized" in this case refers to the discretized state transition based on numerical integration.

The discretized Kalman filter begins with an initial state estimate x_0, defined by the first position reading of the obstacle, followed by zeros for the initial velocity, acceleration, and jerk. This model is adapted from a constant acceleration model, but since constant acceleration cannot be assumed for the dynamic obstacle, the state vector is expanded to include jerk. While constant jerk is then assumed, its process noise covariance is nonzero, meaning it is expected to abruptly change. The following Equations 4.1 through 4.6 define the constants and initial states that are used in this Kalman filter. While all three Cartesian directions are considered in the simulation, only the *x* direction is shown in this example.

$$\hat{x}_0 = \begin{bmatrix} x_0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{Equation 4.1}$$

$$F = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & \frac{1}{6}T^3 \\ 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{Equation 4.2}$$

$$P_0 = \begin{bmatrix} 0.3 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix} \quad \text{Equation 4.3}$$

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.00001 \end{bmatrix} \quad \text{Equation 4.4}$$

$$R = 0.04 \quad \text{Equation 4.5}$$

$$H = [1 \quad 0 \quad 0 \quad 0] \quad \text{Equation 4.6}$$

Here, $\hat{x}_k$ is the state estimate at time *k*, *F* is the discrete state transition matrix, P_0 is the initial state-estimate error covariance matrix, *Q* is the process noise covariance matrix, *R* is the measurement noise covariance matrix, and *H* is the measurement matrix. The measurement noise covariance is determined by estimating the noise present in obstacle detection based on the centroid location of a point cloud region. The state estimate update, as presented by Simon [18], is calculated as follows.

$$P_{k+1}^{-} = FP_k\hat{} + F^T + Q \quad \text{Equation 4.7}$$

$$K_k = P_{k+1}^{-}H^T(HP_{k+1}^{-}H^T + R)^{-1} \quad \text{Equation 4.8}$$

$$P_{k+1}^{+} = P_{k+1}^{-} - K_kHP_{k+1}^{-} \quad \text{Equation 4.9}$$

$$\hat{x}_{k+1} = F\hat{x}_k + K(y - HF\hat{x}_k) \quad \text{Equation 4.10}$$

Where P_{k+1}^{-} is the *a priori* estimate error covariance matrix, K_k is the Kalman gain matrix, P_{k+1}^{+} is the *a posteriori* estimate error covariance matrix, and *y* is the measurement of *x* plus some gaussian zero-mean noise. Continuing this cycle calculates each subsequent state estimate. A plot of this for all three cartesian dimensions is shown in Figure 4.2.

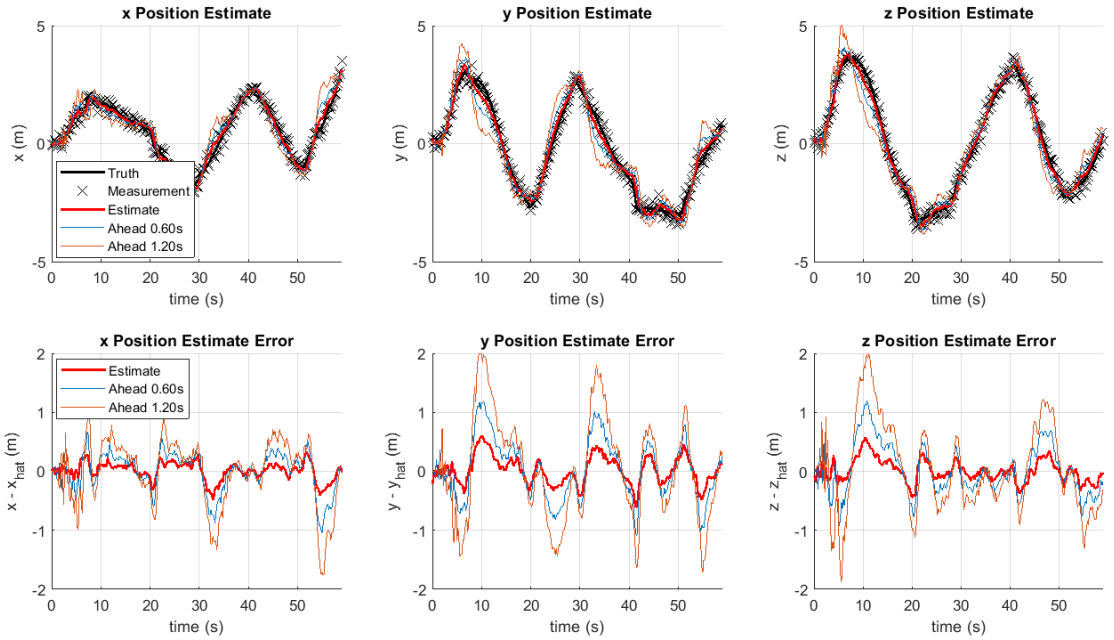

Figure 4.2: Example Kalman Filter Output

In the upper half of the Figure 4.2, the true position of the dynamic obstacle is given as a solid black line. The measurements recorded from that model added with some Gaussian noise are represented as x's. The red solid line is the state estimate at the current timestamp. By propagating the state forward (removing the second right-hand term in Equation 4.10), the predicted path can be plotted against the actual path. The bottom half of the figure shows the error with respect to the original signal. Overall, the error increases the further the prediction is placed in the future, which makes sense intuitively, and is shown by the growth of *P* in Equation 4.7.

5.0 Test Procedures

Sensor suite test scenarios progressed from static bench testing to mobile aerial testing to test various functional aspects. The preparation, goals, and outcomes of each of the scenarios is presented in this section.

5.1 Static Tests

Static tests were performed by mounting the LIDAR on a tripod with an external power supply. At this time, the LIDAR was the only sensor considered for testing due to prioritization of implementing the segmentation method. Static tests were performed both indoors on the bench as well as in the field. Static testing served two purposes; first to verify data acquisition was functional, and second to observe the point cloud representations of various obstacles.

Bench Tests

The first of the static tests was performed indoors, as shown in Figure 5.1.

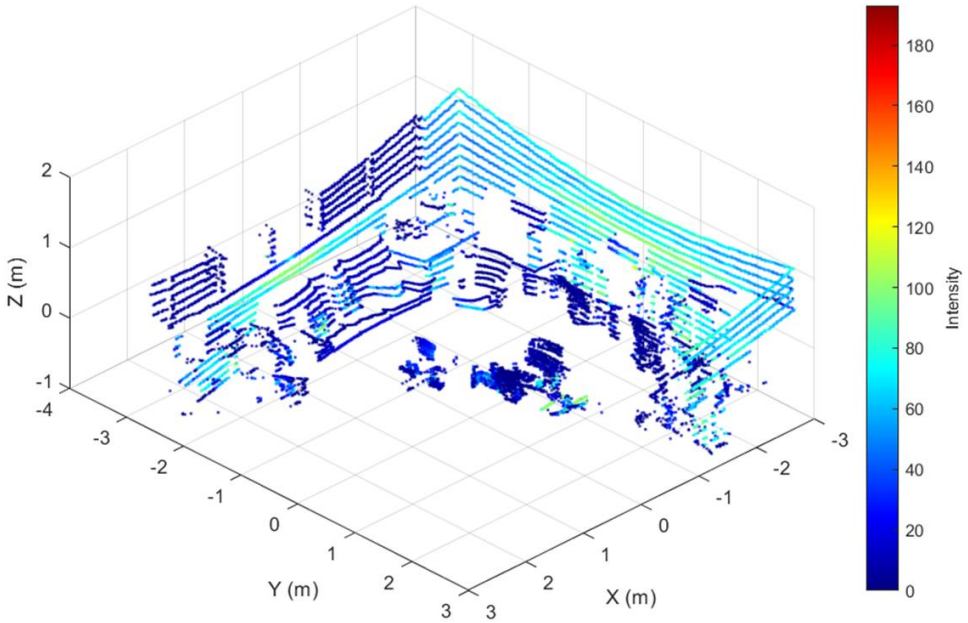

Figure 5.1: Example LIDAR Scan of Room

The color of points in Figure 5.1 indicates object reflectivity. From this test, it was observed that static objects between 1 and 10 meters of the LIDAR have a distinct appearance thanks to the relatively high point density (excluding objects outside of the field of view). This data set was also used to test the point cloud segmentation method, the results of which are shown in Figure 5.2.

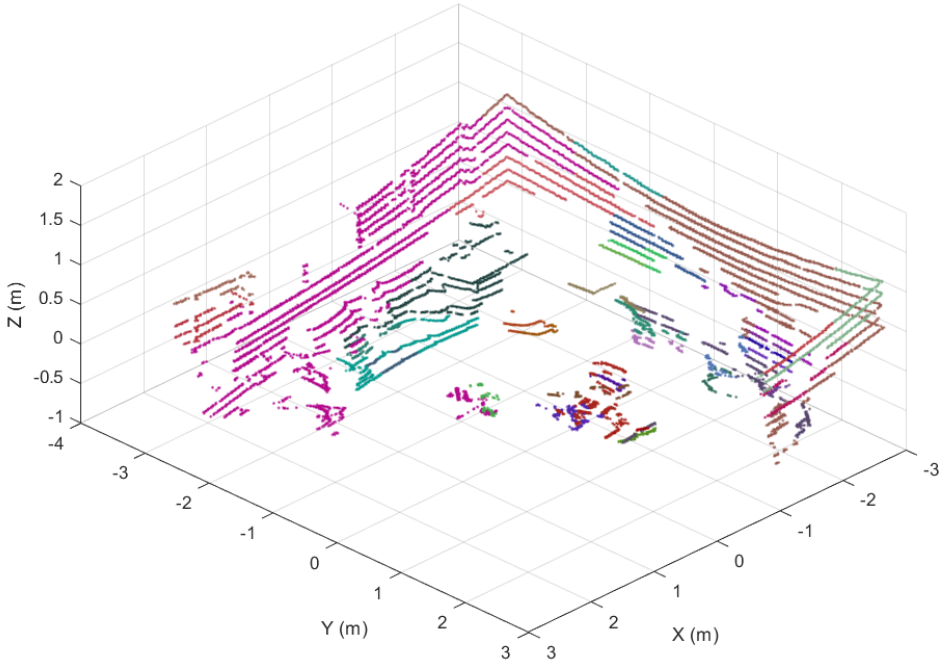

Figure 5.2: Example Segmented Point Cloud

It was discovered here that the segmentation method is not ideal for point clouds without predominantly flat surfaces. This makes sense as this method was originally designed for use on dense point clouds of buildings.

Field Tests

Static field tests were performed with the intent of determining how consistently various UAVs appeared in LIDAR point clouds. All testing was performed under either FAA Part 101 or Part 107. The drones tested using this method are consolidated in the Table 5.1.

Table 5.1: Flyby Test UAVs

UAV	Diagonal Span (cm)	Height (cm)	Maximum Consistent Capture Distance (m)
RYZE Tech Tello **[19]**	13.4	4.1	--
DJI Mavic **[20]**	40.2	8.4	4.2
DJI Phantom 4 **[21]**	35.0	8.9	6.5

The Tello, the smallest of the UAVs, presented an issue in that it was small enough to be undetectable at sub 3-meter range from the sensor. Even with the medium-scale Mavic and Phantom, the distance for consistent capture is still close to the sensor compared to its maximum readable distance. Based on the geometry of the LIDAR's scan, the body height is the primary factor for consistent detectability.

5.2 Ground Mobile Tests

The first of the mobile tests were performed on the ground, using a car LIDAR mount. For safety reasons, this test scenario was used to emulate dual drone flight. From these tests, it was discovered that there was not a considerable difference between stationary and mobile LIDAR in terms of visibility of UAVs. Figure 5.3 shows a Mavic 2 being detected during this test.

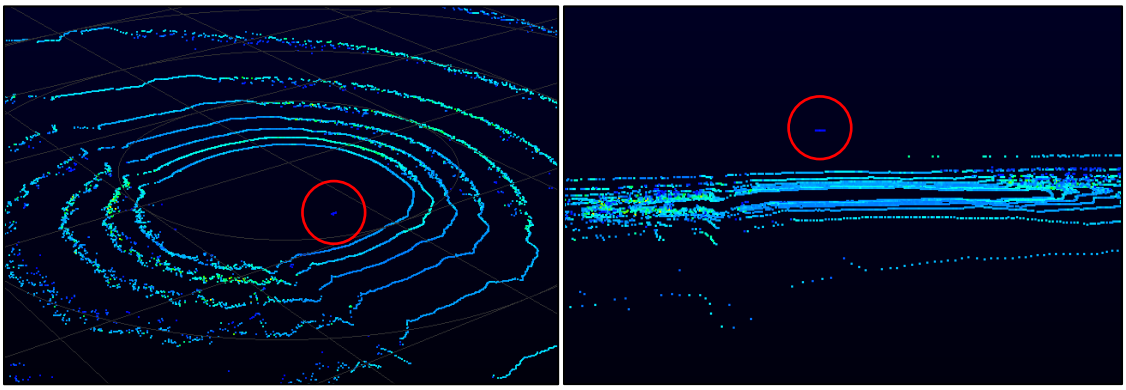

Figure 5.3: Mobile Testing Point Cloud

In Figure 5.3, the same point cloud is shown at two different angles. As indicated by the red circles, the Mavic 2 was able to be captured by the LIDAR in motion.

5.3 Aerial Mobile Tests

The final tests for this project were performed with the complete sensor suite mounted on the ALTA 6 flying over an urban setting. These tests were performed primarily to observe the quality of data gathered at low altitudes (<80 meters AGL). Figure 3.1 is an example of this data set. From these tests, it was noticed that the visual camera was less limited in returns based on distance than the LIDAR. As a result, visual imaging could likely be used for further-ranged detection if needed by this system.

5.4 Visual Camera Linearity Tests

All lensed visual cameras naturally possess some level of image distortion. This negatively effects the accuracy with which images and point clouds are registered. To mitigate this, the distortion can be characterized and then compensated. There are many existing camera calibration tools, in this case MATLAB's *cameraCalibrator* tool was used [22].

The *cameraCalibrator* tool accepts images of checkerboard patterns which provide references of straight lines. Because of distortion, these lines will not appear perfectly straight. The two basic image distortion forms are pincushion and barrel distortion, which cause pixels in the image to appear compressed toward the center of the image or expanded out from the center, respectively. Figure 5.4 shows examples of pre-corrected and post-corrected checkerboard detection images from the visual camera.

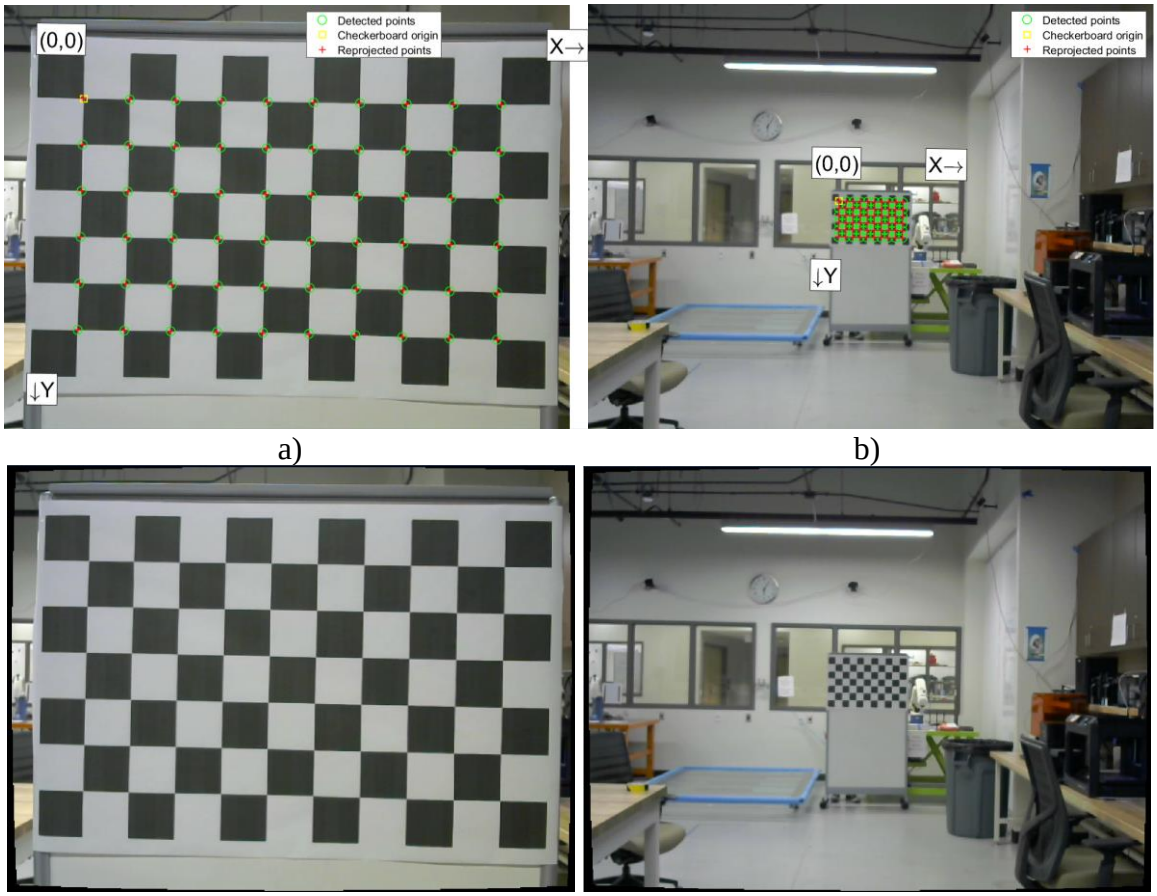

Figure 5.4: Image Distortion Correction
a) Original Image at 1 meter b) Original Image at 8 meters
c) Corrected Image at 1 meter d) Corrected Image at 8 meters

In the Figure 5.4, the corrected images draw pixels away from the edges and toward the center, implying the image originally had pincushion distortion. These images are from a set of 95 with increasing distance from 1 to 8 meters. The reprojection error in a checkerboard image is the average movement required to align perceived and expected checkerboard corners. Figure 5.5 shows the comprehensive reprojection error in the complete set of images.

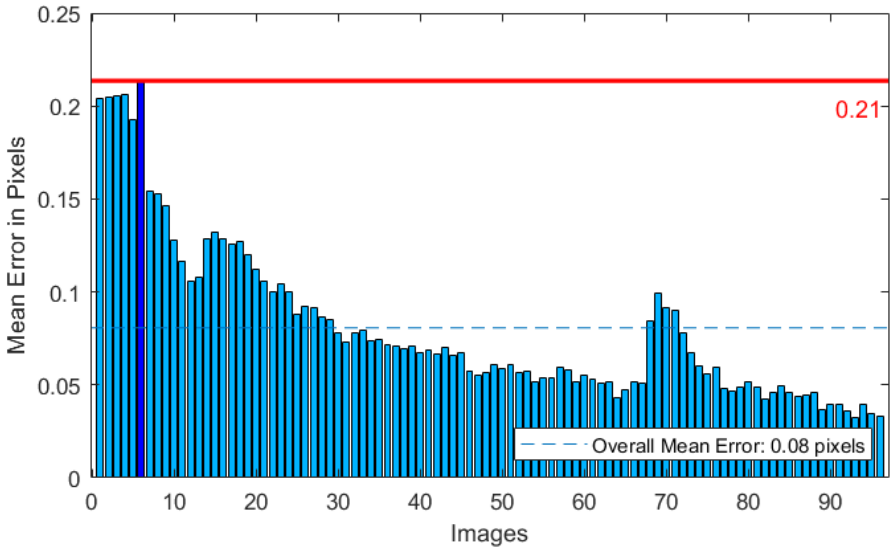

Figure 5.5: Average Reprojection Error by Image

The general trend shown in the above figure is that reprojection error decreases as distance increases. In actuality, it is more accurate to say that distortion in the center of the image is less severe than at the image edges, which was expected. While this test is useful in determining image distortion, it is also useful to know if the size of an observed object in the camera is inversely proportional to its distance from the camera. This is known as the linear region of the camera and it can be found by measuring the length of a checkerboard square in images taken at known distances. Figure 5.6 shows the relationship between distance from camera and the inverse of checkerboard pixel length.

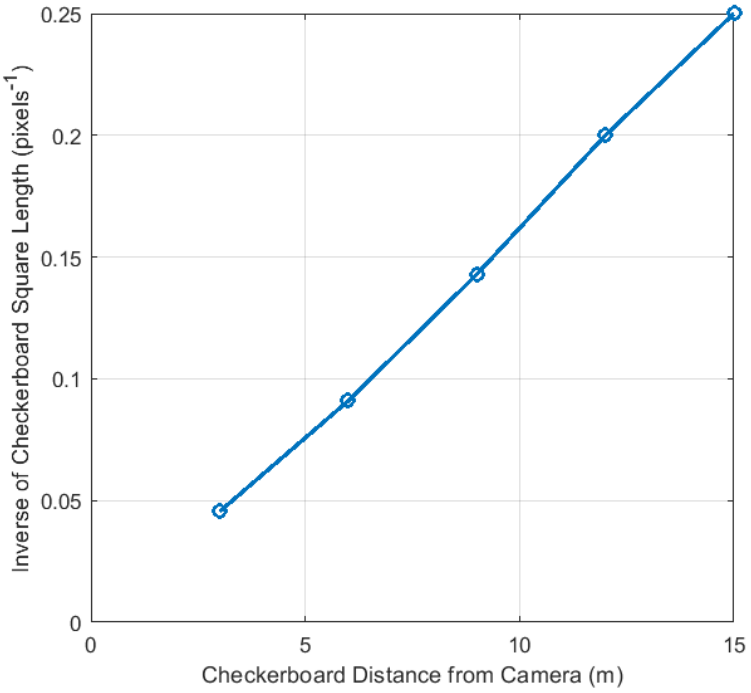

Figure 5.6: Camera Linear Region

As shown in Figure 5.6, between 3 and 15 meters, the relationship between distance from camera is inversely proportional to checkerboard square length. While the exact distance of the shift between the nonlinear and linear regions is not known precisely, it can be assumed to be under 3 meters. This verifies that fusion of point cloud points farther than 3 meters away is not influenced by nonlinear camera effects.

6.0 Analysis

The consideration of this specific sensor suite for use in detection and avoidance of dynamic obstacle in unknown urban environments shows some promise but requires significant development before implementation.

The utilization of voxel-based point cloud segmentation methods is not applicable to scnareios evhibiting sparse point clouds with minimal flat surfaces. Theoretically, fused LIDAR and visual imaging data sets could provide a different route for segmentation that could yield stronger results. By considering similar color between voxels as candidacy for region growth, the computational demand of normal and residual calculation can be circumvented. This may provide faster and more accurate region definition in sparse or non-primarily planar point clouds.

Dynamic obstacle prediction via a discretized Kalman filter allows for preemptive determination of potential collisions but is entirely reliant on a consistent tracking method. The lack of consistent range data of smaller UAVs at safe distances is concerning for the proposed detection system.

7.0 Conclusion and Recommendations

In this thesis, UAS SAA topics of obstacle detection, path planning, and dynamic obstacle avoidance are explored through simulation and post-processing of real-world data. Point cloud segmentation is found to be a method for obstacle avoidance that would benefit from fused point cloud and visual camera image data. A preliminary method for this fusion is described, involving pixel fusion via coordinate transformations and alignment of fields of view. Avoidance strategies for detected dynamic obstacles are explored via trajectory planning and potential field methods, but more optimal alternatives such as trajectory modification are discussed. While the scope of this thesis is relatively broad, the assessment of various urban UAS SAA aspects is largely compartmentalized. For further development of such systems, the compliance between each of the components—sensing, detection, command, and control—must be considered in greater depth. Specific concepts such as independent localization, static obstacle avoidance, and multiple dynamic obstacle avoidance are topics that were not explored in this work but would be necessary for thorough development of independent urban UAS SAA.

8.0 Future Work

8.1 Fusion of LIDAR and EO/IR and Obstacle Detection

The approached method for LIDAR and EO/IR fusion could be replaced with feature-level registration and consolidate the obstacle detection process. Instead of basing detection on fused point cloud segmentation, features could be extracted from the image and given depth via registered point cloud measurements.

8.2 Path Planning

To improve upon the avoidance algorithm explored in this thesis, a dynamic trajectory could provide a smoother response to dynamic obstacles. One way to do this would be to adjust the set of waypoints mid-flight as avoidance scenarios are encountered. This modification could be limited to incorporating new waypoints so that the path of the UAS does not intersect with observed obstacles but will also maintain its originally prescribed waypoint set as best as possible.

8.3 Experimental Data and Processing

The capture method utilized in testing scenarios could benefit from a more automated approach. For instance, having the capture begin immediately on Raspberry Pi 3B+ startup would have simplified data capture and avoided the need for field displays and keyboards. Further, the methods by which data was processed can be improved significantly if the capture and processing were performed in the same environment. Adapting the system entirely within ROS would allow for a centralized system that would have greater potential of utility in real-time.

REFERENCES

[1] P. Patrón and D. M. Lane, "Adaptive mission planning: the embedded OODA loop," in *3rd SEAS DTC Technical Conference*, Edinburgh, Scotland, 2008.

[2] M. Révay and M. Líška, "OODA Loop in Command & Control Systems," Armed Forces Academy of Gen. M. R. Štefánik, 2017.

[3] U.S. Department of Transportation Federal Aviation Administration, "Introduction to TCAS II Version 7.1," 2011.

[4] E. Burgos and S. Bhandari, "Potential Flow Field Navigation with Virtual Force Field for UAS Collision Avoidance," in *International Conference on Unmanned Aircraft Systems*, Arlington, VA, 2016.

[5] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, 1991.

[6] X. Yu and Y. Zhang, "Sense and avoid technologies with applicaitons to unmanned aircraft systems: Review and prospects," *Progress in Aerospace Sciences*, vol. 74, pp. 152-166, 2015.

[7] D. Chen and G. Gao, "Probabilistic Graphical Fusion of LiDAR, GPS, and 3D Building Maps for Urban UAV Navigation," *Navigation*, vol. 66, no. 1, pp. 151-168, 2019.

[8] A. F. Scannapieco, A. Renga, G. Fasano and A. Moccia, "Experimental Analysis of Radar Odometry by Commercial Ultralight Radar Sensor for Miniaturized UAS," *Journal of Intelligent & Robotic Systems,* vol. 90, no. 3, 2018.

[9] J. Pestana, J. Sanchez-Lopez and I. Mondragón, "A General Purpose Configurable Controller for Indoors and Outdoors GPS-Denied Navigation for Multirotor Unmanned Aerial Vehicles," *Journal of Intelligent & Robotic Systems,* vol. 73, no. 1, pp. 387-400, 2014.

[10] Velodyne Lidar, "Puck LITE™," Velodyne Lidar, 2019. [Online]. [Accessed 2019].

[11] Logitech, "Logitech C270 HD Webcam," 2019. [Online]. Available: https://www.logitech.com/en-us/product/hd-webcam-c270. [Accessed 2019].

[12] VectorNav, "VN-200," 2019. [Online]. Available: https://www.vectornav.com/products/vn-200. [Accessed 2019].

[13] Open Source Robotics Foundation, "ROS Wiki," 2018. [Online]. Available: https://wiki.ros.org/. [Accessed 2019].

[14] Freefly Systems, "ALTA 6 Specs," 2019. [Online]. Available: https://freeflysystems.com/alta-6/specs. [Accessed 2019].

[15] A. Vo, L. Truong-Hong, D. Laefer and M. Bertolotto, "Octree-based region growing for point cloud segmentation," *ISPRS Journal of Photogrammetry and Remote Sensing,* vol. 104, pp. 88-100, 2015.

[16] K. Vicencio, T. Korras, K. Bordignon and I. Gentilini, "Energy-Optimal Path Planning for Six-Rotors on Multi-Target Missions," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hamburg, Germany, 2015.

[17] B. Siciliano, L. Sciavicco, L. Villani and G. Oriolo, "Trajectory Planning," in *Robotics Modelling, Planning and Control*, Springer, 2009, pp. 161-167.

[18] D. Simon, Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches, Wiler-Interscience, 2006.

[19] Ryze Robotics, "Tello," 2019. [Online]. Available: https://www.ryzerobotics.com/tello. [Accessed 2019].

[20] DJI, "DJI Mavic 2," 2019. [Online]. Available: https://www.dji.com/mavic-2. [Accessed 2019].

[21] DJI, "DJI Phantom 4," 2019. [Online]. Available: https://www.dji.com/phantom-4. [Accessed 2019].

[22] MathWorks, "Single Camera Calibrator App," 2019. [Online]. Available: https://www.mathworks.com/help/vision/ug/single-camera-calibrator-app.html. [Accessed 2019].

APPENDIX

MATLAB Source Code

Voxelized Point Cloud Segmentation

```
% VoxelSegmentation
% Jonathan Buchholz
% Summer 2019
% ERAU ICARUS

%==========PARAMETERS==========%
% Test Parameter Set
% Point cloud file
scan_file = 'testscan3.csv';
% Voxel grid creation
radius = 70; % range for point consideration from origin (m)
res = 1; % initial voxel grid resolution (m)
max_divisions = 4; % number of allowable voxel octant divisions
min_residual = 0.0005; % amount of allowable voxel "noise" without
division
% Region growth
allow_normal_drift = false; % compare either recent neighbor normal
(true) or initial seed normal (false)
r_th = 0.0005; % residual threshold for considering voxels for seeding
t_th = 0.97; % normal alignment for considering voxels
pl_th = 0.2; % planarity threshold of centroids perpendicular to normal
minimumPoints = 50; % minimum number of points for a valid region
%==============================%

Scan = importdata(scan_file);
numPoints = length(Scan.data(:,1));
min_resolution = res/(2^max_divisions); % minimum reduced resolution
(m)
sSquare = round(radius*2/res); % number of elements along each side of
occupancy grid
voxelGrid = cell(sSquare,sSquare,sSquare);

% Organize points into voxel bins
maxRet = 1;
% j = 1;
% k = 1;
min_xidx = 2*radius;
max_xidx = 1;
min_yidx = 2*radius;
max_yidx = 1;
min_zidx = 2*radius;
max_zidx = 1;
for i = 1:numPoints

    % Limit returns to within box boundry
    xidx = ceil((radius + Scan.data(i,1))/res);
    yidx = ceil((radius + Scan.data(i,2))/res);
```

```
    zidx = ceil((radius + Scan.data(i,3))/res);
    if xidx > sSquare || yidx > sSquare || zidx > sSquare || xidx < 1
|| yidx < 1 || zidx < 1
        continue;
    end

        % max and min index bounds (for cropping empty cells)
        % x
        if xidx < min_xidx
            min_xidx = xidx;
        end
        if xidx > max_xidx
            max_xidx = xidx;
        end
        % y
        if yidx < min_yidx
            min_yidx = yidx;
        end
        if yidx > max_yidx
            max_yidx = yidx;
        end
        % z
        if zidx < min_zidx
            min_zidx = zidx;
        end
        if zidx > max_zidx
            max_zidx = zidx;
        end

    if isempty(voxelGrid{xidx,yidx,zidx})
        voxelGrid{xidx,yidx,zidx}.returns = 1;
        voxelGrid{xidx,yidx,zidx}.points = Scan.data(i,:);
    else
        voxelGrid{xidx,yidx,zidx}.returns =
voxelGrid{xidx,yidx,zidx}.returns + 1;
        voxelGrid{xidx,yidx,zidx}.points =
[voxelGrid{xidx,yidx,zidx}.points; Scan.data(i,:)];
        if voxelGrid{xidx,yidx,zidx}.returns > maxRet
            maxRet = voxelGrid{xidx,yidx,zidx}.returns;
        end
    end
    if ~isfield(voxelGrid{xidx,yidx,zidx},'location')
        voxelGrid{xidx,yidx,zidx}.location = [...
            xidx*res - radius - res/2;...
            yidx*res - radius - res/2;...
            zidx*res - radius - res/2];
    end
    if ~isfield(voxelGrid{xidx,yidx,zidx},'resolution')
        voxelGrid{xidx,yidx,zidx}.resolution = res;
    end

end

% Crop voxel grid to remove majority of empty voxels
voxelGrid([1:(min_xidx - 1), (max_xidx + 1):end],:,:) = [];
voxelGrid(:,[1:(min_yidx - 1), (max_yidx + 1):end],:) = [];
voxelGrid(:,:,[1:(min_zidx - 1), (max_zidx + 1):end]) = [];
```

```
[x_crop, y_crop, z_crop] = size(voxelGrid);

% Find voxel centroid, normal, errors
for xx = 1:x_crop
    for yy = 1:y_crop
        for zz = 1:z_crop

            % Find occupied voxels
            if ~isempty(voxelGrid{xx,yy,zz})
                % Find voxels containing 3 or more points
                if min(size(voxelGrid{xx,yy,zz}.points)) >= 4
                    % Get voxel centroid, normal, and errors
                    voxelGrid{xx,yy,zz} =
getVoxelCNE(voxelGrid{xx,yy,zz});
                    % Reduce voxel size for large point errors
                    voxelGrid{xx,yy,zz} =
reduceVoxel(voxelGrid{xx,yy,zz},min_residual,min_resolution);

                else
                    voxelGrid{xx,yy,zz} = zeros(0,0);
                end
            end

        end
    end
end

% Sort by residual and find neighbors
ResList = sortResiduals(voxelGrid);
ResList = voxelNbyRL(ResList);


%% Region growing
A = ResList; % expendable voxel list
% r_th = min_residual; % residual threshold for seed list creation

Regions = cell(0);
i = 1; % Element of A to being next seed list;

j = 1;
while ~isempty(A)
    currentPoints = 0;
    currentRegion = cell(0);
    currentSeeds = cell(0);
    % Get smallest residual voxel remaining
    while isempty(A{i})
        i = i + 1;
        if i > length(A)
            break;
        end
    end
    if i > length(A)
            break;
    end
    v_min = A{i};
    A{i} = zeros(0);
```

```
    % If the smallest residual remaining is too great, stop region
growing
    if v_min.residual > r_th
        break;
    end

    % Assign voxel to seed list and start of region
    currentSeeds{1} = v_min;
    currentRegion{1} = v_min;


    % Look through each seed voxel's neighbors for matches
    success_ms = zeros(0);
    k = 0;
    while k ~= length(currentSeeds)
        k = k + 1;
        for m = currentSeeds{k}.neighbors
            % If that neighbor is not already part of a region, add to
            % current region
            if ~isempty(A{m})
                % Check angular alignment of neighboring normals
against
                % threshold
                if allow_normal_drift
                    % Recent neighbor seed normal comparison
                    t_allign =
abs(dot(currentSeeds{k}.normal,A{m}.normal));
                    nonplanarity = abs(dot(...
                        (currentSeeds{k}.centroid - A{m}.centroid)/...
                        norm(currentSeeds{k}.centroid -
A{m}.centroid),...
                        currentSeeds{k}.normal));
                else
                    % First seed normal comparison
                    t_allign =
abs(dot(currentSeeds{1}.normal,A{m}.normal));
                    nonplanarity = abs(dot(...
                        (currentSeeds{1}.centroid - A{m}.centroid)/...
                        norm(currentSeeds{1}.centroid -
A{m}.centroid),...
                        currentSeeds{1}.normal));
                end


                if t_allign >= t_th && nonplanarity < pl_th
                    % Add neighbor voxel to current region
                    currentRegion = [currentRegion; A(m)];
                    currentPoints = currentPoints +
length(A{m}.points(:,1));
                    % Add neighbor voxel to seed list if residual fits
                    % threshold
                    if A{m}.residual < r_th
                        currentSeeds = [currentSeeds {A{m}}];
                    end
                    % Erase neighbor from available set
                    A{m} = zeros(0);
                    success_ms = [success_ms, m];
```

```
                end
            end
        end
    end

    % Save region if it is sufficiently occupied
    if currentPoints >= minimumPoints
        Regions{j} = currentRegion;
        j = j + 1;
        % Do not consider this region's voxels in future region growth
        for p = success_ms
            ResList{p}.allocated = true;
        end
    end
end

%% Refinement
for rs = 1:length(Regions)
    SeedR = cell(0);
    for ss = 1:length(Regions{rs})
        % Search for boundary voxels
        if length(Regions{rs}{ss}.neighbors) < minNeighbors
            SeedR = [SeedR, {Regions{rs}{ss}}];
        end
    end
    added_points = zeros(0,13);
    ks = 0;
    while ks ~= length(SeedR)
        ks = ks + 1;
        for ms = SeedR{ks}.neighbors
            if ResList{ks}.allocated == false
                for ps = 1:length(ResList{ms}.points(:,1))
                    ds = (ResList{ms}.points(ps,1:3)'-
SeedR{ks}.centroid)'*SeedR{ks}.normal;
                    if abs(ds) < d_th
                        added_points = [added_points;
ResList{ms}.points(ps,:)];
                    end
                end
            end
        end
    end
    if ~isempty(added_points)
        Regions{rs} = [Regions{rs}; {added_points}];
    end
end

%% Plotting
% Residual list
ResListStats(ResList);
% Original scan and regions
figure(1);
clf;
hold on;
plotVeloScan(scan_file);
voxelRegionPlot(Regions);
xlim([-3,3]);
```

```
ylim([-4,3]);
zlim([-1,2]);
xlabel('X (m)');
ylabel('Y (m)');
zlabel('Z (m)');
% Original scan and occupied voxels
figure(2);
hold on;
plotVeloScan(scan_file);
xlim([-3,3]);
ylim([-4,3]);
zlim([-1,2]);
axis vis3d;
voxelGridPlot(voxelGrid, 'normal', maxRet, true);
```

Funciton to calculate centroids, normals, and residuals of individual voxels

```
function voxelStruct = getVoxelCNE(voxelStruct)
% Finds centroid, normal, errors, and residual of a voxel's points.
% Stores outputs in fields '.centroid', '.normal','.errors', and
% '.residual', respectively.

% Centroid
voxelStruct.centroid = mean(voxelStruct.points(:,1:3),1)';

% Normal
x_c = voxelStruct.points(:,1) - voxelStruct.centroid(1);
y_c = voxelStruct.points(:,2) - voxelStruct.centroid(2);
z_c = voxelStruct.points(:,3) - voxelStruct.centroid(3);
voxelStruct.normal = [...
    (sum(y_c.*z_c)*sum(x_c.*y_c)) - (sum(x_c.*z_c)*sum(y_c.*y_c));...
    (sum(x_c.*y_c)*sum(x_c.*z_c)) - (sum(x_c.*x_c)*sum(y_c.*z_c));...
    (sum(x_c.*x_c)*sum(y_c.*y_c)) - (sum(x_c.*y_c)*sum(x_c.*y_c))];
voxelStruct.normal = voxelStruct.normal/norm(voxelStruct.normal);

% Errors
voxelStruct.errors = zeros(0,0);
for i = 1:length(voxelStruct.points(:,1))
    voxelStruct.errors(i) = (voxelStruct.points(i,1:3)'-
voxelStruct.centroid)'*voxelStruct.normal;
end
voxelStruct.errors = voxelStruct.errors';

% Residuals
voxelStruct.residual =
sqrt(sum(voxelStruct.errors.^2)/length(voxelStruct.errors));

% Placeholder for region growth refinement
voxelStruct.allocated = false;

end
```

Discretized Kalman Filter Testing

```
% Descretized Kalman Filter Test in 3D

%% Settings

% File to read
simout_file = 'samplesimout6.mat';

% Estimator error covariance matrix
p0eec = 0.3; % initial position estimator error covariance
v0eec = 0.1; % "" velocity
a0eec = 0.2; % "" acceleration
j0eec = 0.1; % "" jerk

% R Matrix
sigmax = 0.2;
sigmay = 0.2;
sigmaz = 0.2;

% Q matrix (constant)
pQ = 0;
vQ = 0;
aQ = 0;
jQ = 0.00001;

% Measurements
measureValidRate = 1; % percentage of measurements that are not lost
(>0)

% Predictions
predictions = 2;
timeStBwPred = 3;
jetPred = jet(predictions);


%% Preparation
% Read path data
load(simout_file);
t_s = simout(:,7)';
x_s = simout(:,1)';
y_s = simout(:,2)';
z_s = simout(:,3)';
T = t_s(2) - t_s(1); % assumes constant sampling period
maxPoints = length(t_s);

csi_s = zeros(12,maxPoints);
y_ms = zeros(3,maxPoints);

% Estimator error covariance matrix
P_prep = [p0eec, 0, 0, 0;
    0, v0eec, 0, 0;
    0, 0, a0eec, 0;
    0, 0, 0, j0eec];

P = [P_prep,     zeros(4,8);
```

```
    zeros(4,4), P_prep, zeros(4,4);
    zeros(4,8),            P_prep];

% State vector
csi = [x_s(1) + normrnd(0,sigmax); 0; 0; 0;     y_s(1) +
normrnd(0,sigmay); 0; 0; 0;     z_s(1) + normrnd(0,sigmaz); 0; 0; 0];
% x, dx/dt, d2x/dt2, d3x/dt3, "y", "z"

% R Matrix
R = diag([sigmax^2, sigmay^2, sigmaz^2]); % covariance matrix (scalar
in 1D)

% Discretized state transition matrix (diagonalized)
F_setup = [1, T, 0.5*T^2, (1/6)*T^3;
    0, 1,       T,   0.5*T^2;
    0, 0,       1,         T;
    0, 0,       0,         1];
F = [F_setup,     zeros(4,8);
    zeros(4,4), F_setup,     zeros(4,4);
    zeros(4,8),             F_setup];

% Discretized state transition matrix for state prediction
F_p_setup = [1, (timeStBwPred*T), 0.5*(timeStBwPred*T)^2,
(1/6)*(timeStBwPred*T)^3;
    0, 1,       (timeStBwPred*T),   0.5*(timeStBwPred*T)^2;
    0, 0,       1,         (timeStBwPred*T);
    0, 0,       0,         1];
F_p = [F_p_setup,     zeros(4,8);
    zeros(4,4), F_p_setup,     zeros(4,4);
    zeros(4,8),             F_p_setup];

% Q matrix
Q = diag([pQ, vQ, aQ, jQ, pQ, vQ, aQ, jQ, pQ, vQ, aQ, jQ]);

% Measurement matrix
H = [1, 0, 0, 0,     0, 0, 0, 0,     0, 0, 0, 0;
    0, 0, 0, 0,     1, 0, 0, 0,     0, 0, 0, 0;
    0, 0, 0, 0,     0, 0, 0, 0,     1, 0, 0, 0]; % Measurement matrix

xPred = zeros(predictions,maxPoints + predictions - 1);
yPred = zeros(predictions,maxPoints + predictions - 1);
zPred = zeros(predictions,maxPoints + predictions - 1);
tPred = zeros(predictions,maxPoints + predictions - 1);

%% Kalman Filter
for i = 1:maxPoints

    % A priori estimator error covariance (time i-1)
    Pm = F*P*F' + Q;

    % Kalman gain matrix
    K = Pm*H'/(H*Pm*H' + R);

    % A posteriori estimator error covariance
    P = Pm - K*H*Pm;

    % Gather measurement
```

```
    if rand() <= measureValidRate
        v = [normrnd(0,sigmax);
            normrnd(0,sigmay);
            normrnd(0,sigmaz)];
        y = [x_s(i); y_s(i); z_s(i)] + v;

        y_ms(:,i) = y;

        % A posteriori state estimate
        csi = F*csi + K*(y - H*F*csi);
    else
        % Propagate without measurement
        csi = F*csi;
    end

    csi_s(:,i) = csi;

    % State predictions
    csi_p = csi;
    t_p = t_s(i);

    PredLegend = {};

    % Predictions based on state transition propagation
    for j = 1:predictions
        PredLegend{end + 1} = sprintf('Ahead %2.2fs',j*timeStBwPred*T);
%         F_p_setup = [1, (j*timeStBwPred*T), 0.5*(j*timeStBwPred*T)^2,
(1/6)*(j*timeStBwPred*T)^3;
%             0, 1,       (j*timeStBwPred*T),
0.5*(j*timeStBwPred*T)^2;
%             0, 0,       1,          (j*timeStBwPred*T);
%             0, 0,       0,          1];
%         F_p = [F_p_setup,     zeros(4,8);
%             zeros(4,4), F_p_setup,     zeros(4,4);
%             zeros(4,8),              F_p_setup];
%         F_pmod = F_p*diag(repmat([1, 0.8, 0.75, 0],[1,3]));
% %         F_pmod = F_p;
        csi_p = F^(timeStBwPred)*csi_p;
        t_p = t_p + timeStBwPred*T;
        if i>=(j*(timeStBwPred-1))
            xPred(j,i+((j)*(timeStBwPred-1))) = csi_p(1);
            yPred(j,i+((j)*(timeStBwPred-1))) = csi_p(5);
            zPred(j,i+((j)*(timeStBwPred-1))) = csi_p(9);
            tPred(j,i+((j)*(timeStBwPred-1))) = t_s(i);
        end

    end

end

%% Plotting
figure(1);
clf;

TotLegend = {'Truth', 'Measurement', 'Estimate'};

% x
```

```
subplot(2,3,1);
hold on;
plot(t_s,x_s,'k-','linewidth',1.5); % Truth
plot(t_s,y_ms(1,:),'kx','markersize',8); % Measurement
plot(t_s,csi_s(1,:),'r-','linewidth',1.5); % State Estimate
axis tight;
ylim([-5,5]);
grid on;
title('x Position Estimate');
xlabel('time (s)');
ylabel('x (m)');
legend(TotLegend,'location','SW');
subplot(2,3,4);
hold on;
plot(t_s,x_s - csi_s(1,:),'r-','linewidth',1.5);
axis tight;
ylim([-2,2]);
grid on;
title('x Position Estimate Error');
xlabel('time (s)');
ylabel('x - x_{hat} (m)');

% y
subplot(2,3,2);
hold on;
plot(t_s,y_s,'k-','linewidth',1.5); % Truth
plot(t_s,y_ms(2,:),'kx','markersize',8); % Measurement
plot(t_s,csi_s(5,:),'r-','linewidth',1.5); % State Estimate
axis tight;
ylim([-5,5]);
grid on;
title('y Position Estimate');
xlabel('time (s)');
ylabel('y (m)');
% legend(TotLegend);

subplot(2,3,5);
hold on;
plot(t_s,y_s - csi_s(5,:),'r-','linewidth',1.5);
axis tight;
ylim([-2,2]);
grid on;
title('y Position Estimate Error');
xlabel('time (s)');
ylabel('y - y_{hat} (m)');

% z
subplot(2,3,3);
hold on;
plot(t_s,z_s,'k-','linewidth',1.5); % Truth
plot(t_s,y_ms(3,:),'kx','markersize',8); % Measurement
plot(t_s,csi_s(9,:),'r-','linewidth',1.5); % State Estimate
axis tight;
ylim([-5,5]);
grid on;
title('z Position Estimate');
xlabel('time (s)');
```

```
ylabel('z (m)');
% legend(TotLegend);

subplot(2,3,6);
hold on;
plot(t_s,z_s - csi_s(9,:),'r-','linewidth',1.5);
axis tight;
ylim([-2,2]);
grid on;
title('z Position Estimate Error');
xlabel('time (s)');
ylabel('z - z_{hat} (m)');

ThreeDLegend = {'Truth','Estimate'};

if predictions > 0
    EstLegend = {'Estimate'};
    TotLegend = [TotLegend,PredLegend];
    EstLegend = [EstLegend,PredLegend];
    ThreeDLegend= [ThreeDLegend,PredLegend];


    %x
    subplot(2,3,1);
    hold on;
    plot(t_s,xPred(:,1:maxPoints)); % State Predictions
    legend(TotLegend,'location','SW');
    subplot(2,3,4);
    hold on;
    plot(t_s,x_s - xPred(:,1:maxPoints),'-');
    legend(EstLegend);

    %y
    subplot(2,3,2);
    hold on;
    plot(t_s,yPred(:,1:maxPoints)); % State Predictions
%     legend(TotLegend);
    subplot(2,3,5);
    hold on;
    plot(t_s,y_s - yPred(:,1:maxPoints),'-');
%     legend(EstLegend);

    %z
    subplot(2,3,3);
    hold on;
    plot(t_s,zPred(:,1:maxPoints)); % State Predictions
%     legend(TotLegend);
    subplot(2,3,6);
    hold on;
    plot(t_s,z_s - zPred(:,1:maxPoints),'-');
%     legend(EstLegend);

end

% subplot(2,3,4);
% hold on;
% plot(t_s,sigmax*ones(1,numel(t_s)),'k--');
```

```
% plot(t_s,-sigmax*ones(1,numel(t_s)),'k--');
%
% subplot(2,3,5);
% hold on;
% plot(t_s,sigmay*ones(1,numel(t_s)),'k--');
% plot(t_s,-sigmay*ones(1,numel(t_s)),'k--');
%
% subplot(2,3,6);
% hold on;
% plot(t_s,sigmaz*ones(1,numel(t_s)),'k--');
% plot(t_s,-sigmaz*ones(1,numel(t_s)),'k--');

figure(2);
clf;
hold on;
plot3(x_s,y_s,z_s,'k-','linewidth',1.5);
plot3(csi_s(1,:),csi_s(5,:),csi_s(9,:),'r-','linewidth',1.5);
legend('Truth','Estimate');
xlabel('x (m)');
ylabel('y (m)');
zlabel('z (m)');
axis equal;
view([-1,-1,1]);
grid on;

if predictions > 0
    plot3(xPred',yPred',zPred');

end
```

Point Cloud Image Fusion

```
function [UASxyz, PCcolors, ref_image] =
allignIMG2PC(PCdata,Rb_PC,ob_PC,IMGdata,Rb_IMG,ob_IMG,IMGFOV)
% Color a point cloud with correlated image given FOV parameters and
% transformations for image to point cloud frame

IMGsize = size(IMGdata);

PCx = PCdata.data(:,1)';
PCy = PCdata.data(:,2)';
PCz = PCdata.data(:,3)';

num_points = numel(PCx);

PCxyz = [PCx; PCy; PCz];

UASxyz = zeros(3,num_points);
PCcolors = 0.5*ones(3,num_points);

ref_image = IMGdata;

% Transform points to camera frame
for current_point = 1:num_points
```

```
    UASxyz(:,current_point) = Rb_PC\PCxyz(:,current_point) + ob_PC;
    IMGxyz = Rb_IMG*(UASxyz(:,current_point) - ob_IMG);
    IMGx = IMGxyz(1);
    % Skip points behind the camera
    if IMGx < 0
        continue;
    end
    IMGy = IMGxyz(2);
    IMGz = IMGxyz(3);
    IMGazimuth = wrapTo180(atan2d(IMGy,IMGx));
    IMGelevation = wrapTo180(atan2d(IMGz,sqrt(IMGx^2 + IMGy^2)));
%     disp([IMGazimuth, IMGelevation]);
    % Find points that land within camera field of view
    if IMGazimuth >= -IMGFOV(1)/2 && IMGazimuth <= IMGFOV(1)/2 &&...
            IMGelevation >= -IMGFOV(2)/2 && IMGelevation <= IMGFOV(2)/2
        PixelX = IMGsize(2) - (floor((IMGazimuth +
IMGFOV(1)/2)/IMGFOV(1)*IMGsize(2)));
        PixelY = IMGsize(1) - (floor((IMGelevation +
IMGFOV(2)/2)/IMGFOV(2)*IMGsize(1)));
        if sum(abs(double(IMGdata(PixelY,PixelX,:)) - cat(3,112, 112,
112)) > 35) == 3
            ref_image(PixelY,PixelX,1) = 255 -
IMGdata(PixelY,PixelX,1);
            ref_image(PixelY,PixelX,2) = 255 -
IMGdata(PixelY,PixelX,2);
            ref_image(PixelY,PixelX,3) = 255 -
IMGdata(PixelY,PixelX,3);
        else
            ref_image(PixelY,PixelX,1) = 0;
            ref_image(PixelY,PixelX,2) = 0;
            ref_image(PixelY,PixelX,3) = 0;
        end
        PCcolors(:,current_point) =
double(IMGdata(PixelY,PixelX,:))/255;
    end
end
```