# Drone-Created Art: from PNG to X,Y,Z

Caleb Carrigan, Anna Hylbert, Eric Monson, and Matt Taylor
*University of Illinois at Urbana-Champaign*

**The primary objective of this project was to utilize a Crazyflie drone to draw pictures, motivated by the usage of drones as a medium for art. This could be useful as a possible project idea for University of Illinois students taking the class "CS445: Computational Photography" or as a hardware demonstration for students taking "AE353: Aerospace Control Systems". Additionally, the project could be used as a demonstration at engineering open houses in order to motivate K-12 students to major in STEM. To develop this demonstration, a pipeline was created that included an edge detection model of a single input image to a black and white output image. The model output was then converted into drone move commands so that the drone flew in the specified path of the photo. Due to the importance of being able to measure the X, Y, and Z coordinates in order to draw clear pictures, a lighthouse positioning system and HTC Vive base stations were used to give accurate, X, Y, and Z coordinates for the drone. Additionally, a custom controller and observer were implemented with respect to the sensor inputs of the lighthouse positioning system. Finally, a LED ring was attached to the base of the drone and two cameras were used to take a long exposure image showing the picture that the drone drew out. While using the custom controller and observer, the drone drew out images that resembled the desired ones, but the images weren't perfect as the lines weren't straight and there were significant perturbations in parts of the picture. However, when using the default observer, the images were much more accurate, and with more time, the custom observer could have been tweaked to deliver outputs with similar accuracy to the default observer.**

## I. Nomenclature

$A$ = 9x9 Matrix of coefficients for space-state model
$a_z$ = z-axis linear acceleration of the drone (m/s$^2$)
$B$ = 9x4 Matrix of coefficients for space-state model
$C$ = 3x9 Matrix of coefficients for space-state model
$D$ = 3x4 Matrix of coefficients for space-state model
$G$ = Gaussian function use to blur images
$g$ = gravitational constant (m/s/s)
$H_x$ = Sobel Operator/vertical line detecting kernel
$H_y$ = Sobel Operator/horizontal line detecting kernel
$I$ = an image used in the Gaussian function
$k_f$ = force motor coefficient
$k_m$ = moment motor coefficient
$l$ = spar length of the drone (m)
$M$ = magnitude of the local gradient of an image
$o_x$ = x coordinate of the drone's position (m)
$o_y$ = y coordinate of the drone's position (m)
$o_z$ = z coordinate of the drone's position (m)
$P$ = x-direction gradient
$Q$ = 9x9 Matrix of coefficients used for LQR
$R$ = 4x4 Matrix of coefficients used for LQR
$r_x$ = weights given in the matrix R
$S$ = Input image convolved with Gaussian blur
$s$ = equations of motion for the drone
$u$ = 4x1 input vector for the space-state model
$v_x$ = x-component of the linear velocity of the drone (m/s)
$v_y$ = y-component of the linear velocity of the drone (m/s)
$v_z$ = z-component of the linear velocity of the drone (m/s)
$W$ = y-direction gradient
$W_c$ = controllability matrix
$W_o$ = observability matrix
$w_x$ = x-component of the angular velocity of the drone
$w_y$ = y-component of the angular velocity of the drone
$w_z$ = z-component of the angular velocity of the drone
$x$ = 9x1 state vector for the space-state model
$\dot{x}$ = time derivative of the state vector
$\hat{x}$ = estimate of the state vector
$\dot{\hat{x}}$ = time derivative of the estimate of the state vector
$y$ = 3x1 output vector of the space-state model
$\Gamma$ = angle of the gradient used in the Gaussian function
$\theta$ = roll of the drone (rad)
$\sigma$ = standard deviation
$\phi$ = pitch of the drone (rad)
$\psi$ = yaw of the drone (rad)
$\omega_x$ = x-component of the angular acceleration of the drone
$\omega_y$ = y-component of the angular acceleration of the drone
$\omega_z$ = z-component of the angular acceleration of the drone
$\nabla S$ = gradient of image

## II. Introduction

In this report, the development of a pipeline capable of accepting a random input image, and producing a long-exposure picture of this same image drawn in 2-D space by a Crazyflie drone is described. In order to achieve this, high levels of movement precision were obtained through the implementation of a linear state feedback controller and observer that provided the drone with rotor commands. Furthermore, drone tracking was accomplished by employing the Bitcraze Lighthouse positioning deck and a set of version 1.0 HTC Vive Base Stations (Figure 1).

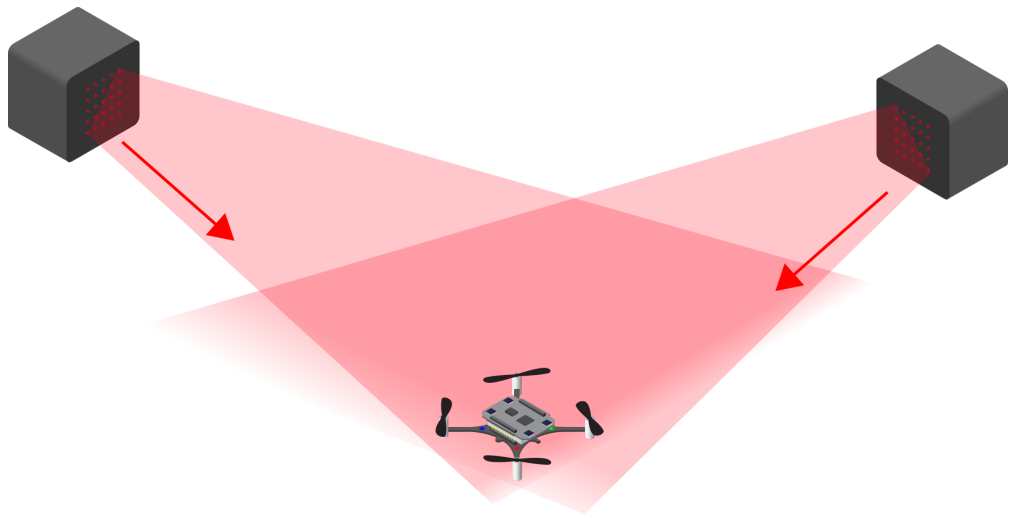


**Fig. 1 Lighthouse Positioning System [1]**

Due to the base stations only using sweep angles to measure the location of the drone, this method of tracking was useful as it allowed the drone to be flown in dark environments where the light output by the Bitcraze LED ring deck was easy to see in the long exposure images that were taken. The team decided to create such a pipeline to fulfill a number of different purposes, from serving as a hardware demo for other controls classes such as AE353 to helping provide a concrete example of the overlap between engineering and art. In any case, being able to have any image come to life in 3D space is an exciting application of modern technology.

Using a drone to 'draw' images in 3-D space is not a new idea. Drone light shows exist across the country to offer a replacement when fireworks are either too dangerous or too costly [2]. Long exposure imagery has also previously been used with drones. Chicago-based photographer Reuben Wu has used LED lights, drones, and long exposure images to capture creative images of natural landscapes [3]. However, in both these cases, the drone(s) can only draw a fixed number of images. In this project, a pipeline that converted images to lines that could be then drawn out by a drone was created in order to increase the robustness of the images that the drone could draw.

The following report outlines both the wide-angle view of the project and the specific details of the hardware and software required to make the proposed pipeline a reality. In this detailed description, explanations are given of some of the challenging concepts and theories applied throughout the project to highlight the team's understanding of controller and observer design. With art being a central theme of the project, a number of long-exposure images are included that demonstrate the efficacy of the pipeline. Finally, the limitations of the pipeline and how it could be improved upon in future research are discussed.

## III. Edge Detector

### A. Edge Detection Theory

Edge detection is not a new field of study in computer vision. In fact, one of the most common edge detection models to date was created in 1986 by a professor at UC Berkeley named John Canny [4]. In recent years, more robust models have been developed using deep learning [5], however these models are supervised in nature, meaning that the model must be trained on hundreds to thousands of images with corresponding labels of edge masks. Additionally, training on large amounts of data is often computationally expensive and takes time to fine-

tune. On the contrary, Canny edge detection utilizes a heuristic, providing an effective multi-step approach to finding the contours of an image. The list of steps below is a variant of the Canny Edge detection model described in Jain et al. [6].

- Smoothing (Gaussian Blur)
- Localized Gradient Calculations (Finite Difference)
- Non-maxima Suppression
- Double Thresholding and Hysteresis

Like the name suggests, Gaussian blur uses a Gaussian function to blur an image. It does this by taking the standard deviation, $\sigma$, essentially applying a Gaussian function to a target and area of neighboring pixels depending on the size of the standard deviation. A larger standard deviation pulls in color values from more pixels, creating a larger blur [6].

$$G[i, j, \sigma] = \frac{1}{2\pi\sigma^2} e^{-\frac{j^2+i^2}{2\sigma^2}} \tag{1}$$

An image, $I[i, j]$, can be convolved with (1) in order to blur an entire image as shown in (2) [6]. Note that $(\cdot)$ represents the convolution operator.

$$S[i, j] = G[i, j; \sigma] \cdot I[i, j] \tag{2}$$

After an image is properly blurred, localized gradients must be calculated. For the implementation in this project, the Sobel operator was used. Two 3x3 kernels shown in (3) drive the local gradient calculations in the X and Y directions. The output from (2) is convolved with the respective X and Y kernels in order to get (4), a gradient approximation. Taking a deeper dive into the x-direction gradient labeled as P, we can see how edges are actually detected! Imagine a simple image containing a thick square. The kernel $H_x$ contains fixed weights with the middle column having weights of 0. When the convolution operation $H_x \cdot S$ is performed, pixels to the left and right of a target pixel are subtracted from each other, giving an output that is a vertical line detector. In a similar fashion, the kernel $H_y$ is a horizontal line detector.

$$H_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad H_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{3}$$

$$\nabla S \approx \begin{bmatrix} P \\ W \end{bmatrix} = \begin{bmatrix} H_x \cdot S \\ H_y \cdot S \end{bmatrix} \tag{4}$$

The gradient $\nabla S$ now contains two separate images, one containing vertical edge detections and one containing horizontal edge detections. How do we find the true edges? We find the magnitude of gradient shown in (5). For example, a 45 degree line would contain components of horizontal and vertical edges, so taking the distance of the two would create an angled line. In addition to the magnitude of the gradient, the direction is also known. The angle of the gradient can be found by taking the inverse tangent of the X and Y components [6].

$$M[i, j] = \sqrt{P[i, j]^2 + W[i, j]^2} \tag{5}$$

$$\Gamma[i, j] = \tan^{-1}\left(\frac{W[i, j]}{P[i, j]}\right) \tag{6}$$

The raw magnitudes of the gradient often contain very thick edges, therefore non-maxima suppression is performed to thin the edge mask. Non maxima suppression uses the calculated angle of the gradient to break the magnitude down into 4 components [6]. Figure 2 shows a breakdown of these four components on a circle as gradient lines. Once a gradient line configuration is determined for a pixel, the two neighboring pixels along the gradient line are found and compared to the target pixel. If the target pixel is smaller than either of its neighbors, it is automatically set to a value of 0 (black) so that it is no longer considered an edge. This process is repeated for the entirety of the image to further thin previously thick edges.

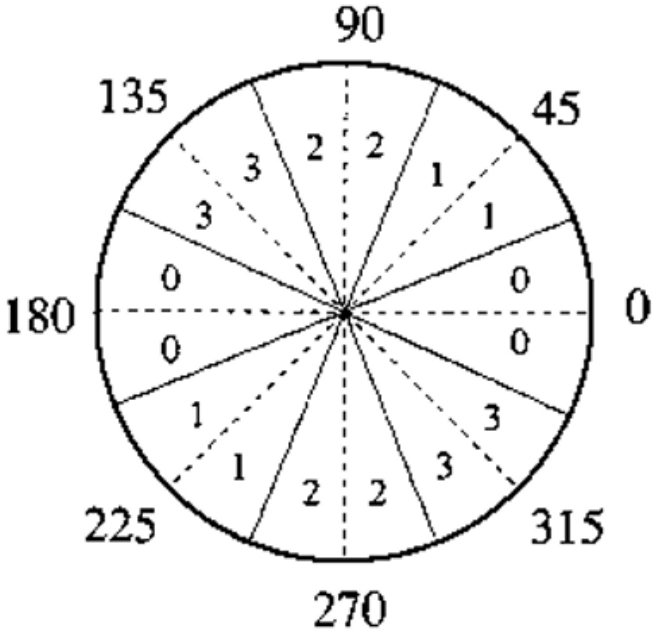



**Fig. 2 Non-maxima Suppression Orientations [6]**

Finally, double thresholding and hysteresis are performed to remove false detections of edges[6]. Figure 3 shows a high and low threshold along with four example edge detections. If a pixel contains a value that is below the low threshold, it is automatically discarded. For the example in the figure below, line D would be removed. If a pixel is in between the high and low thresholds with no connected components passing the high threshold, it is also discarded. For the example below, line E would be completely removed while lines A and CB would be rounded up to fully white (value of 255). Double thresholds tend to work better for edge detection models than traditional single thresholds because weak edges that are important to the image contours will still be considered as an edge. Sticking with the example below, a single threshold would remove 75% of line CB which is undesirable.

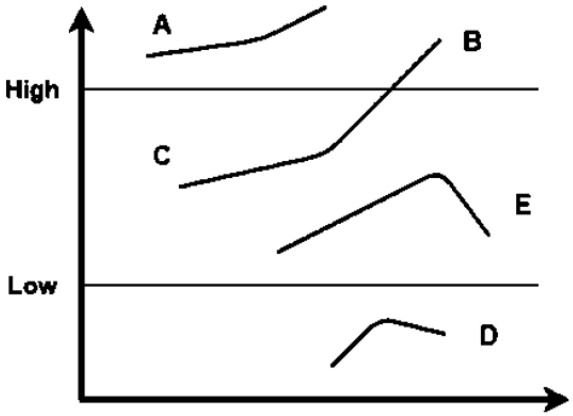



**Fig. 3 Double Thresholding Technique**

### B. Edge Detection Approach

One of the drawbacks of using heuristics instead of a trained deep learning model for edge detection is that there are hyper-parameters that must be tuned in order to achieve best results. As described in the edge detection theory section, Gaussian blur requires a parameter for varying the standard deviation, $\sigma$. A larger standard deviation results in an increased blurred effect. Hysteresis also requires low and high bounds for thresholds in order to remove false detections of edges. Lastly, the image resolution itself is a parameter that can be changed because there is a trade off between compute time and edge detection quality. Kernel convolutions, non-maxima suppression, and double thresholding all require each pixel individually to be looked at. In the case of simple photos such as circles, hexagons, or even the Google logo, a smaller image size commonly results in a similar quality of edge detection with a quicker run time. On the other hand, more complicated photos such as ones containing flowers and human-beings need the high resolution in order to detect fine edges. Unfortunately, this comes at the cost of a longer time to execute the program. Thus, images used in this report did not initially have accurate edge masks. An iterative approach was used to continuously test different hyper-parameters until a desired result is met.

### C. Parsing Edge Detection to Move Commands

The last crucial step in the edge detector is to find a generalized way to parse the edge mask into a list of ordered move commands. In the case of drawing images, 'ordered' means that move commands must be positioned to follow the lines of the edges themselves. A simple and efficient way to do this is to go pixel by pixel (left to right and top to bottom) in the image, adding every single white pixel's x and y position to a list. Unfortunately, there is one major drawback to this approach; there is no way of knowing that successive points in the list are adjacent to each other in the edge mask. Looking at Figure 4, we can see that scanning the edge mask from left to right will create a noisy and unordered list of move commands, forcing the drone to draw a point on the left side of the image and then a point much further away on the right half of the image. This is unacceptable when using long exposure imagery as there will be streaks of light all over the place! In addition, the drone would take a long time to fly in this inefficient path.

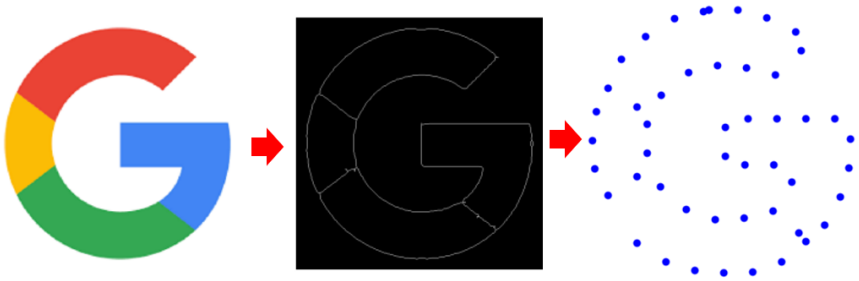


**Fig. 4 Stages of Edge Detection: Photo (left), Edge Mask (middle), and Move Commands (right)**

Our next approach was to use common shortest path algorithms like Dijkstra's algorithm, A* path finding [7], and the Chinese postman problem [8]. These algorithms take a graph theory approach where the shortest path must be found between two vertices. For A* and Dijkstra's algorithm, the path must be traversed exactly once. In the case of the Chinese postman problem, the goal is adjusted to finding the shortest path in a closed walk, allowing a path to be traversed more than once. In any case after much research, the team realized that the problem being solved doesn't require a shortest path algorithm at all. The goal is to find a path that reaches **all** white pixels. If a path finding algorithm isn't necessary, what is you may ask?

A much more convenient approach is to perform an image traversal. More specifically, a variant of a depth first search (DFS) traversal through the edge mask is used. Choosing depth-first instead of breadth-first is an extremely important design decision because DFS allows the path to be traced along neighboring edges. The depth first search algorithm is recursive in nature and must keep track of already visited pixels. An outline of the process is shown below in Algorithm 1. The move commands displayed in Figure 4 show a plot of the x and y positions of the DFS algorithm, which uses a post processing technique to center the data at the origin and minimize the number of move commands by only keeping track of x, y pairs at selected intervals.

**Algorithm 1** DFS Edge Parsing Algorithm

```
for each pixel do
    if pixel is white then recursively do the following:
        Check Base Case: pixel is visited
            exit
        mark pixel as visited
        add x and y position to list of move commands
        look at each neighbor, following steps 3-6
    end if
    if pixel is black then
        mark pixel as visited
    end if
end for
```

## IV. Lighthouse Positioning System

### A. Lighthouse Positioning Theory

Lighthouse positioning is made possible by the Bitcraze Lighthouse position deck and enables the Crazyflie drone to achieve high-precision flight. The system makes use of either HTC Vive base stations (Lighthouse V1) or SteamVR Base Station 2.0 (Lighthouse V2) to generate a field of precisely timed IR pulses and X/Y axis IR laser sweeps. The orientation of these sweeps are displayed with respect to the lighthouse in Figure 5.

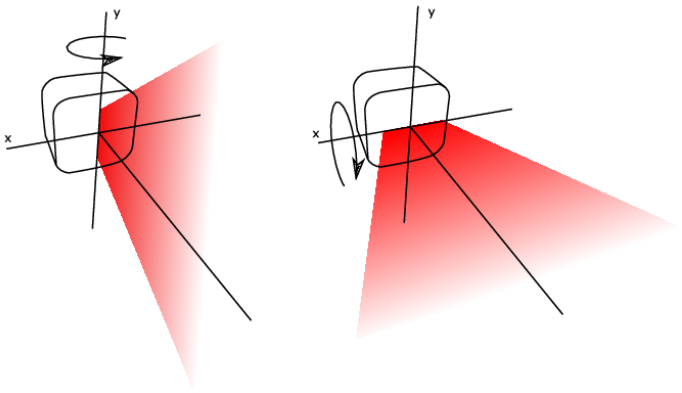



**Fig. 5 Lighthouse Sweep Orientations [9]**

*1. Recorded Data*

By recording when the sweeps are detected by the Lighthouse positioning deck, and by knowing the position of the base stations, the Crazyflie drone is able to place itself within the global reference frame. The Lighthouse positioning deck has four sensors that each record the angles at which the two light planes created by the IR laser sweeps intersect the sensor. Aside from the sixteen angles obtained from the light sensors, the Crazyflie needs two more pieces of system information to be able to estimate its position.

The first additional piece is calibration data of each base station used in the system, and the second, more important piece is base station geometry data. The calibration data for each base station accounts for minor imperfections in the manufacturing of the base stations, and is stored within each base station itself. The details of how the calibration data is applied to the raw sweep angles is very mathematically involved and is beyond the scope of this project. However, more information is available directly from Bitcraze [10].

The second additional piece of information is the geometry data that describes the position and orientation of the base stations. The geometry is important to understanding how the measured sweep angles translate to the global

reference frame, as it relates the local position of the individual light sensors to the fixed, known position of the base stations. The data is stored as the x, y, z position of each base station as well as a series of rotation matrices to relate the global reference frame to the local reference frame of the rotors inside the base station. Unlike calibration data, the geometry data must be re-calculated each time the base stations are moved or reoriented to update the global positioning data stored onboard the Crazyflie.

*2. Sweep Angles to Global Position*

With the sweep angles, calibration data, and base station geometry data passed to the Crazyflie, the drone has two methods of determining its actual position in the global reference frame. The first method provides a simple and robust solution, but requires two base stations to be visible to make use of the intersections of the IR laser sweeps. The second method passes the base station geometry and sweep angles directly to the Kalman estimator, where a series of rotation matrices are applied to back out rotation angle from sensor position. For this second method, one base station is sufficient to estimate the position, but more base stations would add precision and redundancy. A visual representation of these two methods is provided in Figure 6 below, with image A corresponding to the first method and image B corresponding to the second method.

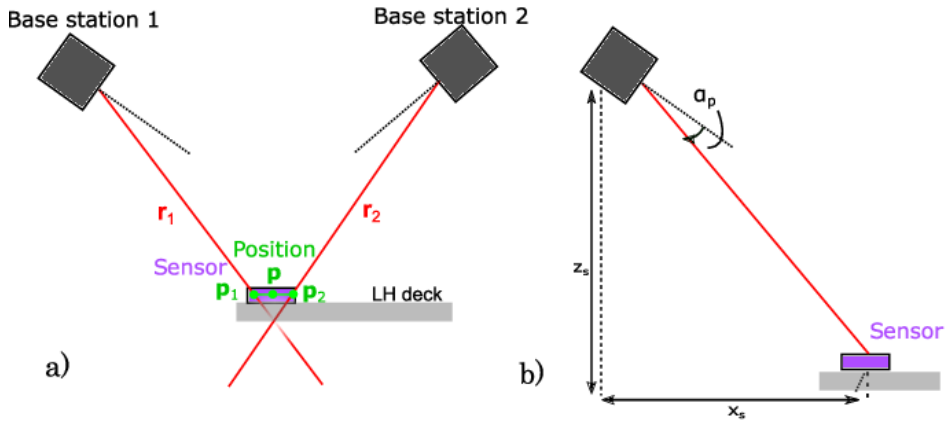



**Fig. 6 Methods of Determining Global Position from Lighthouse Data [9]**

The first method of determining global position will be referred to as the crossing beams method, as it utilizes the intersection point of two rays calculated from the measured sweep angles. More specifically, these two rays are determined using the formula detailed in Equation 7

$$\vec{A}_i = \begin{bmatrix} \sin\theta_{h,i} & -\cos\theta_{h,i} & 0 \end{bmatrix} \qquad \vec{B}_i = \begin{bmatrix} -\sin\theta_{v,i} & 0 & \cos\theta_{v,i} \end{bmatrix}$$
$$\vec{r}_i = [T_i] \begin{bmatrix} \dfrac{\vec{A}_i \times \vec{B}_i}{\left\| \vec{A}_i \times \vec{B}_i \right\|} \end{bmatrix} \tag{7}$$

where $\theta_{h,i}$ represents the angle obtained from the horizontal laser sweep, $\theta_{v,i}$ represents the angle obtained from the vertical laser sweep, $[T_i]$ represents the base station rotation matrix obtained from geometry data, and $\vec{r}_i$ represents the calculated ray rotated into the global reference frame. Equation 7 is used to generate the two rays pointing to each base station, with $i = 1$ representing base station 1 and $i = 2$ representing base station 2. In theory, these two beams should cross exactly at the point where the sensor is located, however, due to errors present in real life, the beams may not exactly meet. To solve this problem, the algorithm supplied by Bitcraze calculates the point that is closest to both beams instead and uses this as the estimated position. The first step in this calculation is to project rays $\vec{r}_1$ and $\vec{r}_2$ onto the two vectors representing the positions of the two base stations ($\vec{o}_1$ and $\vec{o}_2$). The projections are detailed in Equation 8:

$$\vec{w} = \vec{o}_1 - \vec{o}_2$$
$$a = \vec{r}_1 \cdot \vec{r}_1 \qquad b = \vec{r}_1 \cdot \vec{r}_2 \qquad c = \vec{r}_2 \cdot \vec{r}_2 \qquad d = \vec{r}_1 \cdot \vec{w} \qquad e = \vec{r}_2 \cdot \vec{w} \tag{8}$$

Next, the magnitudes of the vectors pointing from each base station to the light sensor on the Lighthouse positioning deck are determined and multiplied by the two previously calculated rays ($\vec{r_1}$ and $\vec{r_2}$). Adding the respective base station position to these two vectors yields the estimated global position of the light sensor according to each base station. This process is outlined in Equation 9 below:

$$\begin{aligned} \vec{R_1} &= \frac{b \cdot e - c \cdot d}{a \cdot c - b \cdot b} \cdot \vec{r_1} \quad \rightarrow \quad \vec{P_1} = \vec{R_1} + \vec{o_1} \\ \vec{R_2} &= \frac{a \cdot e - b \cdot d}{a \cdot c - b \cdot b} \cdot \vec{r_2} \quad \rightarrow \quad \vec{P_2} = \vec{R_2} + \vec{o_2} \end{aligned} \tag{9}$$

where $R_i$ represents the vectors pointing from each base station to the light sensor, $a$ represents the projection of $\vec{r_1}$ onto itself, $b$ represents the projection of $\vec{r_1}$ onto $\vec{r_2}$, $c$ represents the projection of $\vec{r_2}$ onto itself, $d$ represents the projection of $\vec{r_1}$ onto the vector pointing from $\vec{o_1}$ to $\vec{o_2}$, and $e$ represents the projection of $\vec{r_2}$ onto the vector pointing from $\vec{o_1}$ to $\vec{o_2}$.

Finally, as mentioned previously, since these calculations are performed in real life where inaccuracies are present, the two positions obtained are not identical. As a result, the point halfway between the two calculated sensor positions is selected as the true sensor position (Equation 10), with the distance in meters between the two points stored and logged as $\delta$. Note in Equation 10, $j$ represents the particular light sensor for which the position is being calculated. In the case of the Bitcraze Lighthouse position deck, there are four light sensors, arranged in a rectangular pattern, so $j \in [1, 4]$, with $\vec{P_j}$ representing the global position of sensor $j$.

$$\begin{aligned} \vec{P_j} &= \frac{\vec{P_{1,j}} + \vec{P_{2,j}}}{2} \\ \delta &= \vec{P_{1,j}} - \vec{P_{2,j}} \end{aligned} \tag{10}$$

The other method of determining global position from the lighthouse sensor data will be referred to as the raw sweeps method. The working principle behind the raw sweeps approach is the fact that each light sensor must be located in the plane that is defined by the base station geometry and sweep angle. Using this principle, it is possible to determine the position of each light sensor with the geometry data and sweeps of a single base station. The explicit mathematical measurement model required for this approach involves a number of rotation matrices and complex vector calculations which can be explored further on the Bitcraze website [11].

**B. Lighthouse Positioning Approach**

Of the two methods described in Section IV.A, the crossing beams approach was selected to determine the position of the drone in the global reference frame. This decision was made based on the availability of two base stations and the simpler mathematics behind the crossing beams approach which would aid in implementing the lighthouse positioning system with a custom controller and observer.

However, while the theory behind the crossing beams method was well understood, it was still necessary to actually implement the algorithm onboard the drone. Two separate implementations were explored to pass the x, y, and z positions calculated by the crossing beams methods to the custom controller. One solution was to pass raw sweep data into the custom controller in the same way the raw flow data generated by the Bitcraze Flow deck v2 was handled in previous labs. From there, the crossing beams algorithm was implemented within the custom controller itself, converting the raw sweep data to global position coordinates of the drone. Yet, despite the apparent simplicity of this approach, a number of issues arose. Firstly, with this approach, the locations of the two base stations in the global reference frame had to be manually coded into the firmware. This introduced significant error whenever a base station was accidentally moved, as the rays generated by the crossing beams method would be inaccurate. Furthermore, while sweep angles were passed into the custom controller, and base station geometry was manually added, the calibration data unique to each base station was not accounted for. Despite the manufacturing defects in the Lighthouses being incredibly small, the difference between the raw sweep angles and the corrected sweep angles proved to be disastrous for the custom observer and would almost always lead to a failed flight.

As a result, the team had to find a way to pull the x, y, and z locations directly from the lighthouse code, where calibration and geometry data was correctly applied. The first step was to create a function called `ae483UpdateWithLighthouse()` which would accept three inputs representing the x, y, and z coordinates of the drone position. Then, the `controller_ae483.h` header file was included in `lighthouse_position_est.c`, which allowed for the provided code responsible for calculating lighthouse position to call functions defined in the custom controller. Finally, the previously mentioned import function was called to directly pass in the x, y, and z position measured by the lighthouse

positioning deck to the custom controller. One other important note to add is that it was required to explicitly instruct the lighthouse positioning deck to use the crossing beams method when using this approach. This was achieved by setting the `lighthouse.method` parameter to 0 in the `flight.py` file just before the drone began its flight. If this command was not included, the x, y, and z position would only ever be calculated by the Kalman filter and never passed to the custom controller and observer system.

## V. Implementation of Custom Controller & Observer

### A. Calibrating Moments of Inertia and Motor Coefficients

Due to the mass distribution of the drone changing when the Bitcraze Lighthouse position deck and LED ring were added to the top and bottom surfaces of the drone chassis, the team had to recalculate the moments of inertia of the drone as well as the motor coefficients. This was accomplished by using the same procedure that was created for weeks three and four of this course. First, it was assumed that the x, y, and z axes of the frame attached to the drone are aligned with the principle axes of inertia. Secondly, it was assumed that the off-diagonal moments of the moment of inertia matrix were zero. Through making these assumptions, the team was able to connect the moment of inertia with a period of oscillations about the same axis. In order to collect a period of oscillations, the team used a setup that allowed the drone to be swung about one axis at a time. While the drone was swinging, it logged data that was then analyzed in a Jupyter notebook in order to calculate the moment coefficients. A graph displaying an example of the oscilations obtained from such a test is displayed in Figure 7

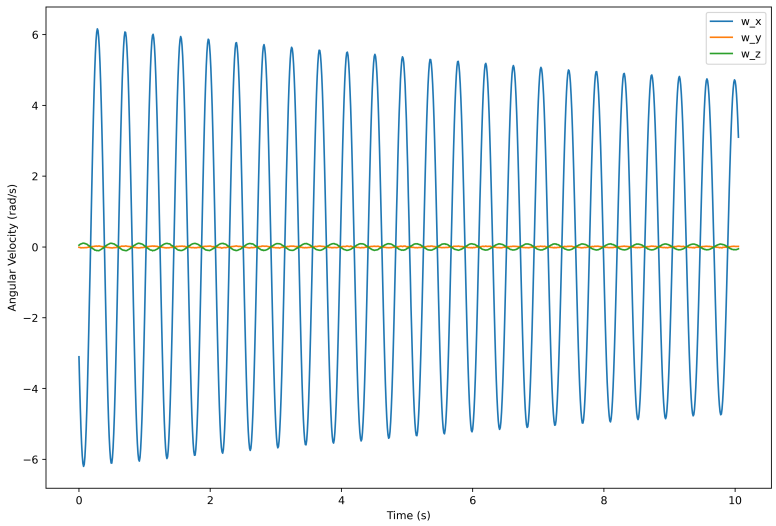



**Fig. 7 Resulting Angular Velocities from Oscillation Test**

Additionally, when deriving the equations of motion for the drone, it was necessary to relate the motor power commands with their resulting torques and forces. To do this, it was assumed that all four motors produced equal moments and forces and that the squared rotor speed of each motor was linearly proportional to the corresponding motor command. Once these assumptions were made, two flight tests were conducted. In the first flight test, the drone hovered at varying heights in order to estimate the force coefficient. In the second flight test, hovered at varying yaw angles in order to estimate the moment coefficient.

Furthermore, later in the project while developing the custom controller and observer, the drone experienced a serious collision causing one of the motor mounts to be damaged. The damage was repaired using hot glue, but the fix added weight to the drone a significant distance away from the center of mass. As a result, the moments of inertia and resulting motor coefficients were slightly inaccurate for the drone after the fix.

## B. Implementing the LQR Method to Controller Design

*1. State-Space Model*

Once the moments of inertia and motor coefficients were estimated, the team implemented a linear quadratic regulator (LQR) to control the drone. LQR is an optimization method that allows controllers to penalize various state errors and therefore put emphasis on states that are measured accurately over those that aren't. However, to implement LQR, it was first necessary to describe the system in state-space notation. The state-space notation formula is given below in Equation 11.

$$\dot{x} = Ax + Bu \tag{11}$$

$$y = Cx + Du \tag{12}$$

In this equation, $A$ is a nxn system matrix where n is equal to the number of states and $B$ is a nxm input matrix where m is equal to the number of inputs. The state matrix and input matrix are found by taking the Jacobian of the equations of motion with respect to the state vector $x$ and the input vector $u$ respectively. In our system, $n = 12$ and $m = 4$. $C$ is the output matrix that relates the state $x$ to the output $y$. $D$ is the feed through matrix that relates the input $u$ to the output $y$. For the states of our system and the inputs of our controller, $x$, $u$, $A$, and $B$, $C$, and $D$ were defined as

$$x = \begin{bmatrix} o_x & o_y & o_z - o_{z_{eq}} & \psi & \theta & \phi & v_z & v_y & v_z & \omega_x & \omega_y & \omega_z \end{bmatrix}^T \tag{13}$$

$$u = \begin{bmatrix} \tau_x & \tau_y & \tau_z & f_z - mg \end{bmatrix}^T \tag{14}$$

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9.81 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -9.81 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

*2. Linear-Quadratic Regulator*

LQR utilizes the $A$ and $B$ matrices from the state-space notation to create two cost matrices $Q$ and $R$. $Q$ is the state cost matrix that penalizes bad performance, and $R$ is the input cost matrix that penalizes the 'effort' the drone has to put in for a successful flight. Because of this, the matrices $Q$ and $R$ are diagonalized and each value along the diagonal of $Q$ is the weight placed on the corresponding state in the $x$ matrix and each value along the diagonal of $R$ is the weight placed on the corresponding input in the $u$ matrix. The higher a weight is for any given variable, the more the controller trusts that variable, and the less it will prioritize correcting it. This increases the stability of the drone as the weights get tuned.

Starting weights were chosen using an identity matrix for $Q$ and using Bryson's Rule for the matrix $R$, which is a way of scaling elements of the state and input so that they are all penalized equally. Bryson's rule looks at the motor commands from the drone, finds the maximum change to each motor command, and uses that to penalize the elements of the state and input. Namely, the weights are shown below in Equation 16.

$$r_x = \frac{1}{20000 l k_f}^2 \tag{16}$$

Where $l$ is the spar length of the drone, $k_f$ is the force coefficient, and $r_x$ is every $x$th component of the matrix $R$. These weights penalize everything equally while helping to minimize the maximum change that any one motor can undergo.

Because Bryson's rule was already applied in week 5 of the course, the team started with the weighted $Q$ and $R$ matrices from that lab and conducted a flight test using these weights. The Root Mean Square Error (RMSE) was then calculated for each of the states. The RMSE values were then continuously recalculated as the controller weights were tuned by the team. This iterative process allowed the team to obtain the weights for the L, Q, and R matrices that were small enough to draw a picture in the x-y plane without introducing noise through the drone either pitching, yawing, or moving unreliably in the x or y direction. The final weights obtained are shown below in Equations 17 and 18.

$$Q = \text{diag}\,(1.5, 1.4, 2.0, 1.28, 1.8, 0.4, 0.69, 0.8, 0.43, 0.65, 0.4, 1.5) \tag{17}$$

$$R = \text{diag}\,(1e6, 5e5, 5e5, 5e2) \tag{18}$$

*3. Verifying Controllability*

In order to ensure that the drone was controllable, we computed the controllability matrix $W_c$ in order to check its rank. If $W_c$ has full rank, meaning that its rank is equal to the size of $A$, then the system is controllable. The controllability matrix is calculated as

$$W_c = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix} \tag{19}$$

The rank of $W_c$ is calculated to be 9, which is the same shape that $A$ has. Because of this, the drone is controllable.

### C. Implementing the LQR Method to Observer Design

*1. State-Space Model*

The observer was then re-implemented by using the procedure from labs seven through nine of this course. First, the equations of motion were modified to match the outputs of the lighthouse positioning system as opposed to the Bitcraze flow deck that was originally attached to the drone. Because the position measurements were taken directly from the lighthouse positioning system, the equation is simply the three measurements as shown below in Equation 20.

$$h = \begin{bmatrix} o_x & o_y & o_z \end{bmatrix}^T \tag{20}$$

With the equations of motion modified, the system and controller were once again put in space-state notation as shown below in Equation 21.

$$\dot{\hat{x}} = A\hat{x} + Bu - L(C\hat{x} + Du - y) \tag{21}$$

Where $A$, $B$, $C$, $D$, $y$, and $u$ are the same as previously defined, but $\hat{x}$ is the state matrix of our new system and is shown below in Equation **??**.

$$\hat{x} = \begin{bmatrix} o_x & o_y & o_z - o_{z_{eq}} & \psi & \theta & \phi & v_z & v_y & v_z \end{bmatrix}^T \tag{22}$$

*2. Linear-Quadratic Regulator*

$L$ is a gain matrix received from using the LQR method described in Section V.B to obtain a new set of weights for the $Q$ and $R$ matrices. These weights were different than the weights used in the controller implementation, as the new space-state model necessitates a different set of weights be used to penalize the observer. These specific weights are shown below in Equations 23, and 24.

$$Q = \text{diag}\,(111111.111, 62500.000, 250000.000) \tag{23}$$

$$R = \text{diag}\,(1479.290, 1479.290, 918.274, 94.260, 24.267, 45.043, 75.614, 28.597) \tag{24}$$

From this, the gain matrix $L$ could be found by using the matrices $A$, $B$, and the new $Q$ and $R$. The gain matrix is shown below in Equation 25

$$L = \begin{bmatrix} 17.758182 & 0 & 0 \\ 0 & 17.722312 & 0 \\ 0 & 0 & 21.430119 \\ 34.333333 & 0 & 0 \\ 0 & -50.75 & 0 \\ 120.120957 & 0 & 0 \\ 0 & 135.915174 & 0 \\ 0 & 0 & 93.5 \end{bmatrix} \tag{25}$$

After the new space-state model was derived and the LQR method was tuned, lab 9 could be run, allowing for further optimization and the final implementation of our observer. Namely, this required running movement tests that measured the RMSE of the drone variables as the drone moved in multiple squares. These tests would show whether or not our observer was implemented well, and whether or not the drone would be able to fly when using the Lighthouse Positioning System. Ultimately, the model turned out to be a good fit and the drone was able to fly.

*3. Verifying Observability*

Now that we had used LQR on the observer, we needed to make sure that the drone was also observable. To do this, we computed the observability matrix $W_o$ in order to check its rank. If $W_o$ has a rank that is equal to the size of $A$, then the system is observable. The observability matrix is calculated as

$$W_o = \begin{bmatrix} C^T & A^T C^T & \dots & (A^T)^{n-1} C^T \end{bmatrix} \tag{26}$$

The rank of $W_o$ is calculated to be 8, which is one less than the shape of $A$. This means that one of our variables in $x$ is unobservable. In the case of our drone, this was found to be $\psi$

Because $\psi$ was unobservable, the drone would need an alternate way to estimate it in order to use it for calculations. To determine psi, we integrated the angular velocity about the z-axis. This could lead to problems because if $\omega_z$ is ever measured incorrectly, then this error would propagate to our estimation of $\psi$. Ultimately, this did not end up being a big issue, but it is something to take into account when it comes to the overall accuracy of the drone.

## VI. Results and Discussion

### A. Edge Detector Performance

The edge detector proved to be quite successful on simple images such as stars, hexagons, and any cartoon without a background, but struggled to find accurate contours on more noisy and complicated images like flowers and human-beings. Figure 8 shows an example of a proper edge detection, where edges are thinned to the length of a single pixel using non-maxima suppression and skeletonization. This is perfect for the recursive move command parsing algorithm because there are very few white pixels to traverse through.

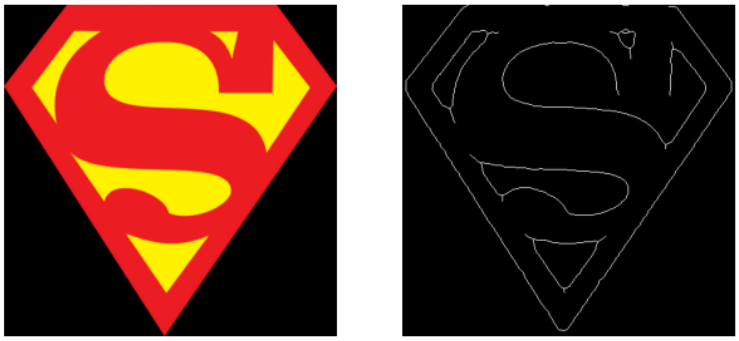


**Fig. 8 Edge Detection Success**

On the other hand, Figure 9 shows a bad example of an edge detection output. The edge detection of the flower shown below is recognizable but is not properly thinned due to background noise in the image and fine detail in the petals. When applying the move command parsing algorithm to the edges, the code crashes due to a recursion error due to the large amount of white pixels. Python provides a recursion limit that acts as a safeguard when a user creates a recursive algorithm. This reduces the chance of a stack overflow error due to an infinite loop during recursion.

**Fig. 9 Edge Detection Failure**

Overall, the team is pleased with the performance of the edge detector. While images must be carefully chosen and fine-tuned with hyper-parameters as mentioned in Section III.B, the capabilities of the model are very promising. Limitations of the model include thick edges and recursive errors which both can be fixed with further time and research. The errors that arose from the implementation of the DFS traversal outlined in Section III.C created a large bottleneck for edge detection success. Future work should focus on making this algorithm more efficient with less recursive calls so that more complicated images can be drawn.

### B. Drone Positioning Performance

The combination of the custom controller and custom observer resulted in drone flights that were certainly under control, but not necessarily accurate. Figure 10 displays the $x$ and $y$ positions recorded from a test flight where the drone was instructed to fly in a 0.5 meter by 0.5 meter square five times. The green line represents the commanded positions for the drone and the orange, dotted line represents the location recorded by the default observer. Finally, the blue solid line indicates the location predicted and recorded by the custom observer whose implementation is described above in Section V.C.

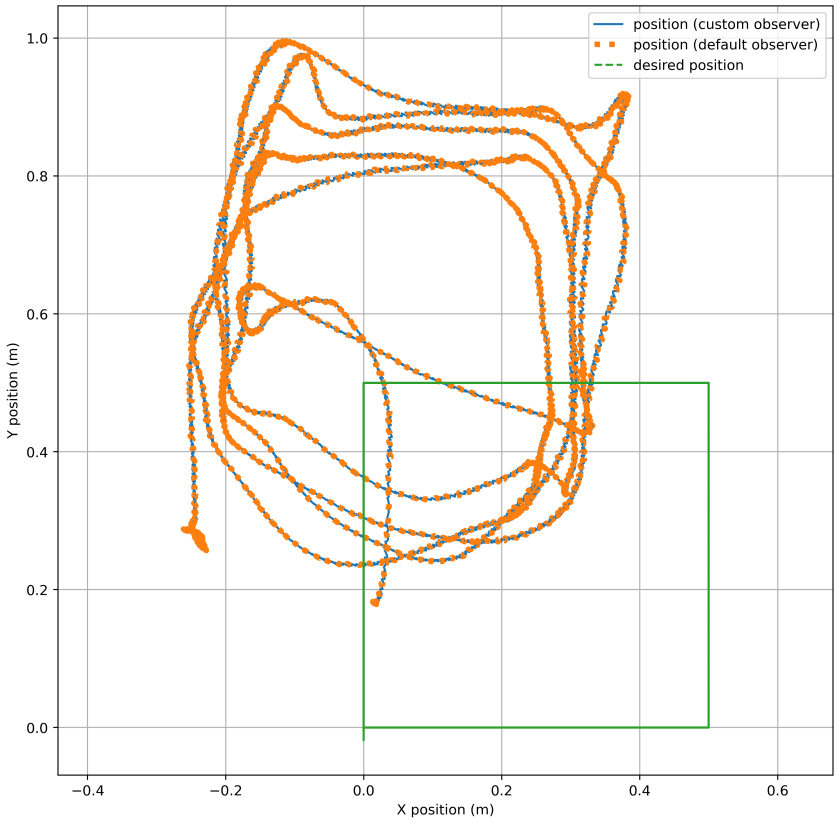


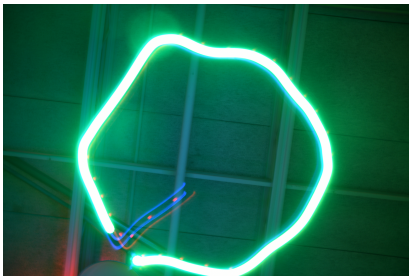

**Fig. 10 Custom Controller and Observer Flight Test**

As the graph clearly shows, the flight of the drone deviates significantly from the desired position in both the $x$ and $y$ directions. While the general square shape may still be recognizable, there is definitely room for improvement. However, aside from the noise, there is another key result that sheds some insight on the performance of the observer. The blue line representing the custom observer and the orange line representing the default observer are practically identical! This result implies the custom observer that was implemented is indeed correctly predicting the position of the drone during flight, with the error observed possibly due to inaccurate predictions of linear velocity in the $x$ and $y$ directions. Such a conclusion is also supported by the way in which the observer was defined, with the state estimates for $x$, $y$, and $z$ position depending heavily on the relatively accurate positions recorded by the lighthouse positioning deck and the state estimates for velocity depending solely on the linearized model.

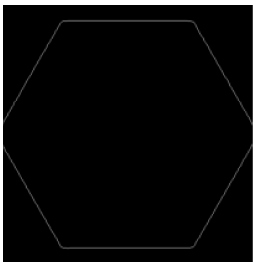
## C. Final Results

*1. Using the Default Observer*

When using the default observer, the drone was able to draw the images produced by the edge detection model with a high level of accuracy. In Figure 11 below, the output of the edge detection model of a hexagon versus the long exposure image taken of the drone are shown.



**(a) Hexagon Drawn by Drone with Default Observer**

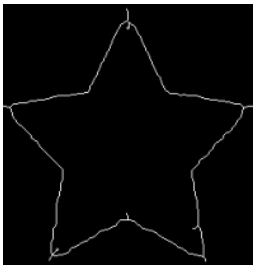


**(b) Hexagon Produced by Edge Detection Model**

**Fig. 11 Hexagon Drawn by Drone with Default Observer vs Produced by Edge Detection Model**

As seen above, the hexagon drown by the drone versus the the one shown in the edge detection model are fairly similar. Additionally, the edge detection model of a star versus the star drawn out by the drone are shown below in Figure 12

**(a) Star Drawn by Drone with Default Observer**



**(b) Star Produced by Edge Detection Model**

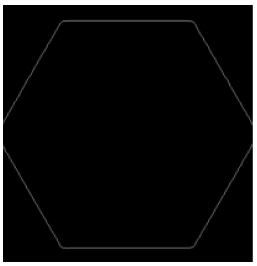
**Fig. 12 Star Drawn by Drone with Default Observer vs Produced by Edge Detection Model**

Besides for the additional line that the drone drew when trying to go from one of the points to the other, the image that was drawn by the drone was fairly similar to the one that was created by the edge detection model. This line appeared due to the way the edge detector functioned, where the model would reach a point on the shape where it could not continue and would pick up drawing at the other endpoint of the shape.

*2. Using the Custom Observer*

As opposed to the default observer, the images that the drone drew when the custom observer was implemented weren't as precise when compared to the image produced by the edge detection model. The same hexagon from figure 11 was attempted with the custom observer implemented and the result is shown below in 13.

**(a) Hexagon Drawn by Drone with Custom Observer**

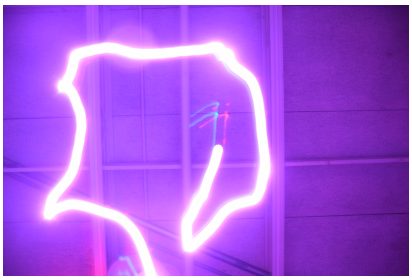


**(b) Hexagon Produced by Edge Detection Model**

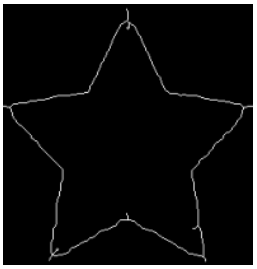
**Fig. 13 Hexagon Drawn by Drone with Custom Observer vs Produced by Edge Detection Model**

Clearly, the shape drawn in the figure above wasn't perfect. Because of this, a star that has a more distinct shape than

a hexagon was attempted with the custom observer and the results are shown below in Figure 14.



**(a) Star Drawn by Drone with Custom Observer**



**(b) Star Produced by Edge Detection Model**

**Fig. 14 Star Drawn by Drone with Custom Observer vs Produced by Edge Detection Model**

While the star produced by the drone wasn't as precise as the star produced by the drone with the default controller, the drone still managed to produce a star-like shape. The image still has the same distinct points of a star and the same line through the middle that the star with the default observer had.

Another interesting observation revealed by the drone's flight is the prediction made by the custom observer. Normally, there is some overall variation between the observer model and the actual flight of the drone due to estimations being used and calculations relying too heavily on noisy sensors. However, the path sketched by the observer is identical to the flight shown in the long exposure image. Figure 15 displays the path of the drone from the flight drawing the star with the custom observer.

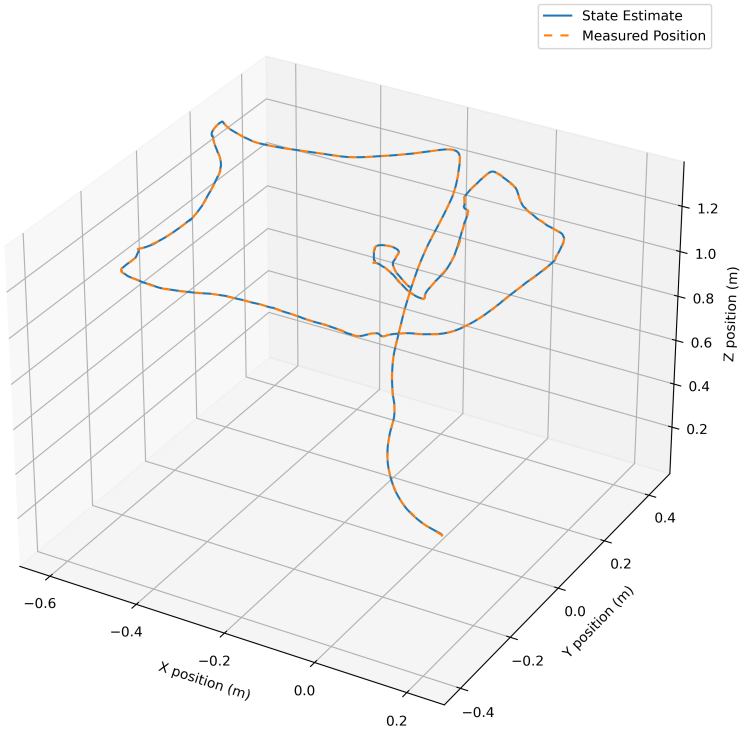



**Fig. 15 Graph of the Observers Measured Position**

This result has many similarities to the outcomes in Section VI.B, where once again the actual position of the drone represented by the default observer was nearly identical to the predicted position from the custom observer. As previously discussed, it is possible that while the drone was accurately predicting position, it was not correctly predicting velocity, causing the drone to 'slide' from position to position. Such a result could also be a result from an issue with the

custom controller In the case of an inaccurate controller, the drone would be unable to fly to the correct position despite receiving the correct state estimates from the observer. In all, the custom controller and observer system implemented for the project could benefit from a more refined set of gain matrices ($K$ and $L$) obtained from subsequent, iterative flights. In the case that tuning these gain matrices does not account for the noisy flight, adding a sensor to directly measure the $x$, $y$, and $z$ components of linear velocity would greatly improve the performance of the controller and observer system.

## VII. Conclusion

Through developing a pipeline to take image inputs, converting the images to lines through an edge detection model, creating move commands from the lines that were produced by the image, utilizing a Lighthouse Positioning System to obtain precise position measurements, and taking a long exposure image of the light produced by a Bitcraze LED Ring as the drone was flying, images were successfully created with the Crazyflie Drone. While the primary goal of drawing images with the drone was achieved, there's a fairly large amount of work that could be conducted in the future. The edge detection model was only able to convert fairly simple pictures to lines and when the custom observer was introduced, the drone was only able to draw out the general shape of the images. Because of this, the edge detection model could be improved so that the drone could draw more complex pictures in the future and the weights of the LQR matrices in the observer could be tuned to increase the accuracy of the observer. Additionally, the edge detection model could be modified so that in addition to producing move commands for the drone, the Bitcraze LED ring could receive inputs to turn on and off when drawing lines that aren't connected. A motivation for conducting this additional work could be to showcase the project at this year's UIUC Engineering Open House in order to motivate the next generation of students to major in STEM.

## VIII. Acknowledgments

We'd like to thank Professor Bretl and Alen Golpashin for their help with the planning and organization of our project idea. The feedback to do hardware tests earlier was greatly appreciated and necessary to make sure that the project is completed within the deadline. We'd also like to thank our lab manager, Dan Block for accepting and providing the necessary hardware and materials for the success of our project. Finally, we'd like to thank Seongyong Hong and his group for aiding us in being able to log position measurements directly from the Lighthouse System and for allowing us to temporarily borrow their drone.

## References

[1] Taffanel, A., Antonsson, T., McGuire, K., Eliasson, M., Richardsson, K., and Rousselot, B., "Lighthouse Positioning System," , 2022. URL `https://www.bitcraze.io/documentation/system/positioning/ligthouse-positioning-system/`.

[2] Lanteigne, A., Kibru, E., Azam, S., and Shammary, S. A., 2017.

[3] Stewart, J., "Long exposure photos capture drones "painting" Light halos over mountains," , Mar 2018. URL `https://mymodernmet.com/reuben-wu-long-exposure-drone-photography/`.

[4] Canny, J., "A Computational Approach To Edge Detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Vol. PAMI-8, 1986, pp. 679 – 698. https://doi.org/10.1109/TPAMI.1986.4767851.

[5] Arbeláez, P., Maire, M., Fowlkes, C., and Malik, J., "Contour Detection and Hierarchical Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 33, No. 5, 2011, pp. 898–916. https://doi.org/10.1109/TPAMI.2010.161.

[6] Jain, R., and et al., *Machine Vision*, McGraw-Hill, 1995, Chap. 5: Edge Detection, pp. 140–180.

[7] Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., and Gunawan, E., "A Systematic Literature Review of A* Pathfinding," *Procedia Computer Science*, Vol. 179, 2021, pp. 507–514. https://doi.org/https://doi.org/10.1016/j.procs.2021.01.034, URL https://www.sciencedirect.com/science/article/pii/S1877050921000399, 5th International Conference on Computer Science and Computational Intelligence 2020.

[8] Sökmen, , Emeç, , Yilmaz, M., and Akkaya, G., "An Overview of Chinese Postman Problem," 2019.

[9] Taffanel, A., Rousselot, B., Danielsson, J., McGuire, K. N., Richardsson, K., Eliasson, M., Antonsson, T., and Hönig, W., "Lighthouse Positioning System: Dataset, Accuracy, and Precision for UAV Research," *ArXiv*, Vol. abs/2104.11523, 2021.

[10] Kimberly, "Advanced Lighthouse Usage Workshop," , Oct 2021. URL https://www.bitcraze.io/about/events/documents/bam2021/Advanced_Lighthouse_Usage_bam2021.pdf.

[11] Taffanel, A., Antonsson, T., McGuire, K., Eliasson, M., Richardsson, K., and Rousselot, B., "Lighthouse Kalman Measurment Model," , 2022. URL https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/lighthouse/kalman_measurement_model/.

# Appendix A

| Day | Task | Person/People |
|---|---|---|
| 10/26 | Worked together to come up with a project idea | Everyone |
| 11/02 | Set up the Overleaf document for the plan | Caleb/Anna |
| 11/03 | Did preliminary on the Lighthouse positioning system and edges to move commands | Eric |
| 11/03 | Did preliminary research for the edge detection model | Matt |
| 11/05 | Worked together to create the presentation for the progress meeting | Everyone |
| 11/07 | Attended progress meeting | Everyone |
| 11/12 | Recalibrated the Moment of Intertia and Motor Coefficients | Matt, Anna, and Caleb |
| 11/21 | Set up Lighthouse Positioning System | Eric |
| 11/22 | Got Lighthouse Positioning System to Interface with Drone | Eric |
| 12/01 | Modified Controller to Match New Moments of Inertia and Motor Coefficients | Matt and Anna |
| 12/04 | Met up in OPEL to work on drone/controller | Everyone |
| 12/05 | Got Edge Detection Model Working | Matt |
| 12/05 | Met up in OPEL to work on drone/controller | Everyone |
| 12/06 | Met up in OPEL to work on drone/controller | Everyone |
| 12/06 | Got Custom Controller to Work! | Everyone |
| 12/06 | Worked together to create video and record audio | Everyone |
| 12/06 | Edited Video for submission | Eric |
| 12/10 | Wrote Abstract | Anna |
| 12/10 | Wrote Introduction | Anna |
| 12/10 | Added edge detection references | Matt |
| 12/10 | Wrote edge detection theory section | Matt |
| 12/10 | Started lighthouse positioning theory section | Eric |
| 12/11 | Contributed first paragraph of Lighthouse Positioning Approach | Anna |
| 12/11 | Contributed first two paragraphs of Implementation of Custom Controller and Observer | Anna |
| 12/11 | Wrote Final Results Section | Anna |
| 12/11 | Wrote Conclusion | Anna |
| 12/11 | Added to the Conclusion | Caleb |
| 12/11 | Wrote about LQR in Custom Controller and Observer | Caleb |
| 12/11 | Edited for grammar mistakes | Caleb |
| 12/11 | Continued Lighthouse Positioning Theory | Eric |
| 12/11 | Added to lighthouse positioning approach | Eric |
| 12/11 | Added figures to lighthouse positioning theory | Eric |
| 12/12 | Wrote sections B and C for edge detection discussion and move command implementation. | Matt |
| 12/12 | Added detail to the Implementation of Custom Controller section | Anna |
| 12/12 | Wrote the LQR Method section | Caleb |
| 12/13 | Wrote the Nomenclature | Caleb |
| 12/13 | Revised edge detection theory section | Matt |
| 12/13 | Completed lighthouse positioning theory section | Eric |
| 12/14 | Completed the LQR Method section | Anna |

| | | |
|---|---|---|
| 12/14 | Wrote the Calibration Moments of Inertia and Moment Coefficients section | Anna |
| 12/14 | Wrote edge detection results section | Matt |
| 12/14 | Completed the lighthouse positioning approach section | Eric |
| 12/14 | Added various graphics throughout the report | Eric |
| 12/14 | Contributed to Observer implementation section | Anna |
| 12/14 | Cleaned up equations in Results and Discussion | Eric |
| 12/14 | Generated plot of observer measured position | Eric |
| 12/14 | Added to the Nomenclature | Caleb |
| 12/14 | Wrote the sections over Verifying Observability and Verifying Controllability | Caleb |
| 12/14 | Wrote about the Observer Graph in the Final Results | Caleb |
| 12/14 | Completed the Drone Positioning Results section | Eric |
| 12/14 | Reviewed report for clarity, grammar, and other small tweaks | Eric |

NOTE: The table above highlights the contributions of each team member to different sections of the paper. In general, people's work and contributions aligned with what they wrote about in the report (i.e. the author of the edge detection section headed the work on the edge detector). Larger tasks such as the video were handled by the whole team, with some people writing the script and others editting the video. Overall, there was no significant disparity in effort/work contributed between each member, as the team worked quite closely throughout the project.