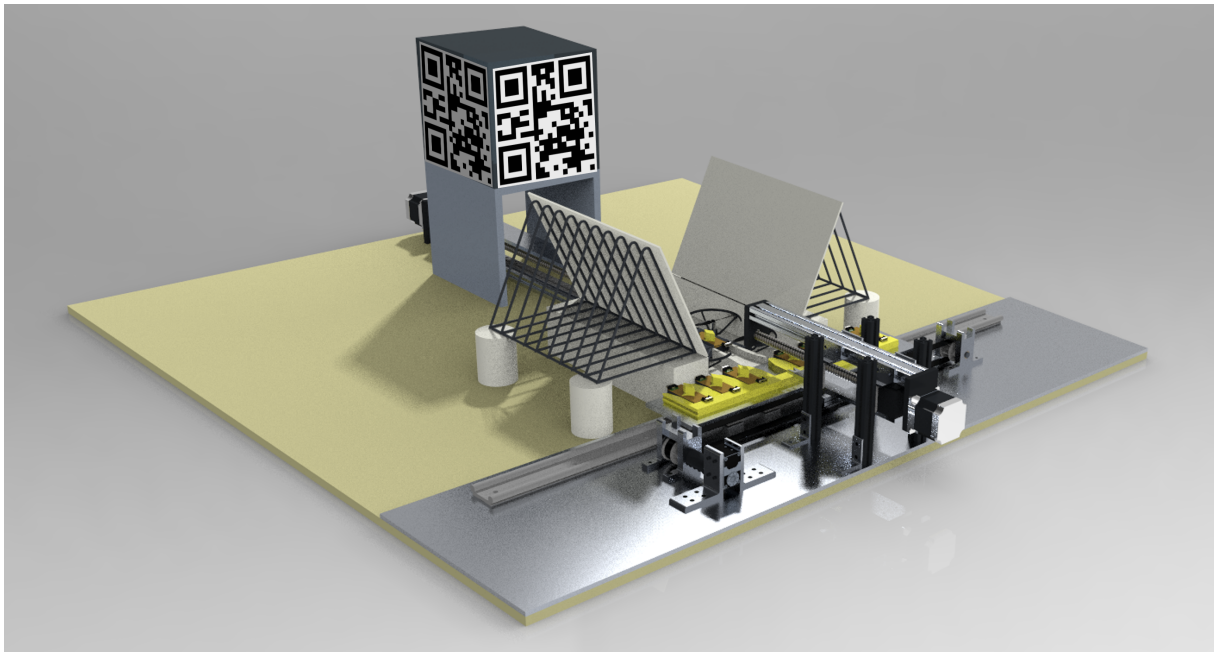
EverFly: An Autonomous Drone Battery Exchange System

John Bachek, Chris Lawrie, Ofek Peres,
and Gabe Roth, '20



Submitted to the
Department of Mechanical and Aerospace Engineering
Princeton University
in partial fulfillment of the requirements of
Undergraduate Senior Thesis.

Final Report

April 29, 2020

Professor Daniel M. Nosenchuck
Professor Michael G. Littman
MAE 444 | 70 Pages | Video Link in Abstract

This thesis represents our own work in accordance with University regulations.

Abstract

This Princeton University senior thesis documents the design and fabrication of an autonomous battery exchange system for a quadcopter. Due to the nature of aircraft flight and battery weight, all drones are constrained by limited flight time. The prototypical EverFly system, outlined in this report, provides a solution to this limitation on drone battery life by enabling a DJI Tello drone to autonomously exchange its battery with a grounded station. The 80-gram Tello was programmed to autonomously guide, navigate, and control itself to the battery station upon detection of low battery. Once the Tello achieves the desired pose and lands, the battery station autonomously secures the drone, extracts its depleted battery, and replaces it with one that is fully charged. Equipped with a fully charged battery, the drone takes off and resumes its autonomous mission. The EverFly system, with a sufficient number of batteries and continuous power input, can facilitate the perpetual quasi-continuous flight of the Tello drone. This report details the modification of the Tello, the design and fabrication of EverFly's hardware subsystems, and the development of EverFly's software architecture. Although full completion of the project was interrupted by the onset of the COVID-19 pandemic, EverFly accomplished the critical goal of executing a successful autonomous battery exchange.

EverFly video demonstration link: `https://ofekperes.github.io/EverFly/`

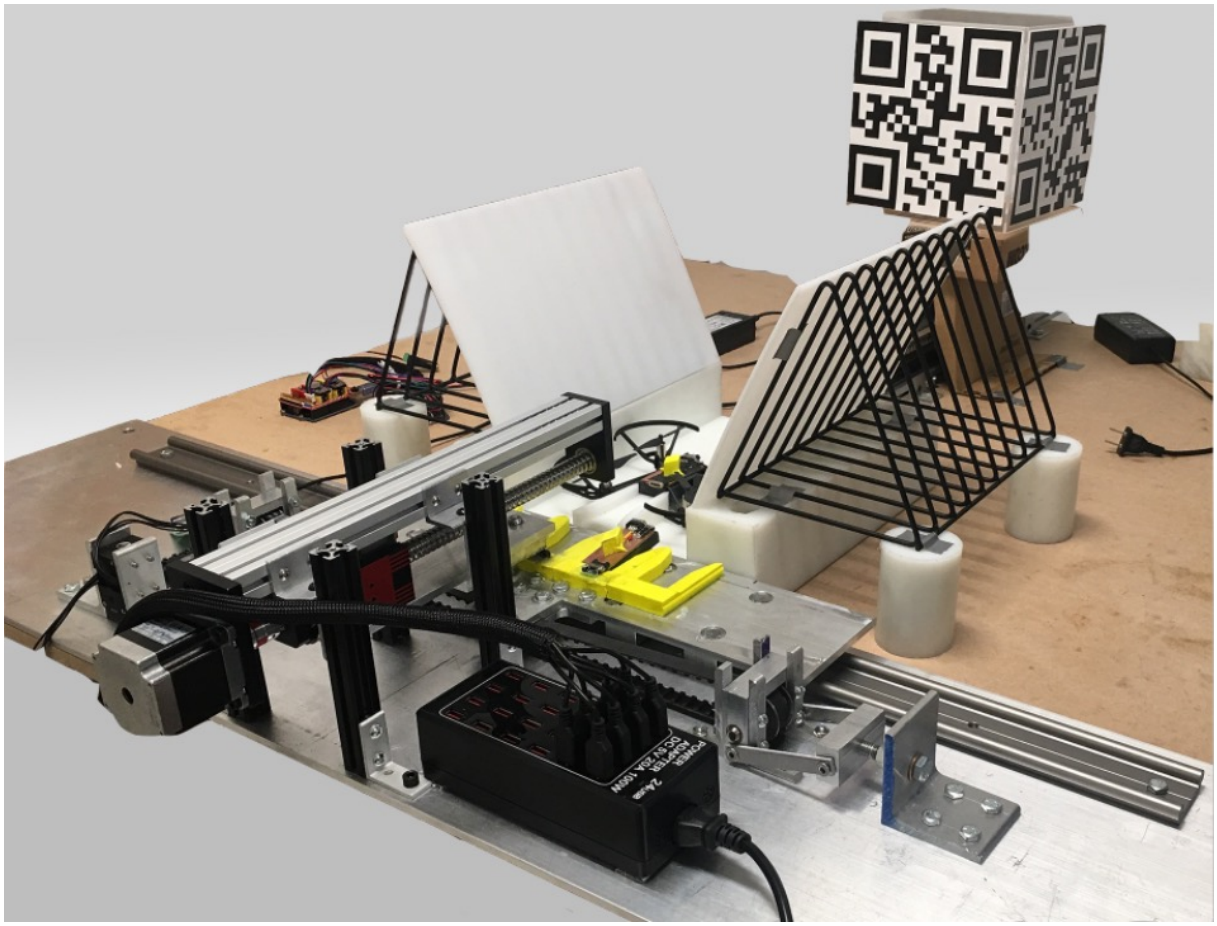


Figure 1: EverFly picture taken on March 17, 2020
prior to campus evacuation due to COVID-19

Acknowledgements

~

We would like to express gratitude to our professor, our advisor, and our mentor, Professor Nosenchuck, for fostering our imaginations whilst keeping our feet on the ground. Thank you for your steadfast support and boundless enthusiasm throughout this thesis project. Your engineering wisdom and product development experience guided us through our design process. Your faith in our ideas and in our ability to execute those ideas empowered us to succeed in our endeavors. It was an honor and a true privilege to work with you, and we look forward to maintaining our relationship with you for years to come.

~

This project could not have been accomplished without the support of the Princeton Mechanical and Aerospace Engineering department faculty. Thank you to our professors for educating us over the past four years; our first-rate education prepared us for this capstone project, as well as our upcoming careers. We also want to acknowledge the dedicated service of the department purchasing office and administration, without whom this project would not have been possible.

~

We want to give a special thanks to Jo-Ann Kropilak-Love for being the world's best undergraduate administrator. Your sincere care for the well being of your students is immeasurable. You are always working to improve the MAE undergraduate experience, and we are deeply grateful for that.

~

We would like to thank Glenn Northey, Al Gaillard, and John Prevost for their expert technical support and pivotal engineering wisdom. Textbooks and classroom lectures cannot compete with the invaluable lessons we have learned from you in the machine shop and in the controls lab. We appreciate your immense patience, devotion, and kindness.

~

Without generous funding, the actualization of our innovative ideas would not have been possible. We are grateful for the financial support provided by the Morgan W. McKinzie '93 Senior Thesis Prize fund and the School of Engineering and Applied Sciences senior thesis fund.

~

We would like to thank Ram Roth, Michael Gottlieb, Nancy and Paul Bachek, and Madelynn Prendergast for their valuable feedback and keen edits to this paper.

~

This project, not to mention our entire academic experience, would have been insurmountable without our friends. Thank you for the late nights, the countless meals, the birthday parties, the problem sets, the study breaks, the board games, the laughs, and all the love. Only friends can turn stress and failure into laughter and memories.

~

Last, but certainly not least, we would like to thank our families for everything. Their endless love, constant support, and faith in our aspirations have gotten us to where we are today.

~

Division of Labor

All team members collaborated on critical design choices, integration of subsystems, and the fabrication of EverFly. The allocation of some specific tasks and division of drafting this report are mentioned below.

John

John contributed to hardware design, fabrication, and implementation. He created many of the CAD parts and dynamic sub-assemblies, and made the engineering drawings compiled in the Appendix. He worked closely with the rest of the team to assess mechanically sound solutions for actuation and component interaction. He manufactured many of the custom-designed parts either manually or using the CNC mill, and integrated them into the final project. In this report, John was a primary contributor for the Battery Station Platform, Battery Extraction, Battery Replacement, and Future Steps subsections.

Chris

Chris was primarily responsible for drone modification, including the magnetic battery modification and drone circuitry modification. Chris designed the battery charging slots and diamond extrusions for battery replacement. Chris was responsible for the conception of both the funnel and landing pad systems, and helped design these in conjunction with the pusher and extractor. Chris also created many of the CAD parts and subassemblies for EverFly. In this report, Chris was a primary contributor for the Abstract, The Drone, Battery Station Platform, Landing Hardware Design, Battery Extraction, Battery Replacement and Future Work sections.

Ofek

Ofek's primary responsibility for this project was software and hardware integration. This included developing the drone's GNC algorithm, as well as integrating the drone's on-board camera with the algorithm. Ofek also designed the control algorithm for the linear actuators, developed the communication protocol between the the master Python program and EverFly's on-board Arduino, and ensured that all of the hardware and software components integrated properly. In this report, Ofek was a primary contributor to the Autonomous GNC, Battery Extraction, and Future Steps sections.

Gabe

On the software side, Gabe developed the GNC algorithm for the drone, as well as the codebase for the Dynamixel servo motor. On the hardware side, Gabe designed and fabricated the landing funnel, the landing pad, and the QR cube. In addition, Gabe created and maintained the EverFly CAD assembly, and ensured accurate actualization of the design during the manufacturing process. In this report, Gabe was the primary contributor for the Abstract, Introduction, Autonomous GNC, Landing Hardware Design, Effects of COVID-19, and Conclusion sections.

Figure 2: EverFly team: Ofek, John, Gabe, and Chris, from left to right

Contents

List of Figures

List of Tables

Listings

Nomenclature

ψ Tello's yaw coordinate, or rotation about the z-direction

$\mathbf{e}_t$ Tello state error vector

$\mathbf{K}_d$ Derivative gain matrix

$\mathbf{K}_i$ Integral gain matrix

$\mathbf{K}_p$ Proportional gain matrix

$\mathbf{r}$ Tello reference state vector

$\mathbf{y}_t$ Tello measured state vector

A_{QR} Area of QR code on Tello camera frame

h height of QR code

N Number of batteries required to provide drone with quasi-continuous flight

s_{QR} Real side length of QR code

t_{charge} Time to fully charge a depleted battery

t_{flight} Time of flight provided by one battery

x Tello reference frame left-right coordinate as shown in Figure 3

y Tello reference frame distance coordinate as shown in Figure 3

z Tello reference frame vertical coordinate as shown in Figure 3

CAD Computer-Aided Design

CEP Circular Error Probable

CNC Machine Computer Numerical Control milling machine

GNC Guidance, navigation, and control

Master Program The Python program that controls all components of EverFly, including the Tello's GNC system, the Dynamixel, and the Arduino Uno

PCB Printed Circuit Board

PID Proportional, Integral, Derivative, in reference to type of control system

PWM Pulse Width Modulation

QR Code Quick response code

Tello DJI drone used for EverFly

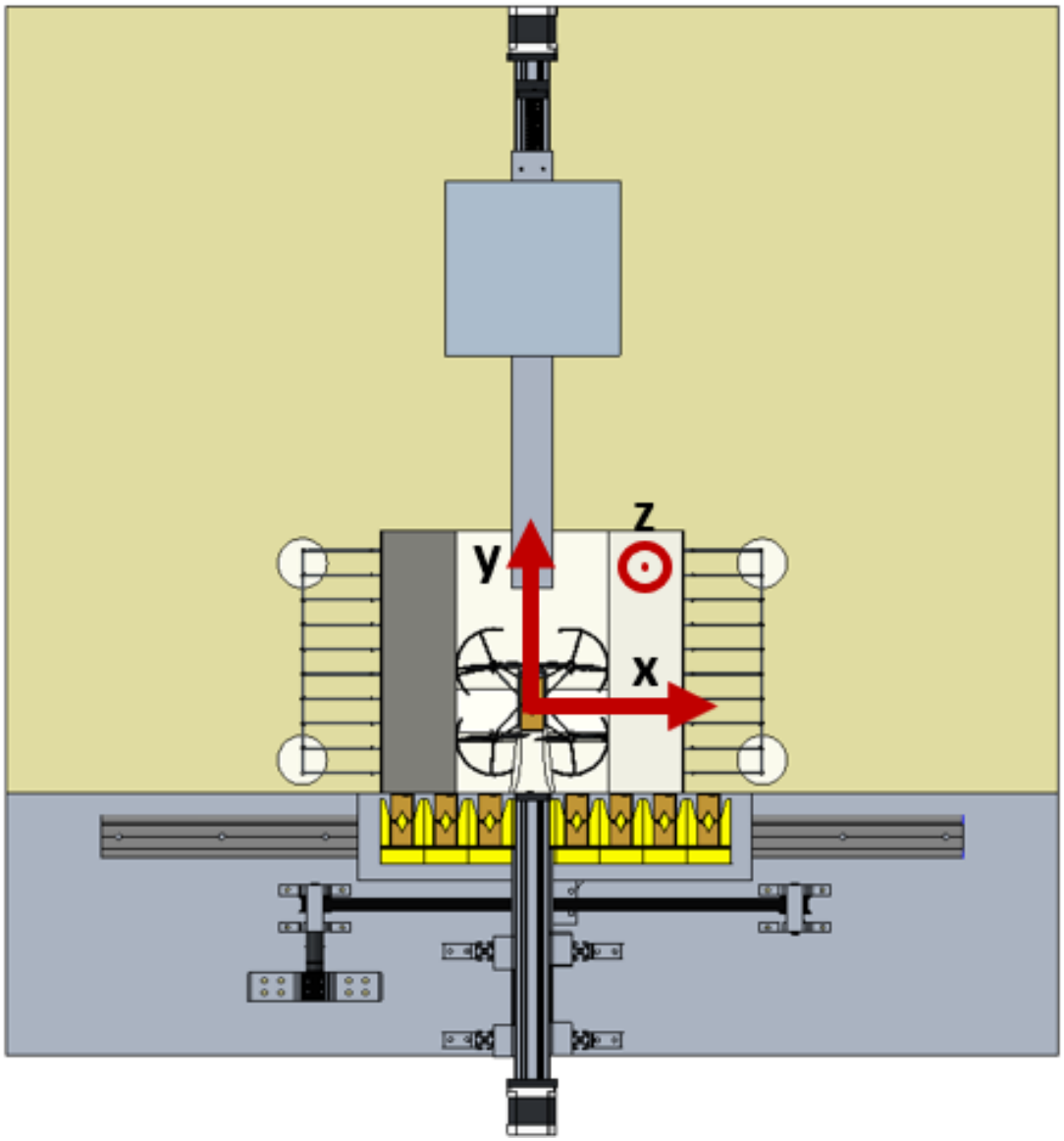


Figure 3: Everfly coordinate system

1 Introduction

Remote controlled drones have increasingly become a part of everyday life. Their wide range of uses includes videography, security, inspection, surveillance, package delivery, and pest control. Flight controls and camera technology have advanced over time to produce highly advanced pieces of engineering. However, despite these improvements, all drones are faced with a suffocating constraint: limited on-board battery life [7].

Commercially available quadcopters[1] have an *advertised* flight time between 10-30 minutes depending on the flight mission. Furthermore, a full battery recharge interval is at least one hour. Even DJI's recent April 2020 release of the Mavic Air 2 only has a battery life of 34 minutes, which DJI advertises as state-of-the-art [9]. The graph in Figure 4 shows the percentage of total use time dedicated to actual flight time for six popular commercial quadcopters. Even higher-end drones like the Phantom 4 and the Mavic 2, which cost about $1600, only approach 30% flight time.

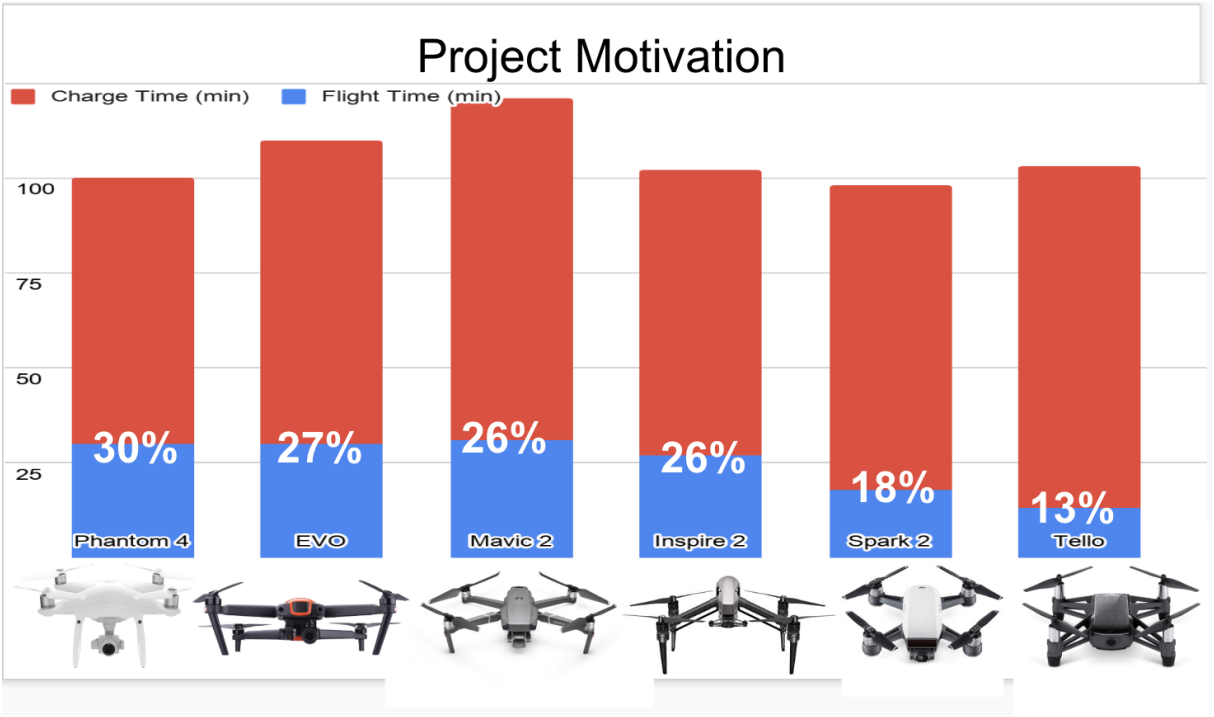


Figure 4: Time allocation for various commercial drones [1] [2] [3] [4] [5] [6]

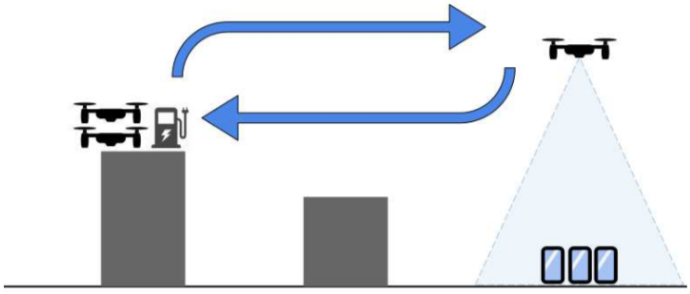
This sample of consumer drone data demonstrates the constraint of battery life for all types of drones. This seriously limits the functionality of quadcopters, and is currently managed with slow recharging or cumbersome hands-on battery changes. Even though hands-on battery exchanging is a quick and energy-efficient way of refueling a drone, it requires human presence and thus disables the drone from being self-sufficient for long periods of autonomous function.

In an age of intelligent autonomous robotics, reliance on a human battery exchanges after short periods of flight is limiting for many applications. An Amazon Prime Air drone that is able to autonomously refuel itself quickly and efficiently is more profitable than one that relies on a human. A military drone that is able to autonomously function for long periods of time is far more valuable than one that has to cease its mission due

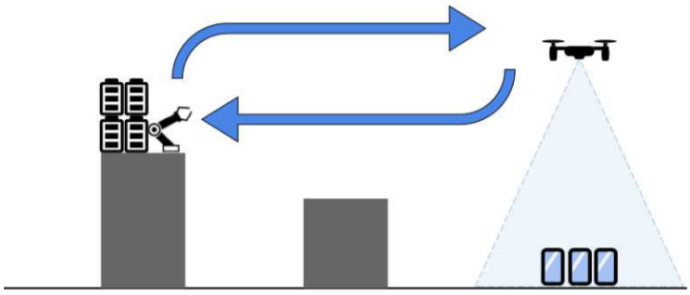
[1] A quadcopter is a subcategory of drones that are controlled by four rotors. Although this paper focuses on quadcopters, the more generic term "drone" will henceforth be used interchangeably with the term "quadcopter."

to low battery. One might suggest using a larger battery, however more battery power adds weight, and weight is costly for aircraft flight. A heavier battery would require more thrust, which would consume more power. Thus, this drone limitation requires a more complex and creative solution.

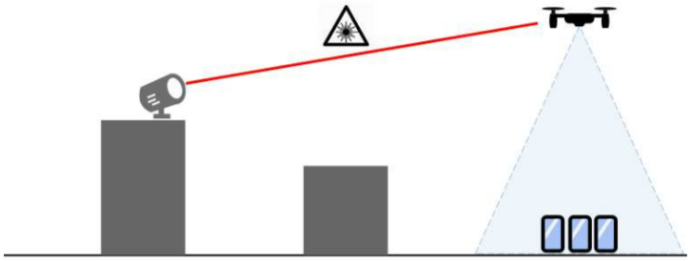
In "UAVs as Mobile Infrastructure: Addressing Battery Lifetime," Galkin et al. outline three general strategies for overcoming a drone's limited battery life while maintaining autonomous functionality. Their schematic is shown in Figure 5. [7]



(a) Cycling through multiple UAVs to cover a hotspot



(b) Hotswapping batteries of a single UAV covering a hotspot



(c) Using lasers to wirelessly power a UAV

Figure 5: Three methods of overcoming limited battery life while maintaining autonomy [7]

The first and most straightforward option, depicted in subfigure (a), is to employ a team of drones so that a fully charged drone can substitute in for the operational drone when its battery is low. Assuming the charging can be accomplished robotically or wirelessly, the main disadvantage of this strategy is the high cost of using multiple drones for a single mission. The second option, depicted in subfigure (c), is to recharge the drone

mid-flight with a laser beam [10]. The weakness of this option is that laser beam charging is very energy inefficient due to the diminishing beam intensity over large distances. The third option, depicted in subfigure (b), is to design a robotic system to replace the drone's battery when it is depleted. This option is both cost-effective and energy-efficient. As a result of these advantages and the exciting mechanical challenge of autonomous battery replacement, this strategy was chosen for the project.

If a drone could land on a battery exchange station that autonomously extracted the old battery and inserted a new one, then all of the weight of the extra batteries would be kept on the ground and the drone would be able to fly quasi-continuously. Figure 6 qualitatively illustrates the advantage of battery exchanging versus recharging.

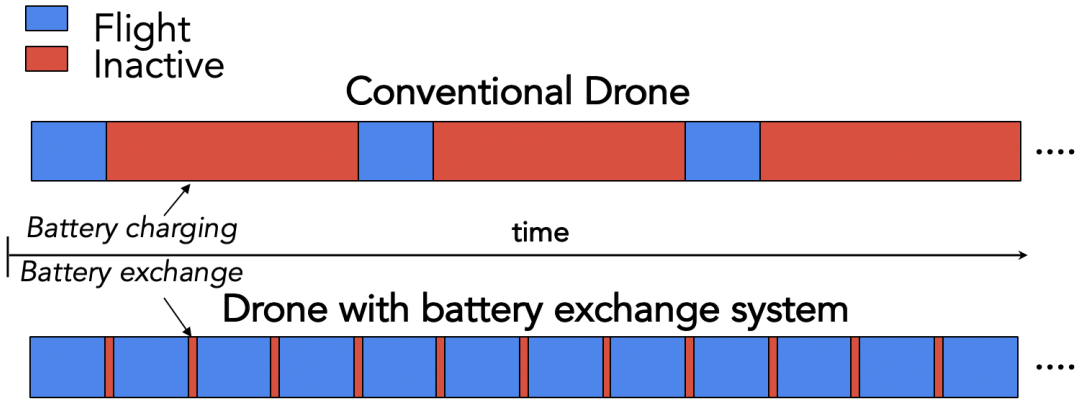


Figure 6: Autonomous battery exchange schematic

Even if supercapacitors are employed for fast charging, it would be less energy efficient than battery swapping. Therefore, to is clear that battery exchanging is superior to recharging. Thus, the EverFly system was designed and built to provide a quadcopter with autonomous battery exchange capabilities. Although the system presented here performs battery exchanges for a specific quadcopter, it serves as a proof-of-concept for any drone. The concept for the system is shown in Figure 7 below.

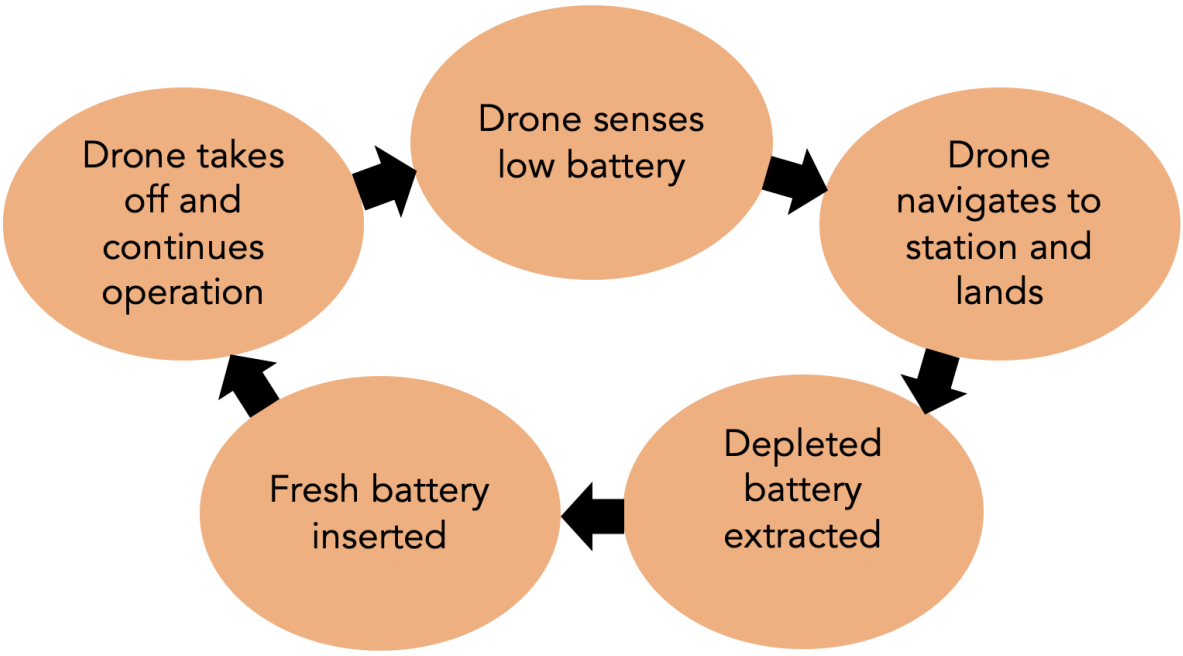


Figure 7: EverFly system schematic

In order for EverFly to operate in a perpetual cycle, the battery station requires a sufficient number of batteries to keep the drone in flight while the depleted batteries have time to recharge. A simple equation based on the flight time of the drone and the charge time of the battery can be derived to give the minimum number of batteries required for the system:

$$N \geq \frac{t_{charge}}{t_{flight}} + 1 \tag{1}$$

Given that the battery station has a power source and at least N batteries, the drone would be able to fly indefinitely with brief interruptions for battery exchanges. Thus, as the name suggests, EverFly can empower a drone to fly forever.

2 The Drone

2.1 Choosing the Drone

The first and most critical design choice was the choice of the drone. Once chosen, the entire battery exchange system would be designed around that drone. One approach would have been to build the drone from scratch. The benefit of this strategy would be that the custom drone is optimized for autonomous battery exchanging (e.g. a circular symmetric magnetic battery at the bottom of the drone). The disadvantage of the custom drone approach is that a lot of time would go into building the drone itself and ensuring it had basic flight capabilities. Alternatively, a high-end consumer drone could have been purchased. The benefit of purchasing a drone like the DJI Mavic [3] is that it is reliable, robust, and has precise GPS navigation. However, DJI does not give access to the software for their higher-end drones, so it would be impossible to completely automate the drone-station system.

Thus, a 'Goldilocks' drone was selected: the DJI Tello. The commercially available DJI Tello has built-in flight capabilities, *and* the drone is programmable in Scratch, Swift, and Python. The ability to accept high level flight commands from a Python script through a WiFi connection was ideal for this project. In addition, the drone is about 6in in diameter, which is suitable for indoor flight testing, and it is only $130, which enables purchasing of multiple drones for testing. An image of the Tello and a CAD model created for the EverFly project are displayed in Figure 8. [11]

Figure 8: DJI Tello

(a) Picture of the Tello

(b) CAD model of the Tello

Parameter	Value
Length	92.5 mm
Width	98.0 mm
Take Off Weight	87 g
Max Flight time	13 mins
Front Camera Resolution	720p
WiFi connectivity	2.4 GHz 802.11n

Table 1: Key Tello Specifications

2.2 Initial Testing

A few initial tests were conducted on the DJI Tello to determine rough guidelines for what modifications would and would not be possible. It was determined that the drone would carry up to 20 grams of extra payload and still function normally. With loads over 20 grams, the drone would either not take off or it would execute an emergency landing. Another key piece of information discovered during initial flight testing is that the Tello sensed if its flight dynamics were being affected by extra payload or even small weights attached to the edges. The Drone stops flying if its moments of inertia are affected in any significant way. This was vital information, since alteration of the battery's location (i.e. not in the designed slot) would prevent the Tello from flying. This initial testing helped determine important design, distribution, and weight constraints, heavily influencing the final outcome of the EverFly system design.

2.3 Battery Modification

Once the Tello was selected and initial flight tests concluded, the battery replacement system design was initiated. The drone is powered by a 3.8V 1100mA-h lithium ion battery of approximately 25mm×76mm×6mm cuboid.[2] The battery is inserted into and extracted from the drone along the y axis, according to the coordinate system shown in figure 3. Once inserted properly, the battery is held in place by friction in the four pin electronic connection between the battery and Tello circuitry, and by a small protrusion and latch system between the battery and the Tello body. This system required modification for the following reasons:

- Fine tactile maneuvering is required to properly grip and extract the battery
- High force is required to break mechanical connection between the battery and the drone
- Precise motion is required to align the battery's four female connectors with the Tello's four male pins, and this alignment is critical for flight

Based on these observations, two main challenges to modification of the battery existed: minimize the mechanical complexity and maximize the reliability and robustness of the battery exchange system. These goals were achieved by making some critical modifications.

A number of design ideas were considered for battery modification, including moving the battery to the bottom of the drone, and replacing the battery vertically. This design was prototyped and worked well; however, the change in location of the battery

[2]https://store.dji.com/product/tello-battery

interfered with the Tello's flight dynamics. Evidently, any significant adjustment to the Tello's weight, center of mass, or moments of inertia could not be tolerated. As a result, modification to the battery had to approximately maintain its original weight and location.

Instead of the default four-pin connectors, magnetic connectors [12] were employed on the battery, as shown in Figure 9. This provided a margin of error during battery insertion since the magnetic connectors would self align by attraction, unlike the original four-pin system. The magnetic connections also provide a secure mechanical connection, which meant friction between the battery and Tello could be reduced for easy insertion. Finally, a simple circuit board was designed, providing circuitry for an additional rear facing magnetic connector. This modification meant the battery only needed to be slid along a linear path between the charging slot and battery, reducing degrees of freedom and mechanical complexity.

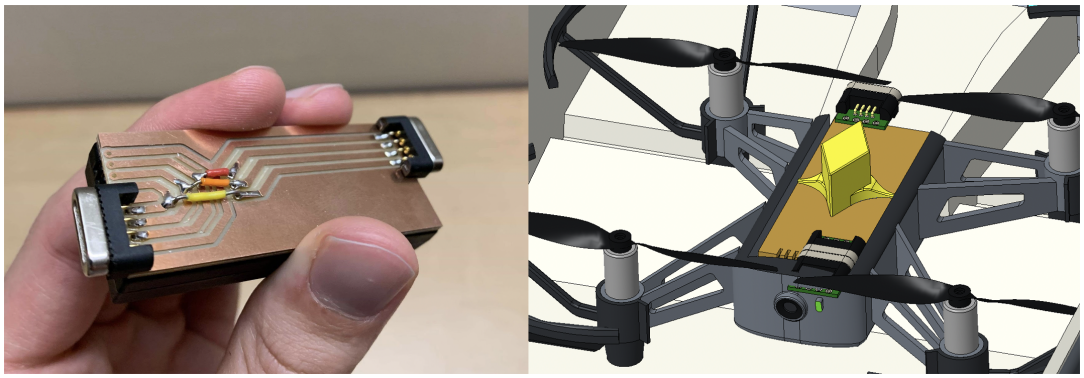


Figure 9: Magnetized Tello battery

2.4 Tello modifications

It was necessary to make a series of modifications to the Tello itself to ensure the success of the project.

The first set of modifications allow the battery to slide in and out with minimal hindrance. This necessitated removing material from the top of the drone's casing in order to make room for the modified battery, which included a PCB board and a diamond flange. Material was also removed from the inside of the drone to reduce sliding friction against the battery. In addition, a magnetic connector was wired directly to the Tello's power supply and epoxied to mate with the battery's magnetic connector. Epoxy was used since it allowed for easy positioning when setting the wires and connector, and once set, the positioning was firm and robust.

The second modification affected the Tello's power switch. The drone has a small power button located on its side. To turn on, one must insert a battery and press the power button. Rather than automating the press of the power button with a mechanical actuator, the Tello's circuit board was modified to automatically power on when connected to power. To achieve this, diagnostic work was conducted on the Tello's circuitry. It was ascertained that the power button caused a momentary dip in voltage, which was sensed by the Tello's microcontroller. Therefore to automatically power on, an electrical component able to simulate a dip in voltage was required.

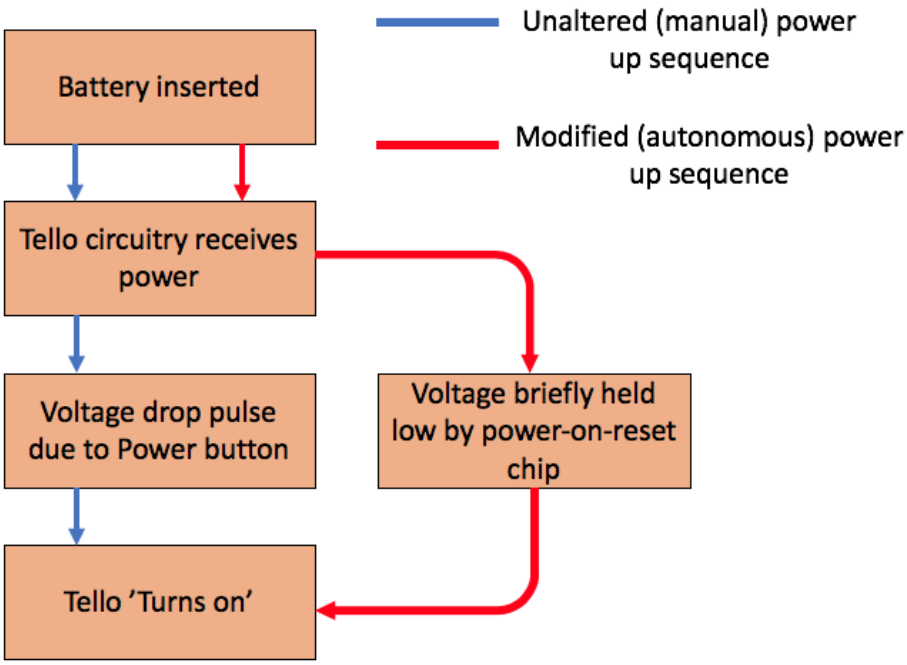


Figure 10: Comparison of modified and unaltered power-on sequences.

A Maxim DS1812 power-on reset was chosen for the task since its voltage operating range was appropriate for the Tello's circuitry.[3] This chip, if soldered across the contacts for the power switch on the Tello's circuit board, would result in automatic power on upon battery insertion. Figure 11 shows the power-on reset's circuit diagram. As shown in the schematic, the circuit requires a DC input voltage, a ground connection, and outputs a voltage that is initially held low, and after a 150 millisecond delay returns to the default.

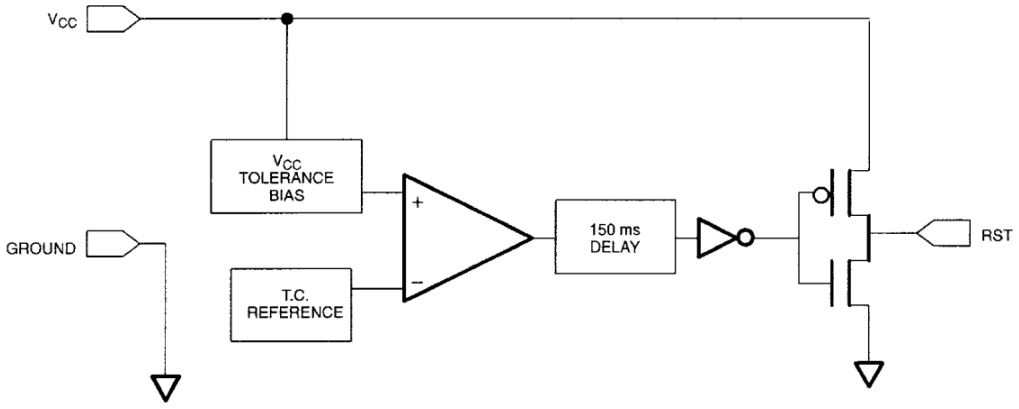


Figure 11: DS1812 Power-on Reset Schematic [8]

Figure 12 shows how the power-on reset chip was integrated into the existing Tello circuitry. Probing tools were used to distinguish between the power contact and the data-

[3]It should be noted that the Tello would also power on if the following sequence was executed: power switch held in, battery inserted, power switch immediately released. This mimics what would be caused by using a power-on reset.

out contact on the power switch. The three leads of the power-on reset chip were then soldered to the corresponding locations on the Tello.

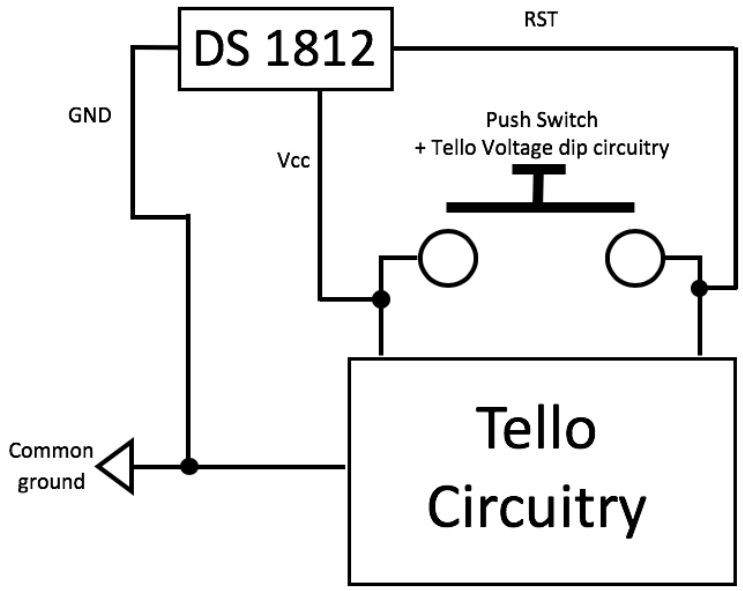


Figure 12: DS1812 bridging the push switch on the Tello

Due to the COVID-19 pandemic, the modified power-on circuitry remained prototypical. A number of issues arose surrounding the robustness of solder connections between the Tello's circuit board and the power-on reset chip. Finding a suitable grounding point for the reset chip also proved difficult. Ultimately, a fully functional automatic turn-on feature was not achieved due to the prioritization of other milestones in light of the sudden termination of lab access. All of the effects of the COVID-19 shut down of Princeton University are summarized later in Section 7.

3 Battery Station Platform

The drone and battery modifications informed the design of the battery station platform. Ultimately EverFly's battery exchange platform needed to incorporate four main components:

- A homing system for the drone to locate the platform and land
- A mechanism to precisely position and secure the drone for the battery exchange
- A mechanism to execute the battery exchange
- A recharging system for the batteries

Plans for the design of the platform addressed these four requirements. The overall size of the platform needed to accommodate whatever form these systems would require. Gauging the degree to which the Tello landing could be finely controlled served as a preliminary estimation of the CEP (Circular Error Probable) before it was formally measured. This informed the range of travel for the actuators, as well as the minimum width of the

battery table (stipulated by Equation 1). Using conservative estimations, a tabletop size of 48in×48in was chosen, offering 16 square feet of space for all of EverFly's components.

A slab of 48in×48in×0.5in MDF served as the base and thus attachment point for all components. To facilitate ergonomic construction of the project, a wooden table was procured from a team member's house, upon which the MDF was placed to bring it to waist height. The MDF was not secured to this table, allowing for easy transport.

In designing the battery table subsystem, discussed in detail in Section 6, it was found that many components would experience a constant and considerable force from the tension of the timing belt that was employed. It was crucial for these components to remain precisely in place to have proper belt tracking, and to maintain actuator precision. The difficulty of precisely drilling into the MDF, as well as its structural weakness, prompted the use of a large 48in×12in×0.25in aluminum plate as a base for these components. Holes could be precisely drilled into this plate using a milling machine, and it offered much more rigidity than the MDF. An additional advantage was that the battery table subsystem was standalone, and could be worked on separately from the rest of the EverFly system. For hardware that protruded from the bottom of the aluminum plate, relief holes were drilled in the MDF so that the plate could lay flush on the MDF. Four holes passing through both the aluminum plate and MDF at the corners of the aluminum plate were used to locate the plate on the MDF, and rigidly fix the two together for the final assembly. The rest of the components of the EverFly system were instead fastened directly to the MDF using hardware.

4 Landing

4.1 Autonomous Guidance, Navigation, and Control

The first task for the EverFly system is the drone's autonomous guidance, navigation, and control (GNC) to the battery station. Almost all state-of-the-art drones have GPS capabilities, so in theory, the drone would use GPS to get within close range of the battery exchange station. Once within a certain range, the drone would use precise computer vision to land in a particular pose. However, since the DJI Tello is not equipped with GPS capabilities, EverFly assumes that the drone is already within vision sensing range. Proof of concept of the close-range GNC was thought to be much more important than the long-range GPS navigation for the purposes of this project.

For the vision-based GNC, EverFly was equipped with QR codes that the drone's camera could detect. When the drone detects low battery, it rotates about the z-axis and searches for a displayed QR code. Once a QR code is detected in the camera's frame, a PID control system drives the drone to the desired pose. Once the drone achieves the desired pose, it lands. Below is a depiction of the high-level flow chart for this vision-based system. The next few subsections will go into detail on the QR code detection, pose estimation of the drone, and control system.

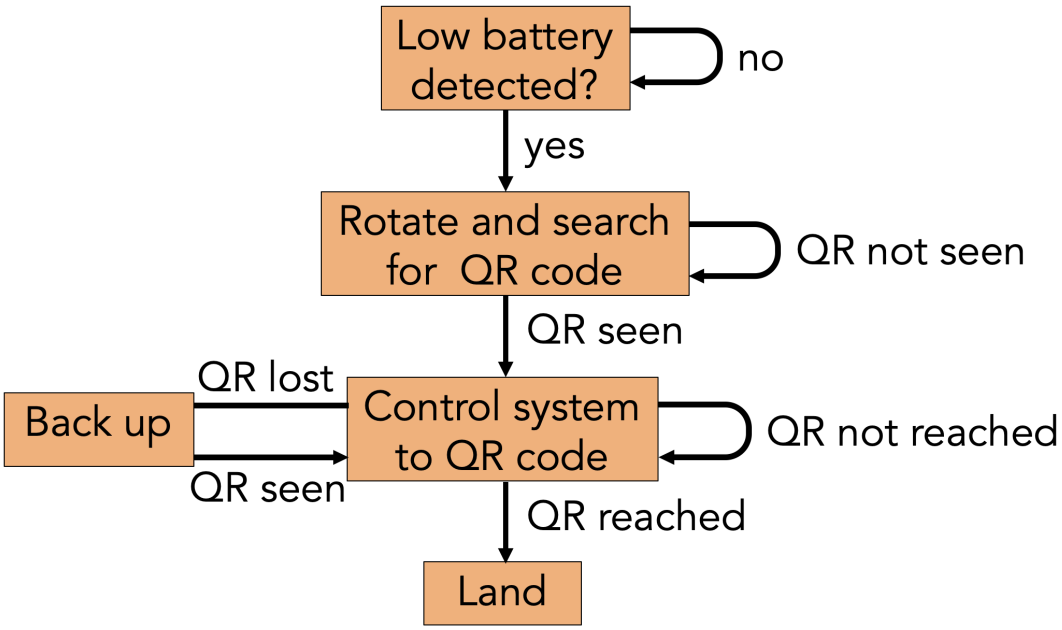


Figure 13: High-level GNC flow chart

4.1.1 QR Code Detection

The first step to QR code detection was connecting with the Tello's video stream over a UDP Socket connection. Once a connection was established, an openCV VideoCapture object was utilized to capture frames from the on-board camera. These frames were the input to the overall control system.

Once a frame was passed into the control system, it was parsed using the Pyzbar Python package, which is able to parse bar and QR codes. Pyzbar was utilized to parse the input frame and extract information about any QR codes that it recognized. The key information that was extracted included the data stored in the QR code (e.g. the side length), the bounding box of the QR code, and the enclosing quadrilateral of the QR code. The side length in inches stored in the QR code as well as the four coordinates of the enclosing quadrilateral were passed into helper functions that calculated the area of the QR code and the height difference of the two sides. These inputs were vital for calculating the distance of the Tello from the QR code as well as its yaw offset.

4.1.2 Pose Estimation

Once the QR code is detected in the drone's visual field, the pose of the drone relative to the QR code must be estimated in order to control the drone to the desired pose. The desired pose of the drone relative to the QR code can be expressed by the state vector $\mathbf{r} = \begin{bmatrix} x_{des} & y_{des} & z_{des} & \psi_{des} \end{bmatrix}^T$, where x is the side-to-side offset of the drone relative to the QR code, y is the distance from the drone to the QR code, z is the vertical offset of the drone relative to the QR code, and ψ is the yaw offset of the drone relative to the QR code. These coordinates are defined above in the Nomenclature Section. Thus, the current pose at each time step can be expressed by $\mathbf{y}_t = \begin{bmatrix} x_t & y_t & z_t & \psi_t \end{bmatrix}^T$, where each entry is computed as some function of the visual input of the drone's camera feed.

The x and z coordinates of the pose estimation are the pixel distances in x and in z, respectively, between the center of the QR code and the center of the drone's field of view.

$$\mathbf{y}_t = \begin{bmatrix} x_t & y_t & z_t & \psi_t \end{bmatrix}^T$$

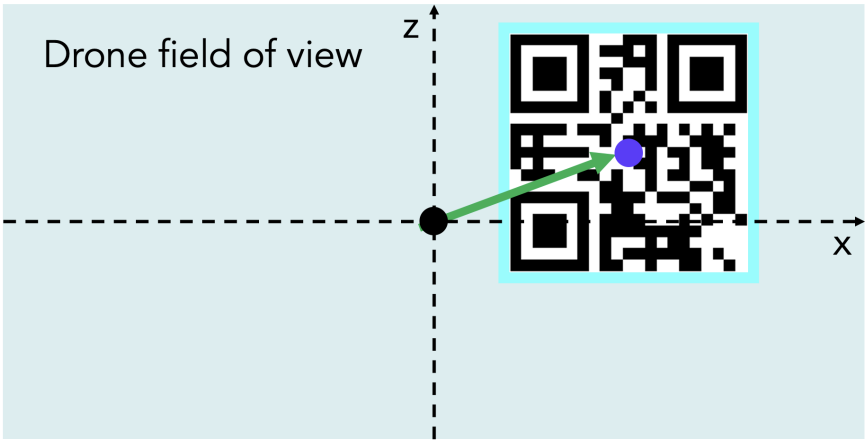


Figure 14: Estimation of x_t and z_t

Initially, a pinhole camera model was used to convert the pixel distance to true distance in inches, but this computation was found to be unnecessary for the purposes of the control system. Thus, x_t and y_t are given by:

$$x_t = \Delta x_{pxl}$$

$$z_t = \Delta z_{pxl}$$

The y value of the pose estimation was computed with a model obtained from training data. The distance from the drone to the QR code was assumed to be a function of the QR code's true size and the ratio of the QR code area on the screen to the entire screen area, i.e. the percent area of the QR code. For different sized QR codes, percent area and distance measurements were recorded as the drone was moved to different distances from the QR code. Figure 15 on the left shows the drone's view during this data collection, and on the right is a plot of the fitted data.

$$\mathbf{y}_t = \begin{bmatrix} x_t & y_t & z_t & \psi_t \end{bmatrix}^T$$

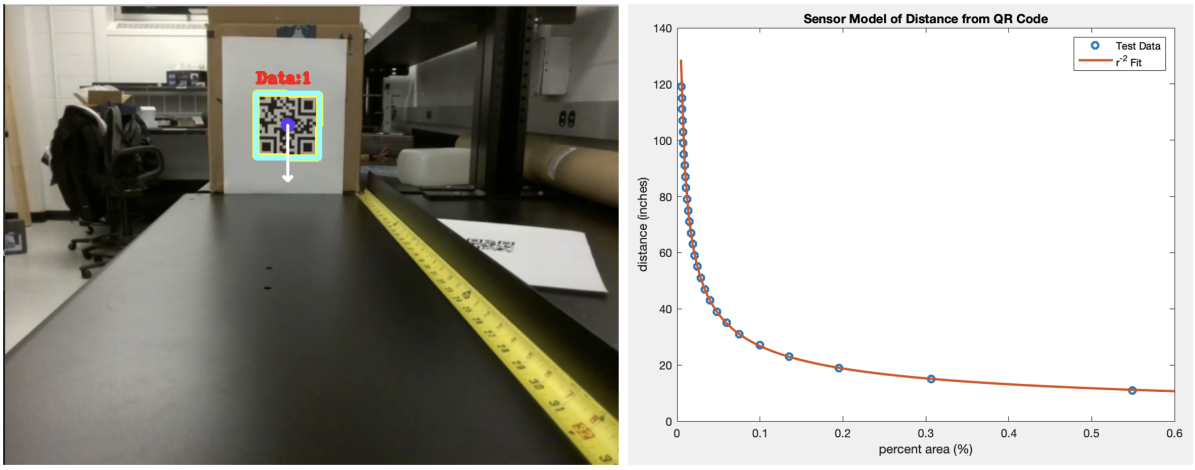


Figure 15: Estimation of y_t

The data was fit to a $\frac{1}{r^2}$ curve, which was then used to output a distance given the percent area of the QR code in the drone's camera frame. Thus y_t is given by:

$$y_t = f(\frac{A_{QR}}{A}, s_{QR}) = a(s_{QR})(\frac{A_{QR}}{A})^{-\frac{1}{2}}, a = 1.07$$

The ψ value of the pose estimation was assumed to be a linear function of the height difference between the left and right sides of the QR code. A yaw offset would cause the QR code to appear distorted on the camera. It was hypothesized that this distortion was well characterized by the difference in heights between the two sides of the QR code. This measurement is depicted below.

$$\mathbf{y}_t = \begin{bmatrix} x_t & y_t & z_t & \psi_t \end{bmatrix}^T$$

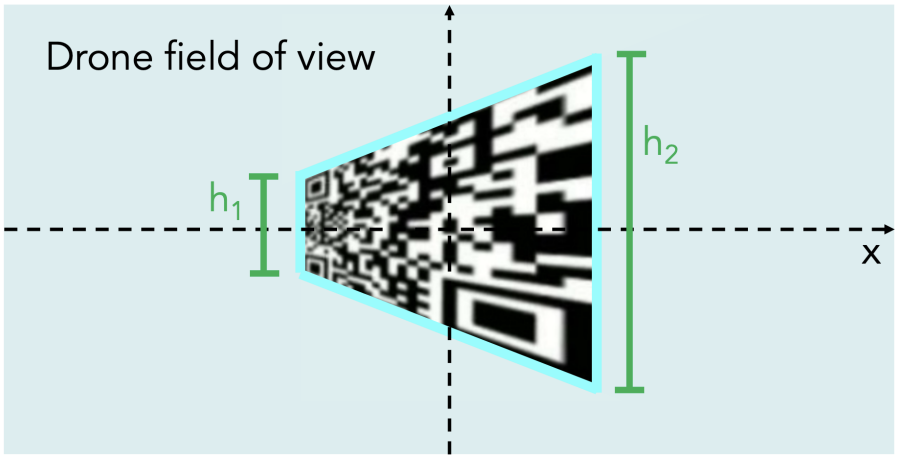


Figure 16: Estimation of ψ_t

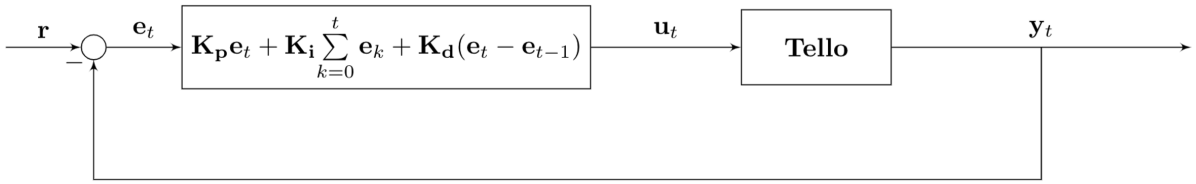
Similar to the x and z measurements, converting to an objective angle offset was deemed unnecessary for the purposes of the control system. Thus, ψ_t is given by:

$$\psi_t = \Delta h$$

This pose estimation, $\mathbf{y}_t = \begin{bmatrix} x_t & y_t & z_t & \psi_t \end{bmatrix}^T$, is computed at every time step from the drone's camera feed, and is sent to the control system which drives the drone to the desired pose.

4.1.3 Control System

In order for the drone to land in front of the QR code, the reference state vector, $\mathbf{r} = \begin{bmatrix} x_{des} & y_{des} & z_{des} & \psi_{des} \end{bmatrix}^T$, is set to $\mathbf{r} = \begin{bmatrix} 0 & 14" & 0 & 0 \end{bmatrix}^T$. This means that the control system drives the drone to a pose that is directly 14" head-on in front of the QR code. With this reference pose and the pose estimation, the error, $\mathbf{e}_t = \begin{bmatrix} e_{x_t} & e_{y_t} & e_{z_t} & e_{\psi_t} \end{bmatrix}^T$, is computed and inputted (along with the previous time step error) to the PID control law. The output of the control law is a vector of velocity commands, $\mathbf{u}_t = \begin{bmatrix} u_x & u_y & u_z & u_\psi \end{bmatrix}^T$, that is sent to the Tello. The block diagram and the parameters of the control loop are shown below.



$$\mathbf{r} = \begin{bmatrix} x_{des} \\ y_{des} \\ z_{des} \\ \psi_{des} \end{bmatrix} \quad \mathbf{y}_t = \begin{bmatrix} x_t \\ y_t \\ z_t \\ \psi_t \end{bmatrix} \quad \mathbf{e}_t = \mathbf{r} - \mathbf{y}_t = \begin{bmatrix} e_{x_t} \\ e_{y_t} \\ e_{z_t} \\ e_{\psi_t} \end{bmatrix} \quad \mathbf{u}_t = \begin{bmatrix} u_x \\ u_y \\ u_z \\ u_\psi \end{bmatrix} = \mathbf{K}_p\mathbf{e}_t + \mathbf{K}_i\sum_{k=0}^{t}\mathbf{e}_k + \mathbf{K}_d(\mathbf{e}_t - \mathbf{e}_{t-1})$$

$$\mathbf{K}_p = \begin{bmatrix} k_{p_x} & 0 & 0 & -Rk_\psi \\ 0 & k_{p_y} & 0 & 0 \\ 0 & 0 & k_{p_z} & 0 \\ 0 & 0 & 0 & k_{p_\psi} \end{bmatrix} \quad \mathbf{K}_i = \begin{bmatrix} k_{i_x} & 0 & 0 & 0 \\ 0 & k_{i_y} & 0 & 0 \\ 0 & 0 & k_{i_z} & 0 \\ 0 & 0 & 0 & k_{i_\psi} \end{bmatrix} \quad \mathbf{K}_d = \begin{bmatrix} k_{d_x} & 0 & 0 & 0 \\ 0 & k_{d_y} & 0 & 0 \\ 0 & 0 & k_{d_z} & 0 \\ 0 & 0 & 0 & k_{d_\psi} \end{bmatrix}$$

Figure 17: PID Control System

The controller was initially a simple proportional controller, with the $\mathbf{K}_p$ matrix consisting of proportional constants on the diagonal and a 'drift' term on the upper right corner, as shown above. This drift term corresponds to the amount that the drone needs to move in the x-direction as it is rotating to correct its yaw error so that the drone drifts around the QR code without losing sight of it. This term is equal to the negative radius of the drift maneuver–which is simply the y_t measurement–multiplied by a proportional constant.

The proportional controller generally worked well for guiding the Tello close to its desired pose; however, there was a minor problem when the controller tried to fine-tune the pose more precisely. When the Tello was close to the desired pose, the control system would generate relatively small inputs for the drone to execute. The Tello can only take velocity commands between $10\frac{cm}{s}$ and $100\frac{cm}{s}$ in magnitude. So when the control system sent inputs below $10\frac{cm}{s}$ in magnitude to the drone, no actuation occurred. In order to solve this problem, an integral term, the diagonal matrix $\mathbf{K}_i$, was added to the control law to accumulate the error over time and exceed the $10\frac{cm}{s}$ threshold. Though the integral term solved the problem of fine inputs, it made the drone too aggressive at far distances from the QR code. Thus a derivative term, the diagonal matrix $\mathbf{K}_d$, was added to the control law to dampen out the aggressive reaction of the control system when the drone was far away from the QR code. Together with well tuned gains, the PID control system worked with 100% accuracy over the course of over 30 tests from different initial poses within a 10ft range.

4.2 Landing Hardware Design

The overarching design goal for the landing hardware was to minimize the number of mechanical actuators and to provide the drone with an error margin for landing. Fewer actuators means more reliable autonomy, fewer breakpoints, and a cheaper system. An error margin for landing means less dependence on the precision of the GNC system, and overall leeway on the autonomy of the system.

EverFly ultimately needs to lock the drone in a specific pose, constrain it, extract the depleted battery, insert a charged battery, and release the drone for take off. In theory, once the drone lands within its CEP it needs to be moved in the landing platform's plane

to a particular position. Assuming the drone's yaw orientation is close enough that it only needs to be translated in two dimensions, two actuators are required to position the drone for battery extraction. However, after extensive testing, the GNC system proved to be quite accurate, especially in the x-direction (axes drawn in the first sub-figure of Figure 19). The drone was reliably able to center its view on the QR code within a 2 inch margin. Therefore, instead of dedicating a separate actuator to moving the drone a few inches after landing, a one-dimensional funnel was designed to center the drone in the x-direction as it landed. The rendering in Figure 18 shows the Tello centered in the x-direction after passing through the funnel.

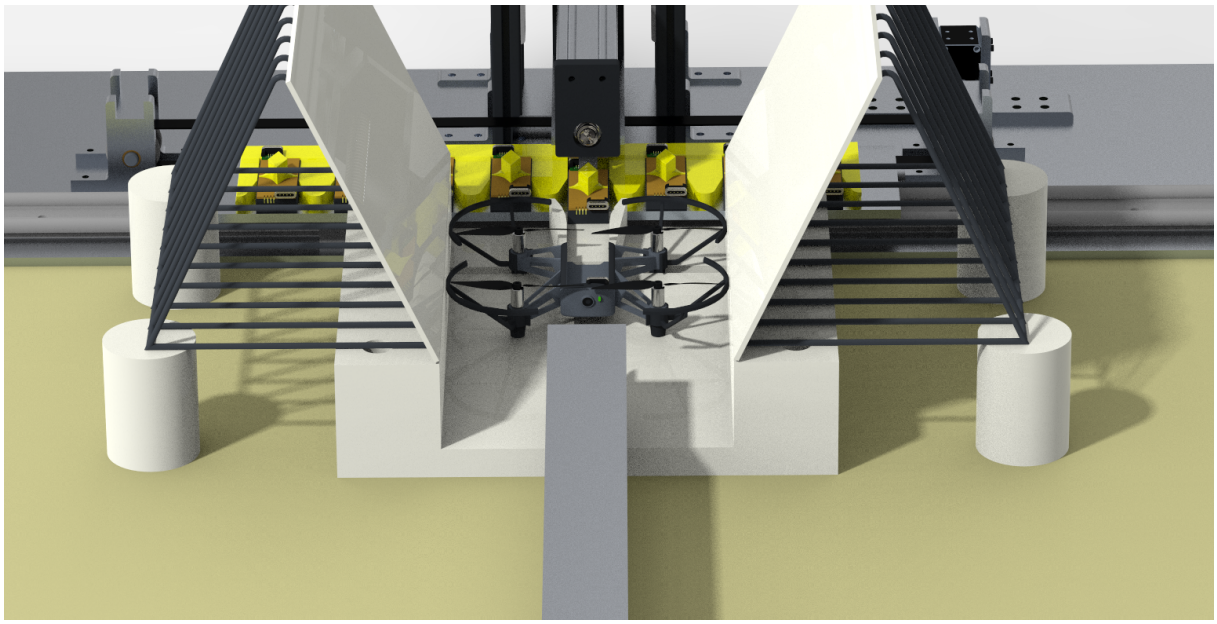


Figure 18: Rendering of Tello in funnel

The Tello's land command has roll stability control built in. This means that when the propeller guards make contact with the funnel, and it exerts a moment on the drone, the drone's internal flight control system corrects its non-zero roll. As a result, the Tello ends up simply shifting over in response to its contact with the funnel. As the drone lowers its altitude, the funnel continues to shift it to the center, correcting for any x-direction error due to the imperfect GNC system. Figure 19 shows a step-by-step funneled landing due to about 1.5 inches of error in the x-direction .

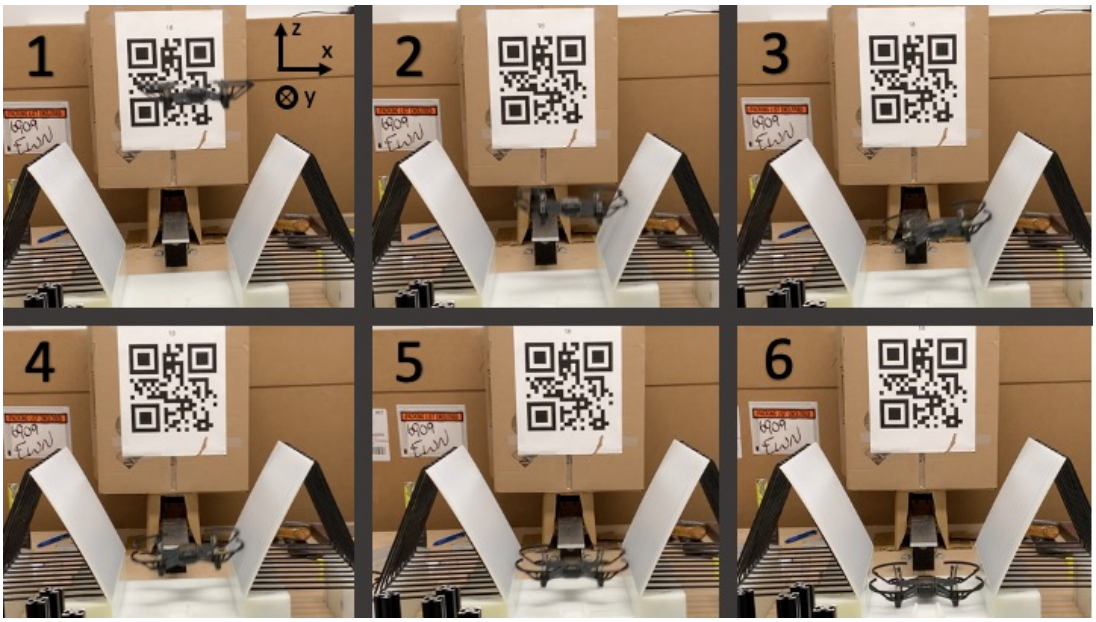


Figure 19: Landing funnel demo

Each funnel was constructed with an 8in by 12in polyethylene sheet fastened onto a wire frame triangular prism with a 25° half angle. Polyethylene was chosen for its smooth, low friction properties. The wire frame was chosen as a simple, lightweight structure for the desired funnel shape. This design accurately positioned the drone in one direction without any actuation.

Once the drone has been centered in the x-direction , it needs to be positioned and constrained in the y-direction . Instead of having one linear actuator to position the drone in the y-direction , and a second linear actuator to push the battery out of the drone, a clever yet simple landing pad design enabled the integration of these two mechanical tasks to be executed by one actuator.

The landing pad was designed such that one linear actuator can interface with the battery, push the drone via its magnetically connected battery, and eventually continue dislodging the battery out of the drone. This was accomplished by designing a unique landing pad with a gradual-incline ramp and a narrow trench at the end of the ramp. Figure 20 shows a cross sectional view (left) and isometric view (right) of this design.

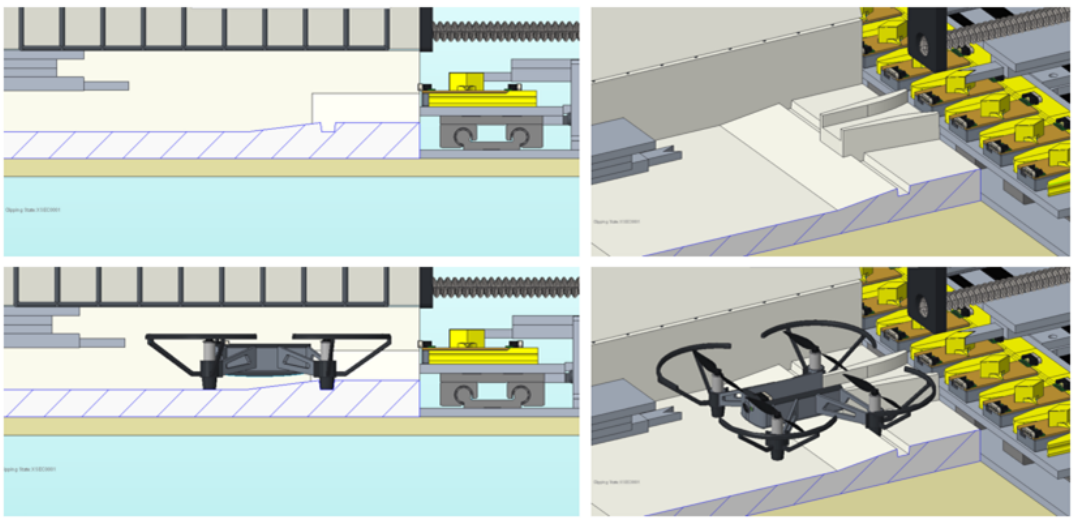


Figure 20: CAD of landing pad

At first, the linear actuator interfaces with and pushes the drone's battery, which pulls the drone due to the strong magnetic connection between the battery and the drone. As the drone is pushed along the landing pad, two of its feet climb the low-angle incline until they fall off the edge into a narrow trench. This trench constrains the motion of the drone, and the continued movement of the linear actuator overcomes the magnetic force connecting the battery to the drone and slides the battery onto the battery channel, which is built into the landing pad. This process is depicted below in Figure 21.

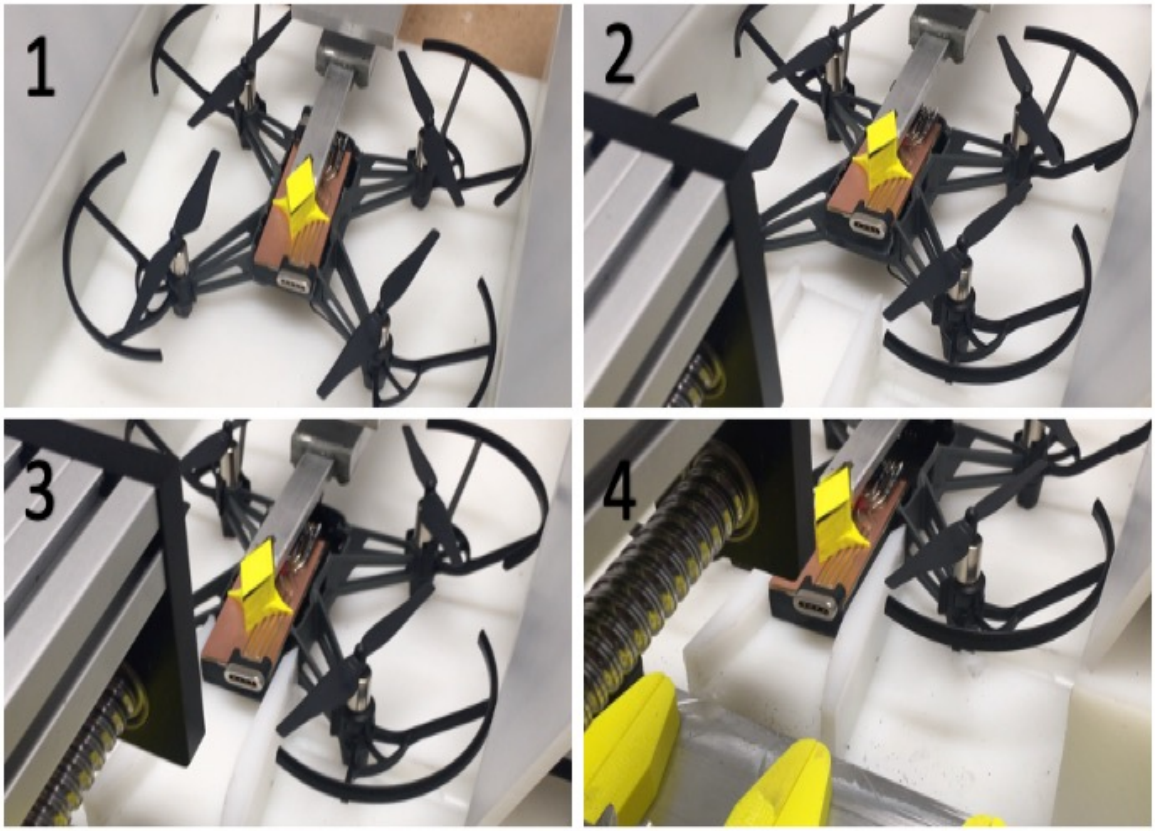


Figure 21: Drone being pushed along landing pad

The trench was designed to perfectly fit the shape of the Tello's feet, and give some tolerance so that the drone could freely take off. The landing pad was manufactured in a CNC machine from a 3 inch thick block of HDPE (high-density polyethylene). HDPE was chosen primarily due to its ease of manufacturing in the CNC, which could capture the intricacies of the design.

Together, the funnel and landing pad designs reduced the theoretical need for three actuators (two for drone positioning and one for battery extraction) down to a single actuator. This was extremely helpful in simplifying a potentially complex series of autonomous tasks.

5 Battery Extraction

5.1 Extraction Procedure

After the drone lands, the battery extraction proceeds as follows:

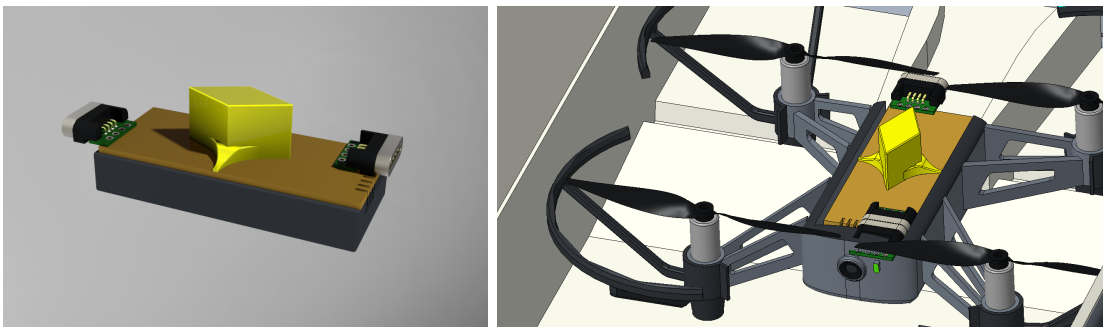
1. Lead screw initiates, advancing the extractor arm towards the static drone.
2. Extractor arm makes contact with drone battery diamond.
3. The extractor pushes the battery, and the drone to which it is still connected,[4] towards the battery table.
4. The drone advances until it can no longer move due to the foot trench in the landing pad (as described in 4.2).
5. Once the drone is fully constrained by the trench, the extractor continues pushing, forcing the battery through the connecting chute between the Tello and battery table.
6. The pusher continues advancing until the battery is received by the empty charging slot. At this point, the pusher fully retreats.

5.2 Extraction Actuator Hardware

Once the drone lands between the funnels in the proper orientation, it is constrained from rotating and translating in the x-direction. The only remaining degree of freedom is in the y-direction. As discussed in Section 4.2, the landing pad design enabled the battery extraction mechanism to constrain the drone in the y-direction in addition to removing the battery.

A small diamond extrusion was designed and epoxied onto the top of each battery, as shown in Figure 22a. This part was designed to interface with the battery extraction and insertion arms on either side.

Figure 22: Battery with PCB, magnetic connectors, and diamond extrusion



(a) (b)

The end effectors of the extraction and insertion arms are shaped to accept the profile of the diamond extrusion, shown in Figure 23. This geometry provides an accurate connection to be made reliably. It also automatically corrects for small errors in the

[4]This happens because the magnetic connectors contact force is greater than the static and/or sliding friction of the drone.

x-position of the drone or pusher-arm (errors that are less than half the width of the diamond in magnitude).

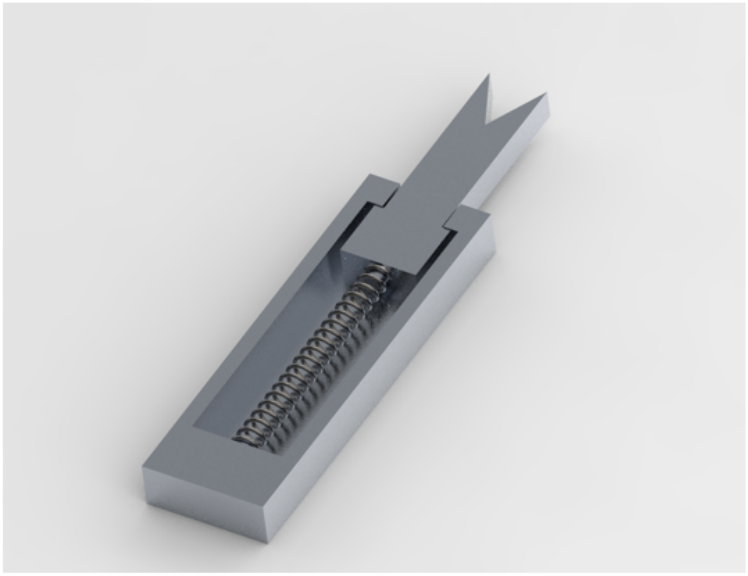


Figure 23: Rendering of spring-loaded end effector on each linear actuator

Since the battery follows the motion of the extraction actuator, which is strictly in the y-direction, error of the battery's position in the z-direction is not a concern. Error in the x-direction must however be mitigated. The battery is constrained in the x-direction during extraction until the channel in the HDPE block begins to taper outwards. The open end of the battery receptacles has a matching taper back inwards to approximately the width of the battery (the inside of the receptacle is roughly 0.010in oversized to allow some tolerance and prevent binding). This taper design compensates for error in the position of the battery. Even in the unlikely misaligned position of the battery at this interface, the battery would still not bind. This scenario is in fact unlikely, since the geometry of the pushers and diamond on the batteries already reliably ensures straight-line motion. A diagram of the battery channel and battery receptacles can be found in Figure 26.

The extraction mechanism itself is actuated by a 400mm stroke ballscrew linear actuator. The ball screw mechanism, as opposed to a rack-and-pinion or timing belt, allows for accurate position control, as well as speed reduction and torque amplification. Instead of manufacturing the mechanism from scratch, a $156 FUYU FSL40 linear guide [13], which was driven by a NEMA23 [14] stepper motor, was purchased in the interest of time. In order to avoid the landing zone, the actuator was positioned behind the landing pad. A cantilevered aluminum arm was fastened to the linear actuator's carriage, such that the carriage's motion along the ball screw corresponded to the arm's motion along the landing pad. At the end of the cantilevered bar is the spring-loaded end effector, which mates with the diamond extrusion atop the drone's battery during extraction. The extraction actuator assembly is shown in Figure 24. Figure **??** shows three snapshots of the battery extraction process.

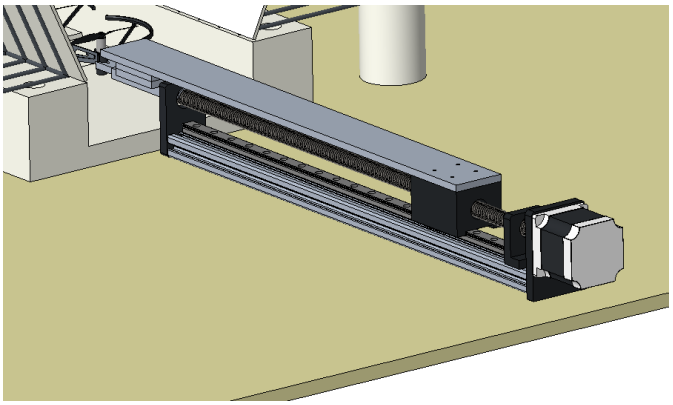
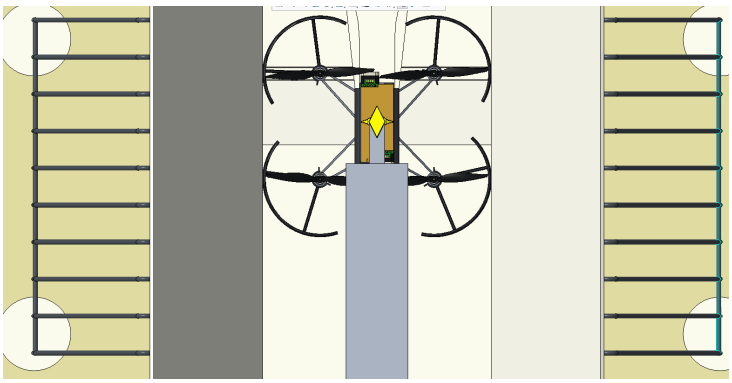
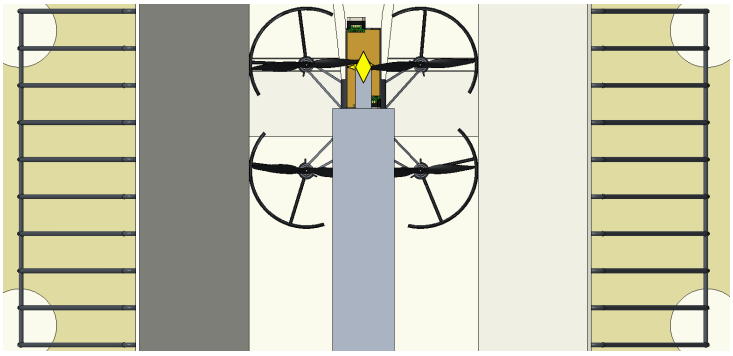


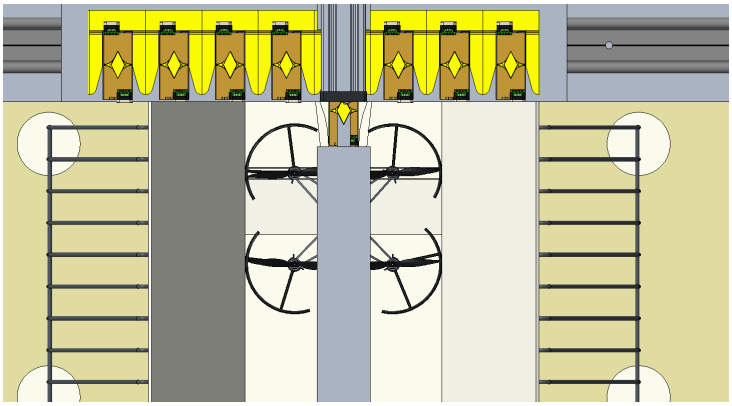
Figure 24: CAD of battery extraction actuator



(a) Battery is disconnected from Tello



(b) Battery is pushed into battery channel



(c) Battery is inserted into charging slot

Figure 25: Battery extraction schematic

5.3 Extraction Actuator Control

5.3.1 Control System Overview

Upon a successful landing, the Tello sends a signal to the master program. Once this signal is received, the battery extraction commences. The master program initiates communication with the on-board computer, an Arduino Uno, that controls the linear actuators. The Uno completes the inputted task (e.g. extract battery and return home) and returns a signal indicating success or failure.

5.3.2 NEMA23 Arduino Custom Functionality

Using an Arduino was a natural choice for controlling the linear actuators' NEMA23 Stepper Motors. However, a key concern was connecting it to EverFly's master program, which is written in Python. A connectivity issue would prevent EverFly from operating autonomously because there would be no data flow between the Arduino and EverFly's Python software. Section 5.3.3 discusses the development of a serial communication protocol that allows the master level control program to communicate with the Arduino Uno. This section discusses the functionality that was built into the on-board Arduino Uno that allowed EverFly to successfully extract and replace batteries.

The Arduino Stepper motor package <Stepper.h> [15] was utilized to send low level, PWM signals to the NEMA23 stepper motors that powered the linear actuators. The NEMA23 was only able to accept integer inputs that were 2 bytes in size. This limited the step range to [-32,768, 32,767].

Higher level functionality was built into the Uno. This functionality included functions to move in increments greater than the allowed 2 bytes, to extend, to retract, and to sweep any of the connected linear actuators. In addition, an important feature was maintaining the state of each linear actuator. This ensured that on a software level, the linear actuator would not extend or retract beyond the physical constraints of its size.

In order to move an arbitrarily large number of steps, the input step size was broken into a sequence of small steps that were smaller than 2 bytes in size, allowing the Arduino to fully extend or retract the linear actuators. Each actuator had more than 100,000 steps of travel.

In order to calibrate the functionality to extend and retract, the linear actuators were retracted fully and then slowly extended while keeping track of the number of steps taken until extension was complete. The total travel, in steps, was maintained for each linear actuator. This meant that as long as the Arduino knew what the current position of the linear actuator was, it could figure out how many steps were needed to move in order to fully extend or fully retract. This control system worked well to maintain the state of the linear actuators because the NEMA23 steppers moved relatively slowly and did not slip or skip any steps; however, in order to increase robustness, future steps would include feedback mechanisms such as limit switches.

Once complete, the on-board Arduino was able to receive a command from the master Python program and respond to it, allowing for smooth integration into the EverFly ecosystem.

5.3.3 Communication Protocol

In order for EverFly to be fully autonomous all software had to communicate with the master program and any hardware it controlled. As such, a simple communication protocol was developed to allow the master program to communicate with the Arduino Uno

over serial. Serial communication allows bytes to be sent one after another through a connection like a USB cable. The protocol allowed for a message with the following structure to be parsed by the Uno.

$$< String, String, Integer, Float >$$

In standby mode, the Uno was programmed to continuously listen for serial inputs. If it found a valid input, then the message was parsed and the four inputs were extracted. The four inputs are as follows:

1. A string identifying the object to be controlled (e.g. "LED" or "StepperX")
2. A string defining an action (e.g. "turnOn" or "Step")
3. An integer value to be used in conjunction with the action (e.g. "500" for stepping action)
4. A floating point value to be used in conjunction with the action (e.g. "0.4" fraction of full-range motion)

Two examples of valid input strings:

1. $< led, turnOn, 0, 0.0 >$ would turn on the Uno's internal LED on pin 13.
2. $< StepperX, Step, 500, 0.0 >$ would make the stepper motor connected to the X terminal on the CNC Shield of the Arduino step 500 times.

5.3.4 Future Steps

Due to the COVID-19 pandemic, two crucial upgrades were not accomplished:

1. The first is a feedback mechanism that activates a calibration sequence when the Uno first powers on. This mechanism would ensure that each linear actuator starts at the correct starting position by retracting the actuator until it hits a limit switch. This would ensure that even if the Nema23 had any skipped steps in a previous run, it would be able to calibrate itself and maintain a high level of precision.
2. The second upgrade would speed up the replacement process. Currently, a full extension or retraction is a blocking function call. This means that once the Uno receives a command to extend, it ignores everything else until it completes its mission or fails. This happens because as of now, the move command takes in a target position and repeatedly sends step commands until the target position is reached. An elegant way to make the movement commands not blocking would be to make only one increment towards a goal position on every iteration of the main loop of the Arduino. This would allow the Arduino to receive commands even in the middle of accomplishing other tasks. Once this upgrade is made, the Uno would be able to begin the insertion process by retracting the extraction linear actuator and simultaneously begin the extension of the insertion linear actuator.

6 Battery Replacement

6.1 Battery Table Design

Battery extraction ends with the depleted battery plugged into a charging slot on the battery table. The battery table must then be shifted over such that a fully charged battery is directly in line with the drone's now empty battery slot. The battery table must therefore have the following features and functionality:

1. An interface between the landing block and battery table
2. Hardware and circuitry for charging multiple batteries
3. Actuation to move the depleted battery out of the way, and move a charged battery in place for insertion

6.1.1 Battery Table-Landing Block Interface

When a depleted battery is pushed out of the drone, it must make the transition from the drone to its charging location on the battery table. Careful consideration was given to accomplish this reliably and repeatedly. To facilitate a smooth transition from the landing block's battery channel to the charging table, one of eight 3D printed U-shaped battery receptacles receive the battery as it is pushed onto the battery table. These battery receptacles are the yellow features shown in Figure 26. The number of battery receptacles corresponds to that dictated by Equation 1, and is specific to the Tello.

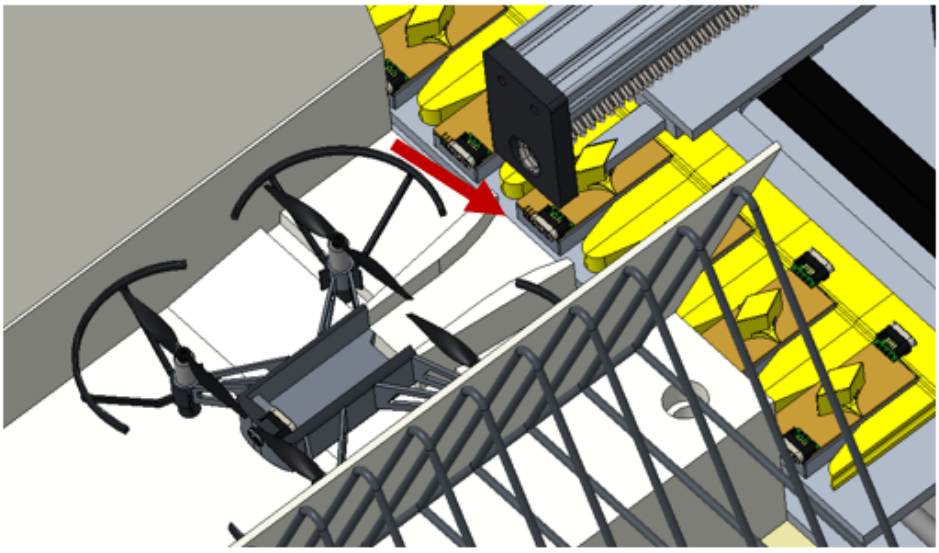


Figure 26: CAD of Battery table-landing pad interface

The battery receptacles have a height slightly lower than that of the battery, to prevent possible interference with either the extraction or insertion actuators. The closed end is the same height as the bottom face of the rear-facing magnetic connector. This ensures a reliable mating of the male and female magnetic connectors at the end of extraction. On the battery table, the battery receptacles are designed such that the batteries are spaced exactly 2in from each other to allow the linear actuation of the battery table to occur in clean increments of 2in.

6.1.2 Charging Functionality

After a depleted battery has been successfully placed on the battery table, it must automatically begin charging for future use. As shown in Figure 22a, each battery has two female magnetic connectors: one to interface with the drone, and another to interface with the charging circuitry on the battery table. A design where a single magnetic connector serving both of these purposes was considered, but was decided against due to the complexity that it would add to extraction and insertion procedures. Namely, the battery would need to be turned 180° in the plane of the table during each extraction and each insertion, which not only adds a degree of freedom and therefore actuator complexity, but introduces an additional chance for mechanical malfunction.

Upon extraction, the magnetic connector on each battery that does *not* interface with the drone mates with a male magnetic connector that is glued to a battery receptacle on the battery table. As mentioned previously, this male magnetic connector is placed such that it is at the same height as its corresponding female connector on the battery when the battery is on the battery table. Its purpose is not only to provide an easily breakable electrical connection between the battery and charging circuitry, but also to locate and secure the battery in the battery receptacle. This male magnetic connector is wired to a Tello battery charger, which receives power from a 24-Port, 5V USB power bank that plugs into wall power. The configuration is shown in Figure 27.

Note that this configuration is repeated eight times, once for each battery on the battery table, in correspondence with Equation 1. Since the battery table moves, and eight sets of wires protrude from the back of it, it was important to keep the wires organized and neat. This prevents the wires from being caught in the actuation mechanisms, which could damage various parts of the system or cause a dangerous short. Due to the circumstances of COVID-19, completion of the wiring in its entirety was not possible. However, all components of the charging circuitry were built and proven to work, with one of the eight intended charging circuits completely built and successfully tested.

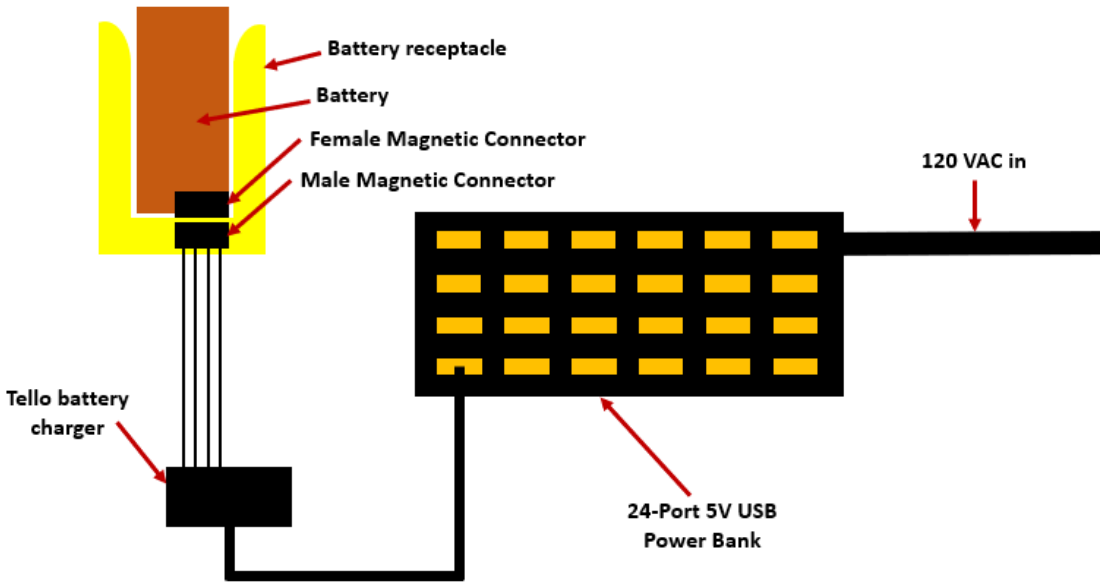


Figure 27: Schematic of the battery table charging circuitry

6.1.3 Actuation

In preparation for battery insertion, the old and now-charging depleted battery must be cleared of the path for the insertion actuator, with a charged battery taking its place. The simplest way to accomplish this was by actuating the battery table along a linear path in the plane of the table and perpendicular to the extraction and insertion actuators, as shown in Figure 28.

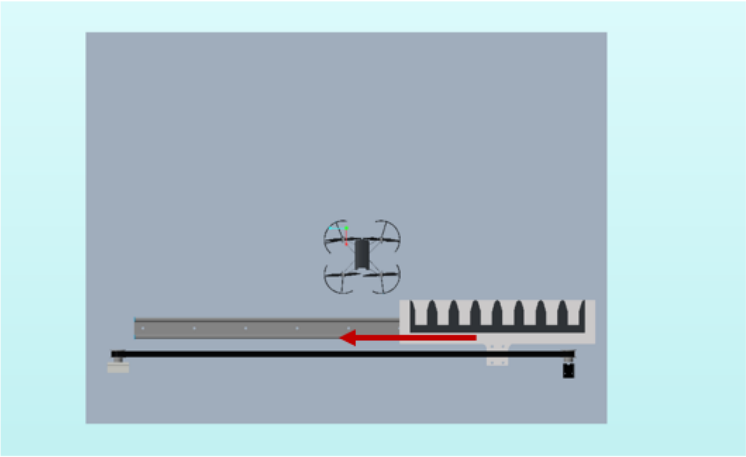


Figure 28: Motion of battery table

Two carriages with nylon bushings slide on an anodized aluminum track, allowing for smooth linear motion. The two carriages support a 4in×18in×0.5in aluminum plate, which forms the base of the battery table. Special care was given to accurately fix these carriages to the aluminum plate. Since the carriages are rigidly attached to the plate, them being even slightly off parallel to one another would inhibit their smooth motion on the track, and cause wear on the contact points and put undue stress on the actuation motor. To decrease the manufacturing precision that was required, a solution in which one carriage is rigidly attached to the aluminum plate and the other carriage is attached at a single point could have been employed. The second carriage would then swivel to self-correct back to parallel with the first carriage if it ever needed to. This solution, however, would have required modifications to be made to one of the carriages and was therefore not employed.

Several methods for linear actuation of the battery table were considered, including a ballscrew, rack and pinion, four-bar linkage, and a timing belt. All of these options could easily combine with a motor to offer precise linear motion. A timing belt was chosen for its relative simplicity, forgiveness of manufacturing imprecision, and low cost.

The actuation mechanism is composed of a drive pulley, an idler pulley, a timing belt and belt clamp, a belt tensioner, and an extended servo motor. These components (with the exception of the tensioner) are shown in Figure 29.

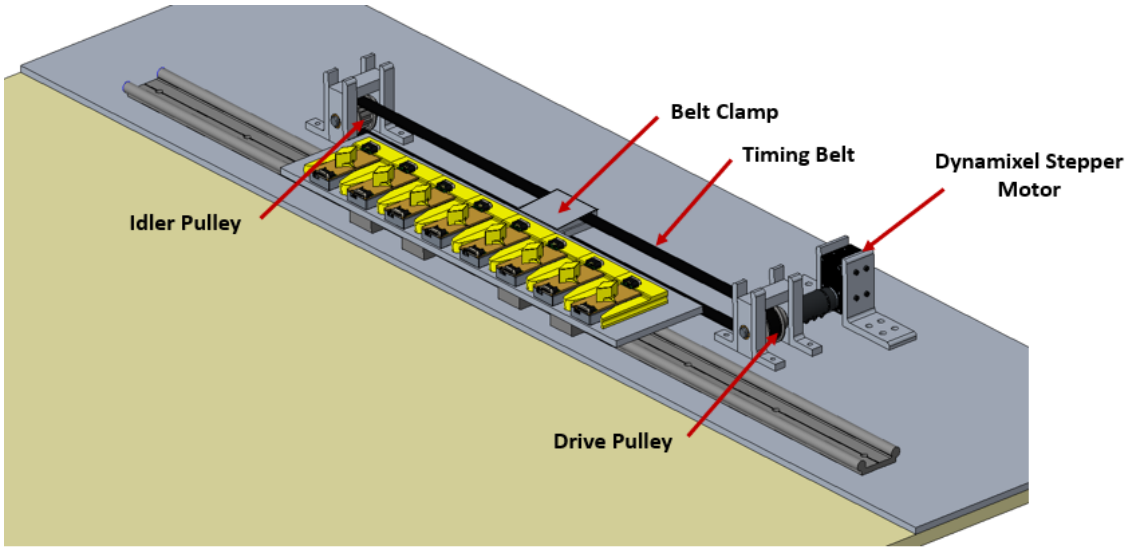


Figure 29: Battery table actuation hardware

The precise rotation of a Dynamixel servo motor allowed for fine control over the position of the linear table. The Dynamixel is discussed in detail in Section 6.2. This precision would be lost, however, without sufficient tension in the timing belt. Therefore a tensioning mechanism was designed to keep the timing belt under constant tension. This mechanism is shown in Figure 30.

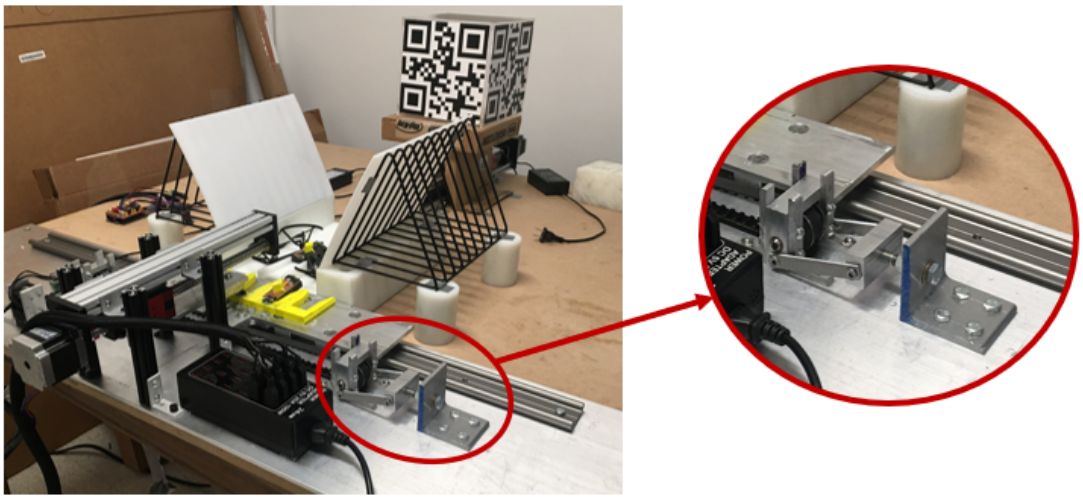


Figure 30: Tensioner mechanism for the battery table's timing belt

The operating principle of the tensioner is simple: a 5/16-18 bolt passes through a piece of L-bracket, and is held captive in the vertical part of the bracket by a nut. The bolt threads into a block that is attached to the idler pulley bracket by two linkages, free to swivel at each of their attachment points. Tightening the bolt will draw the idler pulley toward the tensioner and away from the drive pulley. Increasing the distance between then drive and idler pulleys while maintaining the same belt length therefore increases the belt tension. It should be noted that the selected timing belt was observed to stretch very minimally under more tension than was used in the final design, so wear and fatigue were not prominent concerns. The brackets holding the drive pulley, idler pulley, tensioner, and stepper motor were all fastened to the same large piece of 48in×12in×0.25in aluminum.

Since the belt tension would place a constant and considerable force on these components, the choice to bolt them to the aluminum was to give the entire system more rigidity, and to place them more precisely than would have been possible if drilling directly into the MDF base.

It was an engineering design choice to give the Dynamixel motor and the drive pulleys separate mounting brackets. Otherwise, the drive pulley would be at the end of a cantilevered shaft. Since considerable belt tension is required, this could damage the motor, cause it to bind or miss steps, or otherwise malfunction catastrophically. The drive pulley's bracket maintains the tension in the belt, leaving the Dynamixel motor bracket solely responsible for providing a counter torque to hold the motor in place.

6.2 Battery Table Control

Robotis' Dynamixel XH430-V350-R was used to actuate the timing belt that drove the battery table. The Dynamixel motor is a precise, high torque, and compact actuator designed for high performance robotics applications. It has a variety of modes and is able to function as a DC motor, a servo motor (180° of precision control) and an extended servo motor (1080° of precision control) all while providing very high torque relative to its small size. [16]

In addition to its small size and high torque, the Dynamixel was chosen due to its ability to integrate into EveryFly's software smoothly. The Dynamixel is programmable in Python and can accept inputs through a variety of communication methods, including USB. This allowed the master program to directly control the battery table, Tello and linear actuators in Python.

The Dynamixel was connected to the battery table with a timing belt; the belt's length and the Dynamixel's gearing ratio resulted in 1in of motion for every 100° of rotation. This meant that positioning the batteries, which were exactly each 2 inches apart on the table, was elementary. To reach a different battery, the Dynamixel was simply rotated in increments of 200°.

To ensure that a fresh battery was always prepared and that the battery table knew where to insert the depleted battery being extracted from the Tello, the battery table controller kept track of the positions and order that the batteries were exchanged. This way, the table was always ready to accept a depleted battery and insert a fully charged battery.

6.3 Insertion Actuator Design

The battery insertion actuator serves a similar purpose to the extraction actuator and has a very similar design. A 200mm stroke length ballscrew linear actuator [13] controlled by the same arduino that controls the insertion actuator was employed. Unlike the case of the extraction actuator, the insertion actuator could not be placed directly on the MDF base without protruding excessively over the edge, as it would interfere with the battery table. The solution was to elevate the entire actuator, allowing it to be placed farther inward on the table without interfering with the battery table. A rendering of the bracket assembly that accomplished this is shown in Figure 31.

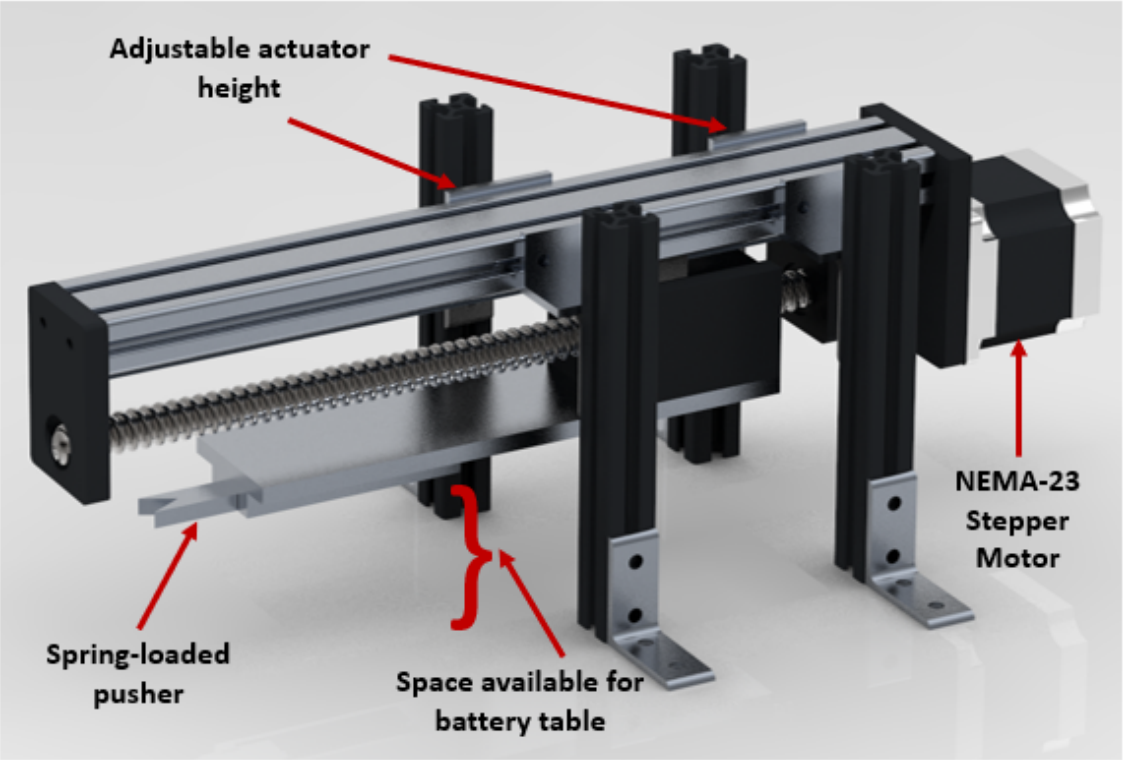
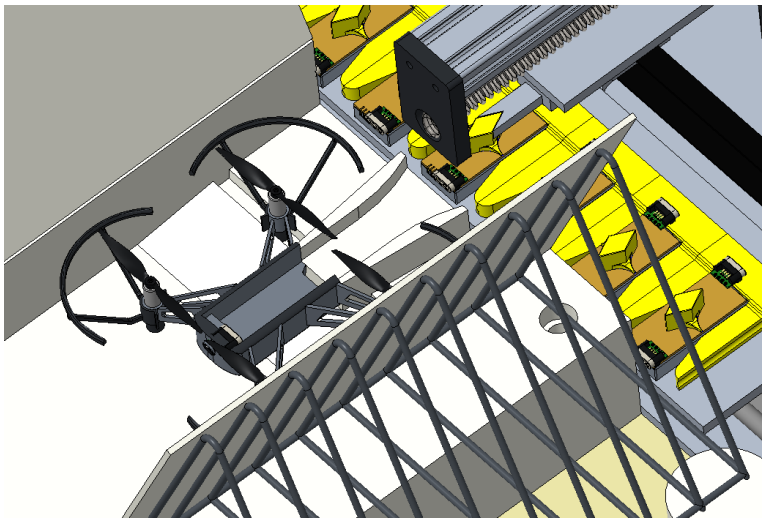
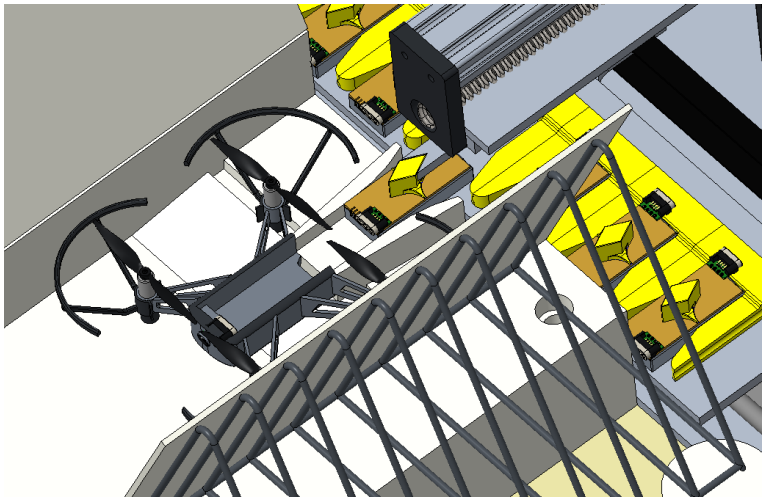


Figure 31: Rendering of the insertion actuator bracket assembly

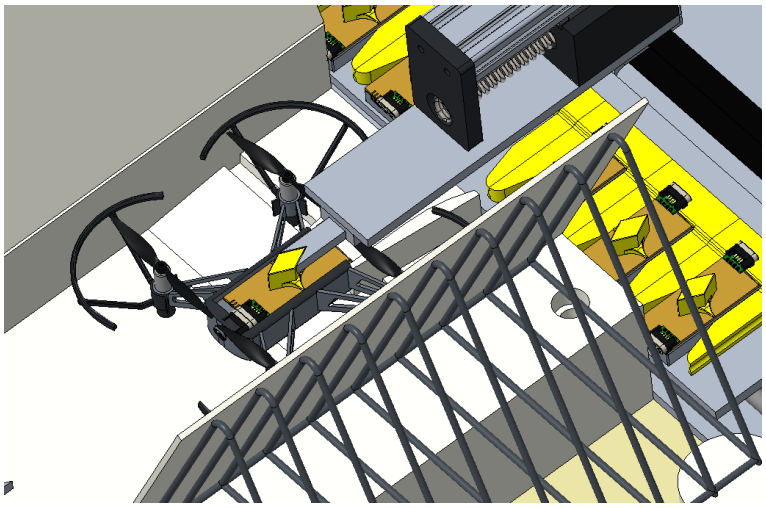
The process of battery insertion is shown in Figure **??**. Like the extraction actuator, the insertion actuator interfaces with the diamond geometry on the battery with a negative of the diamond shape. So long as the insertion actuator is properly adjusted in the z-direction, the diamond geometry affords a margin of error in the x-direction of half the width of the diamond. A margin of error in the y-direction is afforded by the spring loaded mechanism behind the negative diamond shape. This diamond design was chosen because it was easy to 3D print and manufacture a negative of it, and importantly it did not interfere with the drone rotors.



(a) Battery removed from charging slot



(b) Battery inserted into battery channel



(c) Battery inserted into Tello

Figure 32: Battery insertion schematic

Once the new battery is inserted into the drone, the insertion actuator retracts, and the Tello is free to take off and resume its autonomous operation.

7 Effects of COVID-19

Unfortunately, the construction of EverFly was halted due to the COVID-19 pandemic. Most of the system was built and proven to work before the onset of the virus, but some features and finishing touches were left incomplete:

- The wiring for the battery chargers was incomplete
- Although the GNC system was tested extensively on one QR code, the QR cube (with four QR codes) was not yet implemented
- Automatic power-on of the drone was not fully realized
- Sensors for state estimation, calibration, and safety, were not integrated
- Refactoring of EverFly codebase (e.g. 5.3.4)

The wiring of the battery chargers and the implementation of the QR cube were simply a matter of time, and were not expected to be a technical challenge. The automatic power on was a bit of a technical challenge, but it was in the process of being solved. As mentioned in section 2.3 the drone needed to be rigged to turn on automatically once it received a new battery. A Maxim DS1812 component was soldered onto the Tello's electrical circuit to simulate the function of the on button. Though it looked promising, the integration of the component was left incomplete.

Finally, in order to have a robust autonomous system, sensors were needed to estimate the state of EverFly at any point in time. Instead of blindly relying on the success of the autonomous actuators to exchange batteries, it would be valuable to have live feedback on the system. Photoelectric sensors could have been used to detect the locations of batteries as well as to verify successful extraction and insertion. Sensors would allow the system to verify its state and detect a failure. In addition, limit switches would have been integrated to prevent all actuators from extending beyond their physical bounds. They would also have been used to run calibration sequences for each linear actuator on startup. However, due to the pandemic, these sensors were not integrated into EverFly.

Despite these minor shortcomings, the system was built very close to completion, and is sufficient to prove the autonomous battery exchange concept. Perhaps once the pandemic subsides EverFly can be completed and perfected.

8 Results and Conclusion

The success of the EverFly system was demonstrated by its ability to perform the prescribed task: facilitating an autonomous mid-mission battery exchange. In this respect this thesis project was a success despite the abrupt dissolution of the school year due to COVID-19.

Extensive testing for each of EverFly's subsystems allowed isolated component failures to be quickly identified and remedied. By the March 19th Princeton University evacuation, all subsystems were operational and robust. After iterative tuning of the GNC PID gains, the system accurately landed the drone with a success rate of 100% over more than 30 GNC trials. The battery extraction mechanism demonstrated a success rate of about 90% over more than 30 extraction trials. The only observed point of failure was the maladjustment of the extraction arm height, which caused a loss of engagement with the battery's diamond extrusion. The straightforward design and workmanship of the battery table enabled it to function accurately and reliably. Once the battery table actuation

software was fine-tuned, it was able to accurately position the battery slots with a success rate of 100% over more than 40 positioning trials. The battery insertion mechanism did not encounter the misalignment issue that affected the extraction mechanism, and therefore proved to insert the new battery with a success rate of 100% over more than 30 insertion trials. Furthermore, the safety measure introduced by the spring-loaded end effector on the extraction and insertion arms worked as designed. In a number of trials, the end effectors prevented hardware damage in cases where the battery was caught or where either actuator was overextended.

Full system testing (i.e. the sequential actuation of the extraction mechanism, the battery table, and insertion mechanism) demonstrated successful integration of the subsystems. Though this system testing was limited to just over 10 trials, EverFly's success rate was a promising 90%. Given the initial success of the system, additional hardware tweaking and software tuning would have undoubtedly perfected EverFly's performance. A picture of EverFly is shown in 33 along with a link to a video demonstration.

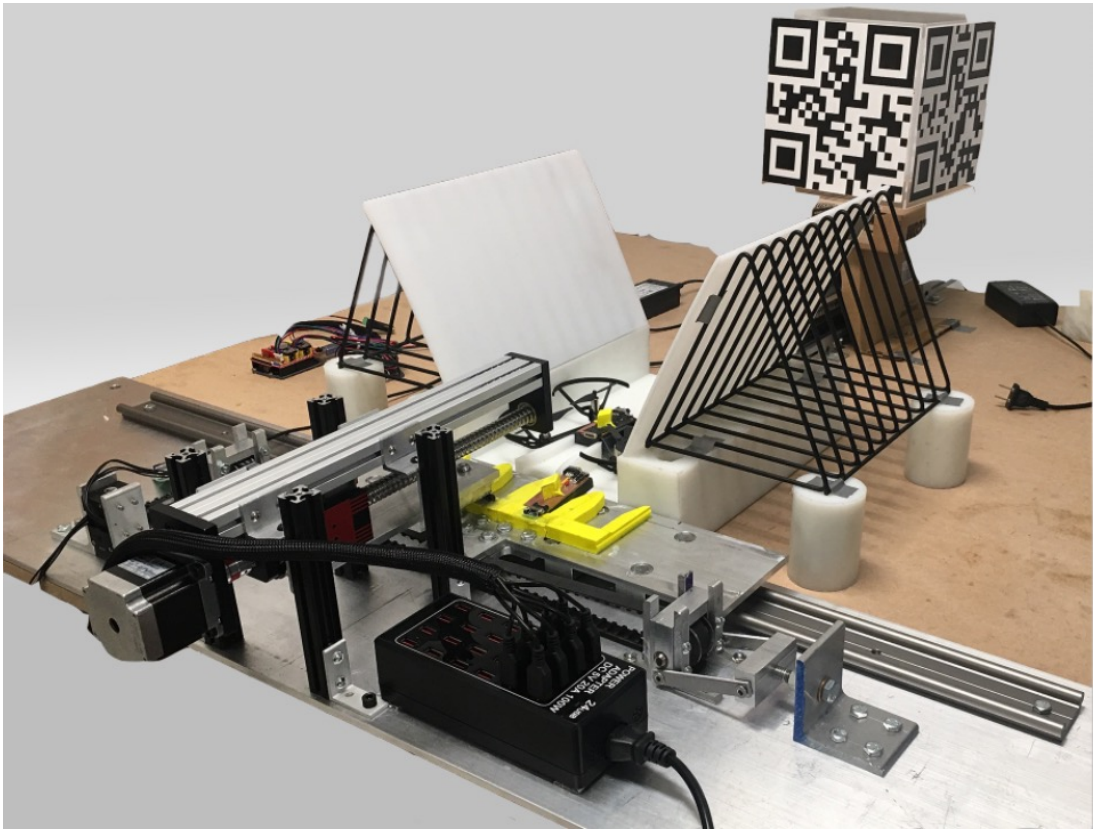


Figure 33: EverFly
video demonstration link: `https://ofekperes.github.io/EverFly/`

The EverFly prototype demonstrates that autonomous battery swapping is an ideal solution to the battery lifetime constraint on drone missions. Although completion of the project was interrupted by the onset of the COVID-19 pandemic, EverFly accomplished the overarching goal of executing a successful autonomous battery exchange. In just five months and with a $3000 budget, an autonomous battery exchange system was fabricated for a commercially available drone. Of course, EverFly was customized for the DJI Tello, but the proven concept can be applied to any quadrotor drone. Drone manufacturers can even modify their product lines to optimize for battery exchanging capabilities (e.g. magnetize the drone-battery connection or modify the accessibility of the drone's battery). Ideally, drone manufacturers would have the option to adopt a set of standards for their

drones in order to meet the requirements of a 'universal' battery exchange system. Thus, EverFly is a foundational innovative robotic technology that will enable drones to fly forever.

9 Future Work

Though EverFly was a rudimentary design for a specific drone model, it demonstrates the concept of a drone battery exchange system, upon which layers of complexity can be added. Some natural improvements to the EverFly concept are discussed below.

9.1 Multi-Drone Station Sharing

A simple yet powerful feature that was not implemented in EverFly is the sharing of the battery station by multiple drones. In theory, a single battery exchange station could service a multitude of drones for an indefinite amount of time. In order to achieve this, a station would need to be able to store and charge many more batteries, and the system of drones would need to communicate with each other and the station in order to exchange batteries in turn. This functionality would greatly improve the commercial prospects of EverFly.

9.2 Mobile Battery Station

One of the greatest assets autonomous Drones have is their mobility. A mobile EverFly station, perhaps mounted on a robotic rover, would allow for extreme long distance drone flight. Similar in function to an aircraft carrier, drones would be able to autonomously survey thousands of miles of land if they had a local refueling base nearby. To name only a few examples, this simple addition to the EverFly system would have applications in large scale security, wild-fire prevention, and street level package delivery.

9.3 Scaling of the Design for Larger Drones

Commercialization of drone technology will necessitate the use of larger drones to carry out missions beyond photography and surveillance. Larger drones will be designed to carry heavy payloads over potentially very long distances. As drones increase in size and capability, they will also increase in power demand making the EverFly system not only convenient but an integral part of their operation. If the EverFly system is to service these physically larger drones, the hardware that it employs must similarly be scaled up physically and in strength. This may take the form of high-strength and mass-produced, interchangeable parts.

References

[1] D. T. Company. https://www.dji.com/phantom-4.

[2] A. Robotics. https://auteldrones.com/products/evo.

[3] D. T. Company. https://www.dji.com/mavic-2.

[4] D. T. Company. https://www.dji.com/inspire-2.

[5] D. T. Company. https://store.dji.com/shop/spark-series.

[6] D. Fuentes, "Djitello github repository." https://github.com/damiafuentes/DJITelloPy.

[7] B. Galkin, J. Kibilda, and L. A. DaSilva, "Uavs as mobile infrastructure: Addressing battery lifetime," *IEEE Communications Magazine*, vol. 57, no. 6, pp. 132–137, 2019.

[8] Dallas Semiconductor, *5V Econo Reset with Active High Push-Pull Output*. DS1812.

[9] D. T. Company. https://www.dji.com/mavic-air-2.

[10] K. Mizokami, "Drones recharged by a laser could fly forever," *Popular Mechanics*, 2012.

[11] D. T. Company. https://store.dji.com/product/tello-edu.

[12] H. Technologies. https://www.hyte.pro/product/m417p.html.

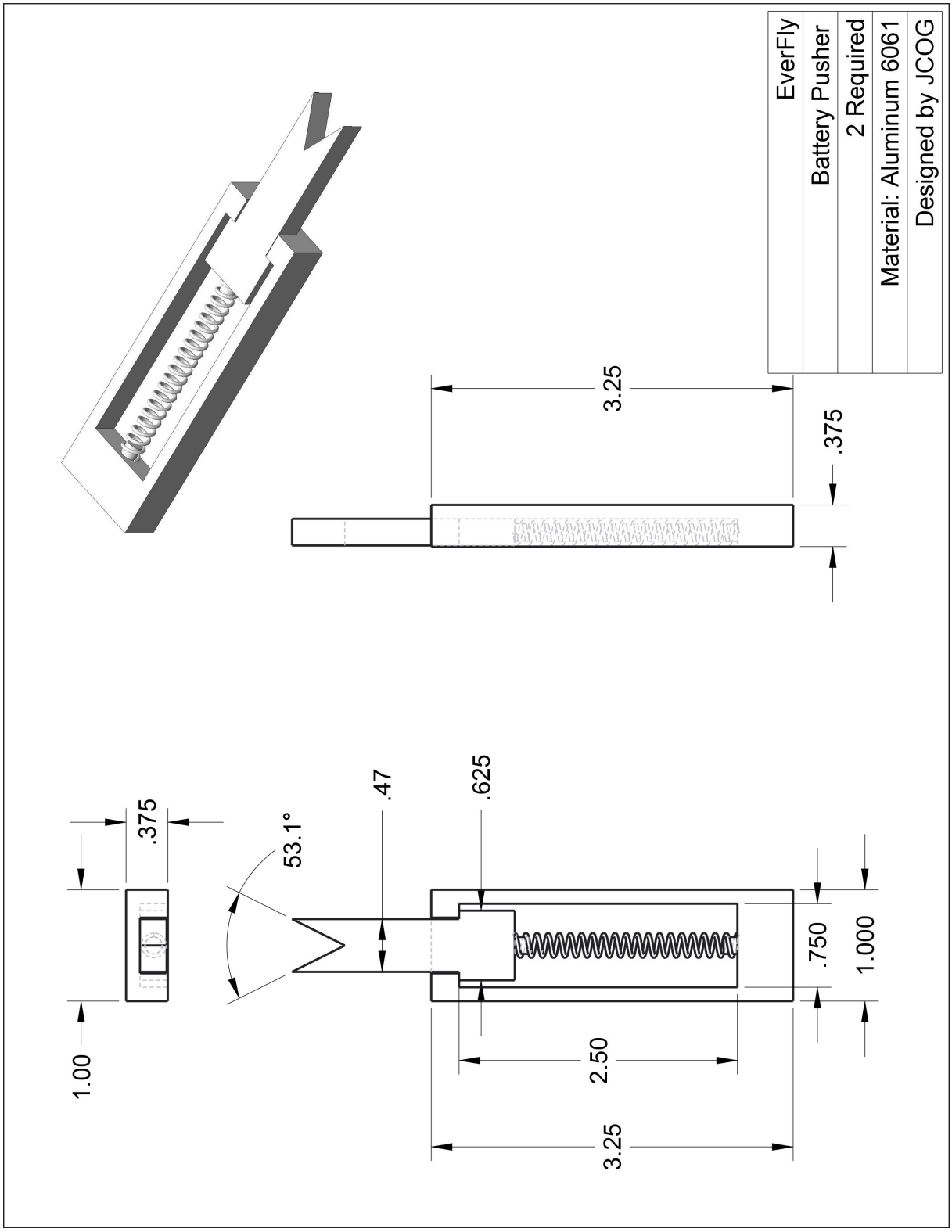
[13] F. Technology, "Series fsl40 high precision ball screw linear motion guide." https://www.fuyumotion.com/high-precision-ball-screw-linear-motion-guide.html.

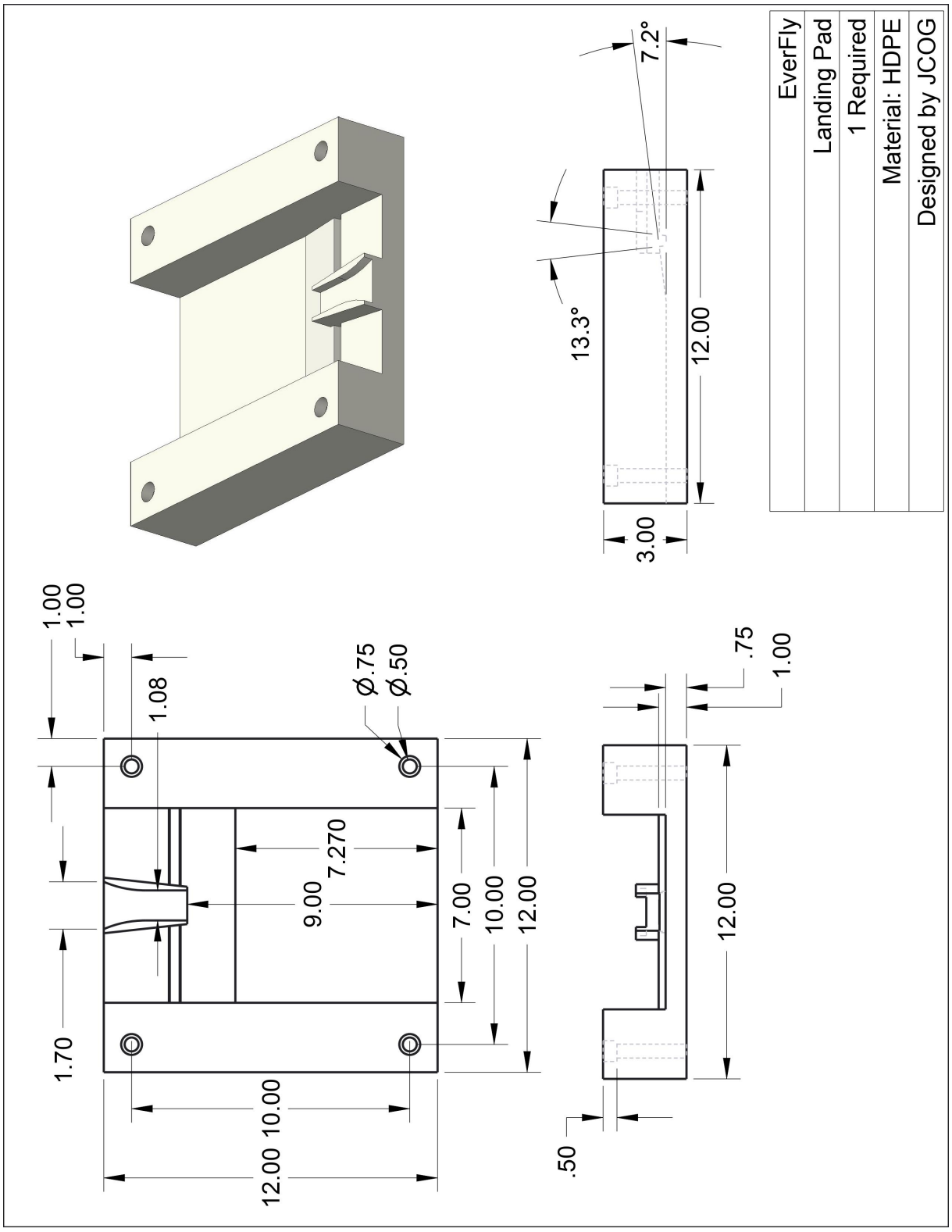
[14] Pololu, "Nema23 data sheet and product info." https://www.pololu.com/product/1478.

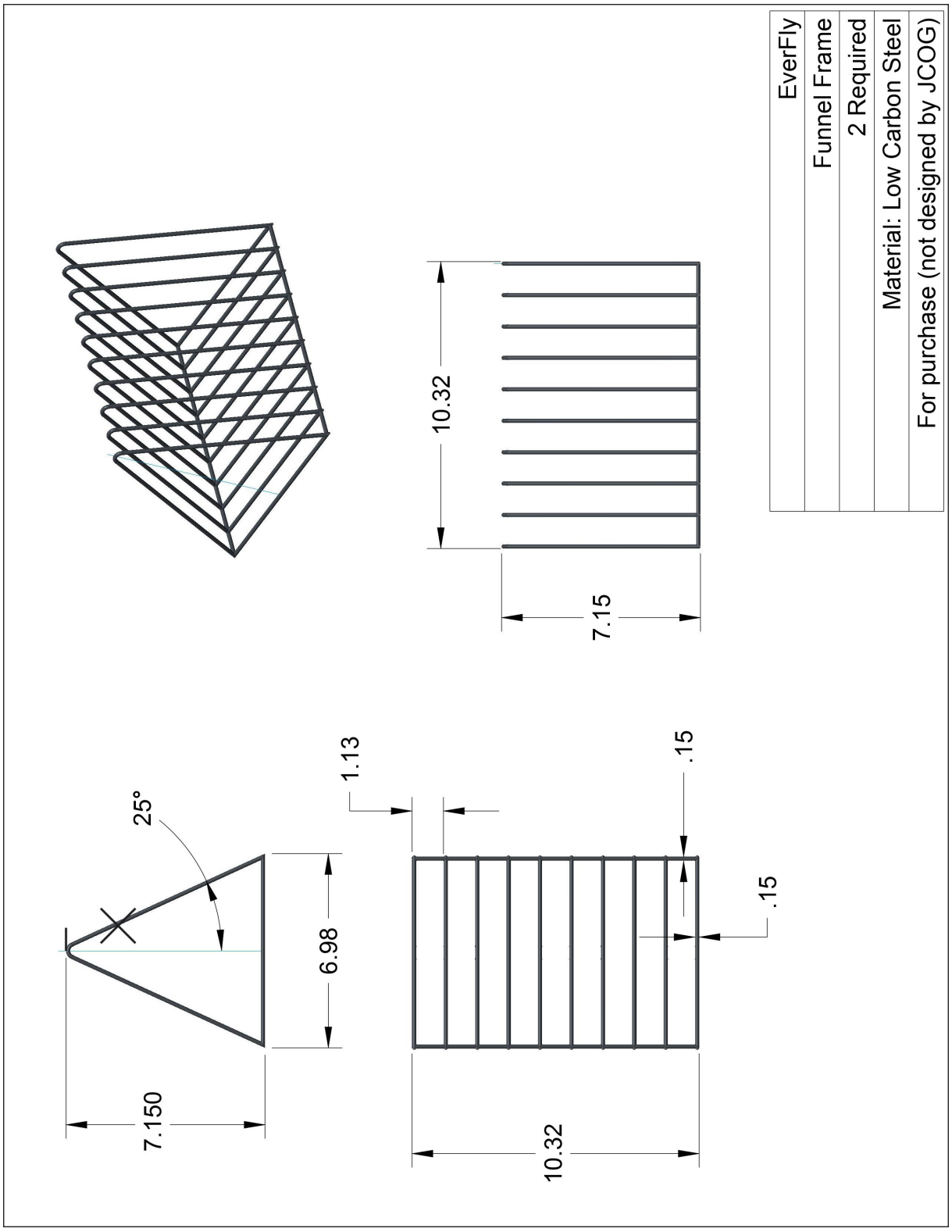
[15] Arduino, "Arduino stepper motor library." https://www.arduino.cc/en/reference/stepper.

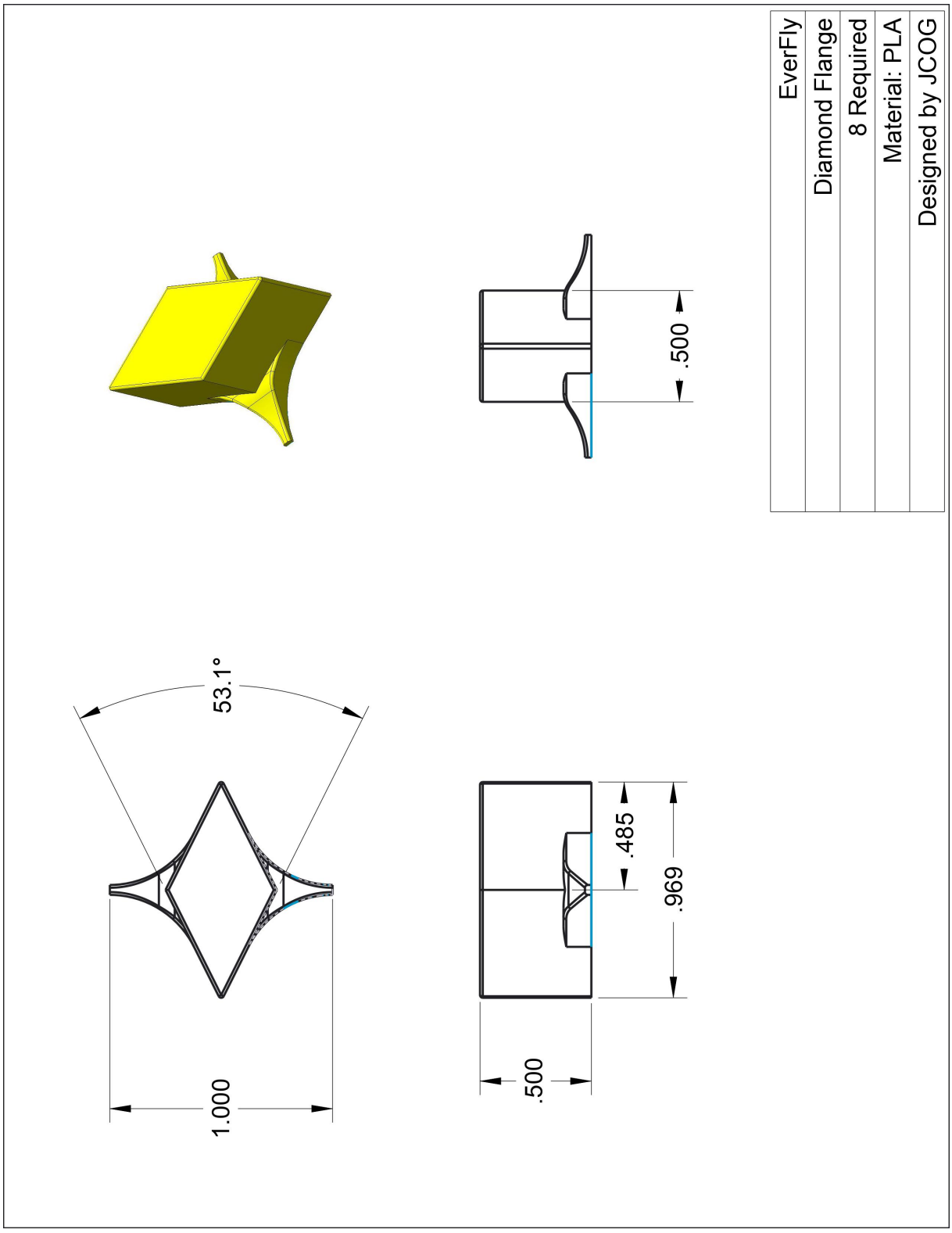
[16] ROBOTIS. http://www.robotis.us/dynamixel-xh430-v350-r/.

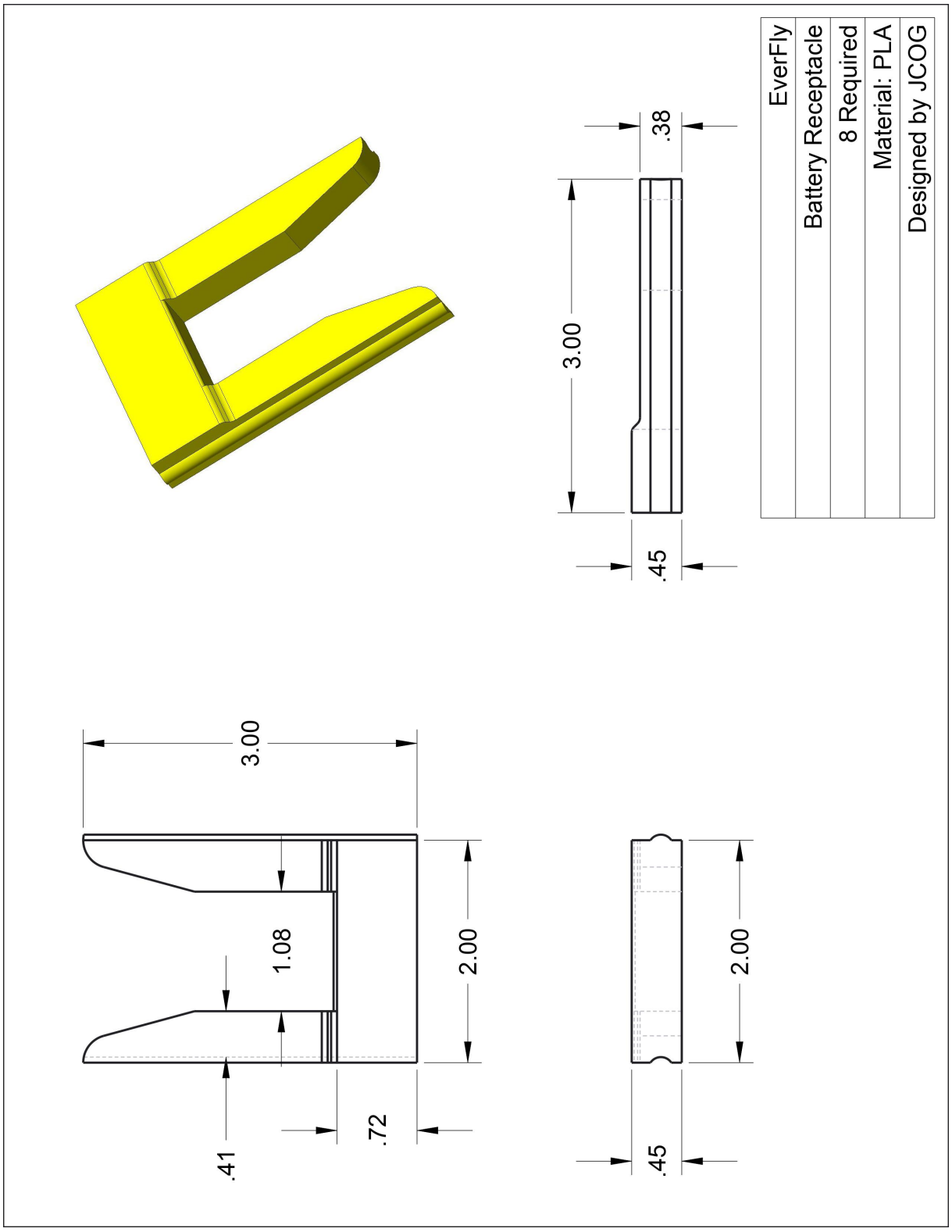
10 Appendix

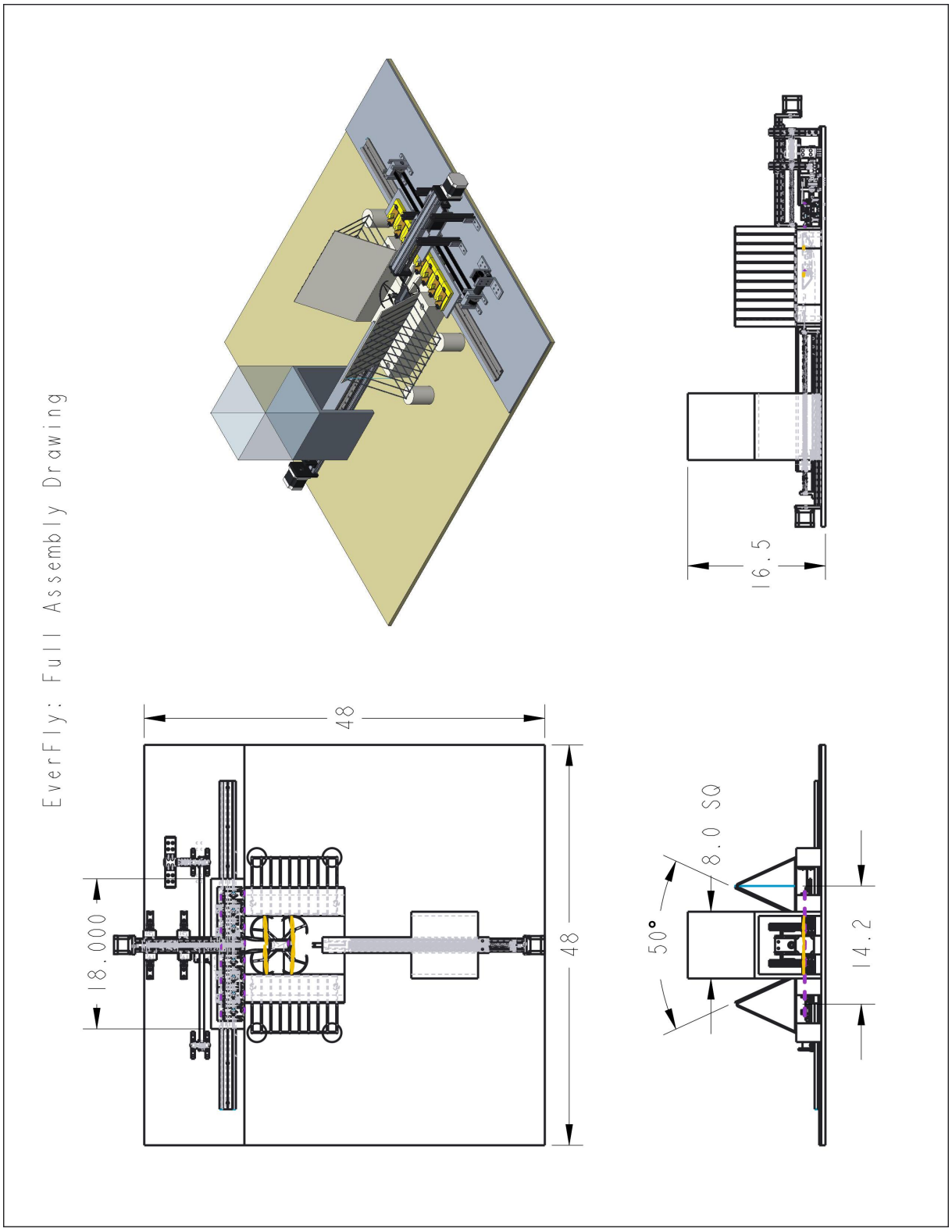

EverFly
Battery Pusher
2 Required
Material: Aluminum 6061
Designed by JCOG
3.25
.375
.375
53.1°
.47
.625
.750
1.000
1.00
2.50
3.25


EverFly
Landing Pad
1 Required
Material: HDPE
Designed by JCOG
7.2°
13.3°
12.00
3.00
1.00
1.00
1.08
Ø.75
Ø.50
7.270
9.00
7.00
10.00
12.00
1.70
12.00 10.00
.75
1.00
12.00
.50


EverFly
Funnel Frame
2 Required
Material: Low Carbon Steel
For purchase (not designed by JCOG)
10.32
7.15
25°
1.13
.15
.15
6.98
7.150
10.32


EverFly
Diamond Flange
8 Required
Material: PLA
Designed by JCOG
.500
53.1°
1.000
.485
.969
.500


EverFly
Battery Receptacle
8 Required
Material: PLA
Designed by JCOG
.38
3.00
.45
3.00
1.08
2.00
.41
.72
2.00
.45


EverFly: Full Assembly Drawing
16.5
48
48
18.000
8.0 SQ
50°
14.2

Listing 1: Python Master Program to control Tello's GNC system, Dynamixel, and Arduino Uno [6]

```
'''
This class will represent a drone object, which will be able to tell the Tello what to do
at a higher level (i.e. hover) and will also keep track of keyboard inputs.

Keyboard events are tracked using the python package pygame which is useful for displaying images and
responding to keyboard inputs
'''
import time
import pygame
from djitellopy import Tello
import cv2
import numpy as np
import imutils
import matplotlib.pyplot as plt
from pyzbar import pyzbar
from tello_utils import get_y, polygon_area, get_x, get_z_measurement, get_z, draw_april_tag_bounding_box, get_control_inputs, get_pid_control_inputs


from Dynamixel import Dynamixel
from ArduinoConnect import ArduinoConnect
from dynamixel_utils import *


SPEED = 50
FPS = 15
HAS_NOT_SEEN_QR_CODE = 0
TRACKING_QR_CODE = 1
LOST_QR_CODE = 2
Found_and_Centered_on_QR_Code = 3
landed = 4
class Drone:

    def __init__(self):
        self.forward_back_velocity = 0
        self.left_right_velocity = 0
        self.up_down_velocity = 0
        self.yaw_velocity = 0
        self.speed = 10
        self.rc_mode = True
        self.drone_state = 'landed'
        self.stage = HAS_NOT_SEEN_QR_CODE
        self.program_active = True
        self.tello = Tello()
        self.centered = False
        self.closeEnough = False
        self.land = False
        self.e_integral = np.array([0,0,0,0])
        self.e_prev = np.array([0,0,0,0])

        self.finished_controlling_actuators = False

        # Create Dynamixel and Arduino Objects

```

```
54          self.dxl = Dynamixel(1, '/dev/tty.usbserial-FT3WHRMU')
55          self.dxl.disableTorque()
56          self.dxl.setMode('extended_position')
57          self.dxl.write2byte(84, 3000)
58          self.dxl.enableTorque()
59          self.dxl.moveTo(166)
60
61          self.unoConnect = ArduinoConnect()
62          # Setup pygame window
63          pygame.init()
64          # self.screen = pygame.display.set_mode([960,720])
65          self.screen = pygame.display.set_mode([600,450])
66          pygame.display.set_caption("Tello Video Stream")
67          # create update timer
68          pygame.time.set_timer(pygame.USEREVENT + 1, 50)
69
70
71
72      # Code very similar to example.py from DJItello [github: ...
            https://github.com/damiafuentes/DJITelloPy]
73      def activate(self):
74
75          self.tello.connect()
76          self.tello.streamon()
77
78          print(self.tello.get_battery())
79
80          vs = self.tello.get_frame_read()
81          time.sleep(2)
82
83          while self.program_active:
84
85              for event in pygame.event.get():
86
87                  if event.type == pygame.USEREVENT + 1:
88                      self.update()
89                  elif event.type == pygame.QUIT:
90                      self.program_active = False
91                  elif event.type == pygame.KEYDOWN:
92                      if event.key == pygame.K_ESCAPE:
93                          self.program_active = False
94                      else:
95                          self.keydown(event.key)
96                  elif event.type == pygame.KEYUP:
97                      self.keyup(event.key)
98              if vs.stopped:
99                  vs.stop()
100                 break
101
102
103             frame = vs.frame
104             frame = imutils.resize(frame, width=600)
105
106
107             # Find QR Codes in Image
108             aprilTags = pyzbar.decode(frame)
109
110             if len(aprilTags) ≥ 1:
111                 aprilTag = aprilTags[0]
112             else:
```

```
113                 aprilTag = None
114             if aprilTag is not None:
115                 draw_april_tag_bounding_box(frame, aprilTag, ...
                        self.centered, self.closeEnough)
116             if not self.rc_mode:
117                 self.stageUpdate(len(aprilTags))
118                 self.controller(frame, aprilTag)
119 
120             # Prepare image for display in Pygame
121             self.screen.fill([0,0,0])
122             frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
123             frame = np.rot90(frame)
124             frame = np.flipud(frame)
125             frame = pygame.surfarray.make_surface(frame)
126             self.screen.blit(frame,(0,0))
127             pygame.display.update()
128 
129             time.sleep(1/FPS)
130 
131         print(self.tello.get_battery())
132         self.tello.end()
133         pygame.quit()
134 
135     # Functions to update actions
136     def keydown(self, key):
137         """ Update velocities based on key pressed
138         Arguments:
139             key: pygame key
140         """
141         if key == pygame.K_UP:  # set forward velocity
142             self.forward_back_velocity = SPEED
143         elif key == pygame.K_DOWN:  # set backward velocity
144             self.forward_back_velocity = -SPEED
145         elif key == pygame.K_LEFT:  # set left velocity
146             self.left_right_velocity = -SPEED
147         elif key == pygame.K_RIGHT:  # set right velocity
148             self.left_right_velocity = SPEED
149         elif key == pygame.K_w:  # set up velocity
150             self.up_down_velocity = SPEED
151         elif key == pygame.K_s:  # set down velocity
152             self.up_down_velocity = -SPEED
153         elif key == pygame.K_a:  # set yaw counter clockwise velocity
154             self.yaw_velocity = -SPEED
155         elif key == pygame.K_d:  # set yaw clockwise velocity
156             self.yaw_velocity = SPEED
157         # Toggle Mode between RC and Autonomous
158         elif key == pygame.K_m:
159             self.rc_mode = not self.rc_mode
160             print(self.rc_mode)
161             self.forward_back_velocity = 0
162             self.left_right_velocity = 0
163             self.up_down_velocity = 0
164             self.yaw_velocity = 0
165         # Begin Hover
166         elif key == pygame.K_h:
167             self.rc_mode = True
168             self.forward_back_velocity = 0
169             self.left_right_velocity = 0
170             self.up_down_velocity = 0
171             self.yaw_velocity = 0
```

```
172             print(self.rc_mode)
173             print("Hovering")
174 
175         # Reset state of autonomous mode and set drone to hover
176         elif key == pygame.K_r:
177             self.stage = HAS_NOT_SEEN_QR_CODE
178             self.rc_mode = True
179             self.forward_back_velocity = 0
180             self.left_right_velocity = 0
181             self.up_down_velocity = 0
182             self.yaw_velocity = 0
183             self.e_integral = np.array([0,0,0,0])
184             self.e_derivative = np.array([0,0,0,0])
185             self.land = False
186             self.finished_controlling_actuators = False
187             print(self.rc_mode)
188             print("Hovering")
189 
190 
191 
192     def keyup(self, key):
193         """ Update velocities based on key released
194         Arguments:
195             key: pygame key
196         """
197         if key == pygame.K_UP or key == pygame.K_DOWN:  # set zero ...
                forward/backward velocity
198             self.forward_back_velocity = 0
199         elif key == pygame.K_LEFT or key == pygame.K_RIGHT:  # set ...
                zero left/right velocity
200             self.left_right_velocity = 0
201         elif key == pygame.K_w or key == pygame.K_s:  # set zero ...
                up/down velocity
202             self.up_down_velocity = 0
203         elif key == pygame.K_a or key == pygame.K_d:  # set zero yaw ...
                velocity
204             self.yaw_velocity = 0
205         elif key == pygame.K_t:  # takeoff
206             self.tello.takeoff()
207             self.send_rc_control = True
208             self.land = False
209         elif key == pygame.K_l:  # land
210             self.tello.land()
211             self.land = True
212             self.send_rc_control = False
213 
214     def update(self):
215         """ Update routine. Send velocities to Tello."""
216         # if self.rc_mode:
217         command = "rc {} {} {} {}".format(self.left_right_velocity, ...
                self.forward_back_velocity, self.up_down_velocity,
218                                           self.yaw_velocity)
219         printInfo = False
220         self.tello.send_command_without_return(command, printInfo)
221         # self.tello.send_rc_control(self.left_right_velocity, ...
                self.forward_back_velocity, self.up_down_velocity,
222         #                            self.yaw_velocity)
223 
224     def controller(self, frame, aprilTag):
225         if self.stage == HAS_NOT_SEEN_QR_CODE:
```

```
226             self.rotate()
227         if self.stage == TRACKING_QR_CODE and aprilTag is not None:
228             self.left_right_velocity, self.forward_back_velocity, ...
                    self.up_down_velocity, self.yaw_velocity, ...
                    self.e_integral, self.e_prev, self.land = ...
                    get_pid_control_inputs(frame,aprilTag, ...
                    self.e_integral, self.e_prev)
229         if self.stage == LOST_QR_CODE:
230             self.backup()
231         if self.stage == Found_and_Centered_on_QR_Code:
232             command = "rc {} {} {} {}".format(0, 0, 0, 0)
233             printInfo = False
234             self.tello.send_command_without_return(command, printInfo)
235             self.tello.land()
236             self.stage = landed
237             self.land = True
238         if self.stage == landed:
239             command = "rc {} {} {} {}".format(0, 0, 0, 0)
240             printInfo = False
241             self.tello.send_command_without_return(command, printInfo)
242         if self.land and not self.finished_controlling_actuators:
243             # Move long actuator, shift dynamixel by 200, move short ...
                    actuator, send takeoff command
244             self.unoConnect.sweepLongPusher()
245             time.sleep(19)
246             self.dxl.moveTo(366)
247             self.unoConnect.sweepShortPusher()
248             self.finished_controlling_actuators = True
249
250
251
252
253     # Tracks what autonomous stage tello is currently at (e.g. has it ...
            seen the QR code yet?)
254     # And updates its control algorithms based on these milestones
255     def stageUpdate(self, numQRCodes):
256         # print("Number of QR Codes is: {}".format(numQRCodes))
257         # print("Stage in stageUpdate is: {}".format(self.stage))
258         if self.stage == HAS_NOT_SEEN_QR_CODE and numQRCodes ≥ 1:
259             self.stage = TRACKING_QR_CODE
260
261         elif self.stage == TRACKING_QR_CODE and numQRCodes == 0:
262             self.stage = LOST_QR_CODE
263
264         elif self.stage == LOST_QR_CODE and numQRCodes ≥ 1:
265             self.stage = TRACKING_QR_CODE
266         elif self.stage == TRACKING_QR_CODE and self.land == True:
267             self.stage = Found_and_Centered_on_QR_Code
268
269
270
271
272     def rotate(self):
273         print("Rotating")
274         self.left_right_velocity = 0
275         self.forward_back_velocity = 0
276         self.up_down_velocity = 0
277         self.yaw_velocity = 20
278
279
```

```
    def backup(self):
        print("Backing up")
        self.left_right_velocity = 0
        self.forward_back_velocity = -15
        self.up_down_velocity = 0
        self.yaw_velocity = 0



test_drone = Drone()
test_drone.tello.takeoff()
test_drone.tello.emergency()
```

Listing 2: Python Program to control the Dynamixel with high level functions that can be utilized in the Master Program

```
'''
This class will represent a drone object, which will be able to tell the Tello what to do
at a higher level (i.e. hover) and will also keep track of keyboard inputs.

Keyboard events are tracked using the python package pygame which is useful for displaying images and
responding to keyboard inputs
'''
import time
import pygame
from djitellopy import Tello
import cv2
import numpy as np
import imutils
import matplotlib.pyplot as plt
from pyzbar import pyzbar
from tello_utils import get_y, polygon_area, get_x, get_z_measurement, get_z, draw_april_tag_bounding_box, get_control_inputs, get_pid_control_inputs


from Dynamixel import Dynamixel
from ArduinoConnect import ArduinoConnect
from dynamixel_utils import *


SPEED = 50
FPS = 15
HAS_NOT_SEEN_QR_CODE = 0
TRACKING_QR_CODE = 1
LOST_QR_CODE = 2
Found_and_Centered_on_QR_Code = 3
landed = 4
class Drone:

    def __init__(self):
        self.forward_back_velocity = 0
        self.left_right_velocity = 0
        self.up_down_velocity = 0
        self.yaw_velocity = 0
```

```
38          self.speed = 10
39          self.rc_mode = True
40          self.drone_state = 'landed'
41          self.stage = HAS_NOT_SEEN_QR_CODE
42          self.program_active = True
43          self.tello = Tello()
44          self.centered = False
45          self.closeEnough = False
46          self.land = False
47          self.e_integral = np.array([0,0,0,0])
48          self.e_prev = np.array([0,0,0,0])
49
50          self.finished_controlling_actuators = False
51
52          # Create Dynamixel and Arduino Objects
53
54          self.dxl = Dynamixel(1, '/dev/tty.usbserial-FT3WHRMU')
55          self.dxl.disableTorque()
56          self.dxl.setMode('extended_position')
57          self.dxl.write2byte(84, 3000)
58          self.dxl.enableTorque()
59          self.dxl.moveTo(166)
60
61          self.unoConnect = ArduinoConnect()
62          # Setup pygame window
63          pygame.init()
64          # self.screen = pygame.display.set_mode([960,720])
65          self.screen = pygame.display.set_mode([600,450])
66          pygame.display.set_caption("Tello Video Stream")
67          # create update timer
68          pygame.time.set_timer(pygame.USEREVENT + 1, 50)
69
70
71
72      # Code very similar to example.py from DJItello [github: ...
            https://github.com/damiafuentes/DJITelloPy]
73      def activate(self):
74
75          self.tello.connect()
76          self.tello.streamon()
77
78          print(self.tello.get_battery())
79
80          vs = self.tello.get_frame_read()
81          time.sleep(2)
82
83          while self.program_active:
84
85              for event in pygame.event.get():
86
87                  if event.type == pygame.USEREVENT + 1:
88                      self.update()
89                  elif event.type == pygame.QUIT:
90                      self.program_active = False
91                  elif event.type == pygame.KEYDOWN:
92                      if event.key == pygame.K_ESCAPE:
93                          self.program_active = False
94                      else:
95                          self.keydown(event.key)
96                  elif event.type == pygame.KEYUP:
```

```
97                      self.keyup(event.key)
98              if vs.stopped:
99                  vs.stop()
100                 break
101
102
103             frame = vs.frame
104             frame = imutils.resize(frame, width=600)
105
106
107             # Find QR Codes in Image
108             aprilTags = pyzbar.decode(frame)
109
110             if len(aprilTags) ≥ 1:
111                 aprilTag = aprilTags[0]
112             else:
113                 aprilTag = None
114             if aprilTag is not None:
115                 draw_april_tag_bounding_box(frame, aprilTag, ...
                        self.centered, self.closeEnough)
116             if not self.rc_mode:
117                 self.stageUpdate(len(aprilTags))
118                 self.controller(frame, aprilTag)
119
120             # Prepare image for display in Pygame
121             self.screen.fill([0,0,0])
122             frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
123             frame = np.rot90(frame)
124             frame = np.flipud(frame)
125             frame = pygame.surfarray.make_surface(frame)
126             self.screen.blit(frame,(0,0))
127             pygame.display.update()
128
129             time.sleep(1/FPS)
130
131         print(self.tello.get_battery())
132         self.tello.end()
133         pygame.quit()
134
135     # Functions to update actions
136     def keydown(self, key):
137         """ Update velocities based on key pressed
138         Arguments:
139             key: pygame key
140         """
141         if key == pygame.K_UP:  # set forward velocity
142             self.forward_back_velocity = SPEED
143         elif key == pygame.K_DOWN:  # set backward velocity
144             self.forward_back_velocity = -SPEED
145         elif key == pygame.K_LEFT:  # set left velocity
146             self.left_right_velocity = -SPEED
147         elif key == pygame.K_RIGHT:  # set right velocity
148             self.left_right_velocity = SPEED
149         elif key == pygame.K_w:  # set up velocity
150             self.up_down_velocity = SPEED
151         elif key == pygame.K_s:  # set down velocity
152             self.up_down_velocity = -SPEED
153         elif key == pygame.K_a:  # set yaw counter clockwise velocity
154             self.yaw_velocity = -SPEED
155         elif key == pygame.K_d:  # set yaw clockwise velocity
```

```
156             self.yaw_velocity = SPEED
157         # Toggle Mode between RC and Autonomous
158         elif key == pygame.K_m:
159             self.rc_mode = not self.rc_mode
160             print(self.rc_mode)
161             self.forward_back_velocity = 0
162             self.left_right_velocity = 0
163             self.up_down_velocity = 0
164             self.yaw_velocity = 0
165         # Begin Hover
166         elif key == pygame.K_h:
167             self.rc_mode = True
168             self.forward_back_velocity = 0
169             self.left_right_velocity = 0
170             self.up_down_velocity = 0
171             self.yaw_velocity = 0
172             print(self.rc_mode)
173             print("Hovering")
174 
175         # Reset state of autonomous mode and set drone to hover
176         elif key == pygame.K_r:
177             self.stage = HAS_NOT_SEEN_QR_CODE
178             self.rc_mode = True
179             self.forward_back_velocity = 0
180             self.left_right_velocity = 0
181             self.up_down_velocity = 0
182             self.yaw_velocity = 0
183             self.e_integral = np.array([0,0,0,0])
184             self.e_derivative = np.array([0,0,0,0])
185             self.land = False
186             self.finished_controlling_actuators = False
187             print(self.rc_mode)
188             print("Hovering")
189 
190 
191 
192     def keyup(self, key):
193         """ Update velocities based on key released
194         Arguments:
195             key: pygame key
196         """
197         if key == pygame.K_UP or key == pygame.K_DOWN:  # set zero ...
                forward/backward velocity
198             self.forward_back_velocity = 0
199         elif key == pygame.K_LEFT or key == pygame.K_RIGHT:  # set ...
                zero left/right velocity
200             self.left_right_velocity = 0
201         elif key == pygame.K_w or key == pygame.K_s:  # set zero ...
                up/down velocity
202             self.up_down_velocity = 0
203         elif key == pygame.K_a or key == pygame.K_d:  # set zero yaw ...
                velocity
204             self.yaw_velocity = 0
205         elif key == pygame.K_t:  # takeoff
206             self.tello.takeoff()
207             self.send_rc_control = True
208             self.land = False
209         elif key == pygame.K_l:  # land
210             self.tello.land()
211             self.land = True
```

```
212             self.send_rc_control = False
213
214     def update(self):
215         """ Update routine. Send velocities to Tello."""
216         # if self.rc_mode:
217         command = "rc {} {} {} {}".format(self.left_right_velocity, ...
                self.forward_back_velocity, self.up_down_velocity,
218                                           self.yaw_velocity)
219         printInfo = False
220         self.tello.send_command_without_return(command, printInfo)
221         # self.tello.send_rc_control(self.left_right_velocity, ...
                self.forward_back_velocity, self.up_down_velocity,
222         #                            self.yaw_velocity)
223
224     def controller(self, frame, aprilTag):
225         if self.stage == HAS_NOT_SEEN_QR_CODE:
226             self.rotate()
227         if self.stage == TRACKING_QR_CODE and aprilTag is not None:
228             self.left_right_velocity, self.forward_back_velocity, ...
                    self.up_down_velocity, self.yaw_velocity, ...
                    self.e_integral, self.e_prev, self.land = ...
                    get_pid_control_inputs(frame,aprilTag, ...
                    self.e_integral, self.e_prev)
229         if self.stage == LOST_QR_CODE:
230             self.backup()
231         if self.stage == Found_and_Centered_on_QR_Code:
232             command = "rc {} {} {} {}".format(0, 0, 0, 0)
233             printInfo = False
234             self.tello.send_command_without_return(command, printInfo)
235             self.tello.land()
236             self.stage = landed
237             self.land = True
238         if self.stage == landed:
239             command = "rc {} {} {} {}".format(0, 0, 0, 0)
240             printInfo = False
241             self.tello.send_command_without_return(command, printInfo)
242         if self.land and not self.finished_controlling_actuators:
243             # Move long actuator, shift dynamixel by 200, move short ...
                    actuator, send takeoff command
244             self.unoConnect.sweepLongPusher()
245             time.sleep(19)
246             self.dxl.moveTo(366)
247             self.unoConnect.sweepShortPusher()
248             self.finished_controlling_actuators = True
249
250
251
252
253     # Tracks what autonomous stage tello is currently at (e.g. has it ...
            seen the QR code yet?)
254     # And updates its control algorithms based on these milestones
255     def stageUpdate(self, numQRCodes):
256         # print("Number of QR Codes is: {}".format(numQRCodes))
257         # print("Stage in stageUpdate is: {}".format(self.stage))
258         if self.stage == HAS_NOT_SEEN_QR_CODE and numQRCodes ≥ 1:
259             self.stage = TRACKING_QR_CODE
260
261         elif self.stage == TRACKING_QR_CODE and numQRCodes == 0:
262             self.stage = LOST_QR_CODE
263
```

```
264         elif self.stage == LOST_QR_CODE and numQRCodes ≥ 1:
265             self.stage = TRACKING_QR_CODE
266         elif self.stage == TRACKING_QR_CODE and self.land == True:
267             self.stage = Found_and_Centered_on_QR_Code
268
269
270
271
272     def rotate(self):
273         print("Rotating")
274         self.left_right_velocity = 0
275         self.forward_back_velocity = 0
276         self.up_down_velocity = 0
277         self.yaw_velocity = 20
278
279
280     def backup(self):
281         print("Backing up")
282         self.left_right_velocity = 0
283         self.forward_back_velocity = −15
284         self.up_down_velocity = 0
285         self.yaw_velocity = 0
286
287
288
289 test_drone = Drone()
290 test_drone.tello.takeoff()
291 test_drone.tello.emergency()
```

Listing 3: Python program that contained utility functions to assist in requisite calculations in the control algorithms for the Tello

```
1  '''
2  This class will represent a drone object, which will be able to tell ...
       the Tello what to do
3  at a higher level (i.e. hover) and will also keep track of keyboard ...
       inputs.
4
5  Keyboard events are tracked using the python package pygame which is ...
       useful for displaying images and
6  responding to keyboard inputs
7  '''
8  import time
9  import pygame
10 from djitellopy import Tello
11 import cv2
12 import numpy as np
13 import imutils
14 import matplotlib.pyplot as plt
15 from pyzbar import pyzbar
16 from tello_utils import get_y, polygon_area, get_x, ...
       get_z_measurement, get_z, draw_april_tag_bounding_box, ...
       get_control_inputs, get_pid_control_inputs
17
18
19 from Dynamixel import Dynamixel
20 from ArduinoConnect import ArduinoConnect
21 from dynamixel_utils import *
```

```
22
23
24 SPEED = 50
25 FPS = 15
26 HAS_NOT_SEEN_QR_CODE = 0
27 TRACKING_QR_CODE = 1
28 LOST_QR_CODE = 2
29 Found_and_Centered_on_QR_Code = 3
30 landed = 4
31 class Drone:
32
33     def __init__(self):
34         self.forward_back_velocity = 0
35         self.left_right_velocity = 0
36         self.up_down_velocity = 0
37         self.yaw_velocity = 0
38         self.speed = 10
39         self.rc_mode = True
40         self.drone_state = 'landed'
41         self.stage = HAS_NOT_SEEN_QR_CODE
42         self.program_active = True
43         self.tello = Tello()
44         self.centered = False
45         self.closeEnough = False
46         self.land = False
47         self.e_integral = np.array([0,0,0,0])
48         self.e_prev = np.array([0,0,0,0])
49
50         self.finished_controlling_actuators = False
51
52         # Create Dynamixel and Arduino Objects
53
54         self.dxl = Dynamixel(1, '/dev/tty.usbserial-FT3WHRMU')
55         self.dxl.disableTorque()
56         self.dxl.setMode('extended_position')
57         self.dxl.write2byte(84, 3000)
58         self.dxl.enableTorque()
59         self.dxl.moveTo(166)
60
61         self.unoConnect = ArduinoConnect()
62         # Setup pygame window
63         pygame.init()
64         # self.screen = pygame.display.set_mode([960,720])
65         self.screen = pygame.display.set_mode([600,450])
66         pygame.display.set_caption("Tello Video Stream")
67         # create update timer
68         pygame.time.set_timer(pygame.USEREVENT + 1, 50)
69
70
71
72     # Code very similar to example.py from DJItello [github: ...
           https://github.com/damiafuentes/DJITelloPy]
73     def activate(self):
74
75         self.tello.connect()
76         self.tello.streamon()
77
78         print(self.tello.get_battery())
79
80         vs = self.tello.get_frame_read()
```

```
81          time.sleep(2)
82
83          while self.program_active:
84
85              for event in pygame.event.get():
86
87                  if event.type == pygame.USEREVENT + 1:
88                      self.update()
89                  elif event.type == pygame.QUIT:
90                      self.program_active = False
91                  elif event.type == pygame.KEYDOWN:
92                      if event.key == pygame.K_ESCAPE:
93                          self.program_active = False
94                      else:
95                          self.keydown(event.key)
96                  elif event.type == pygame.KEYUP:
97                      self.keyup(event.key)
98              if vs.stopped:
99                  vs.stop()
100                 break
101
102
103             frame = vs.frame
104             frame = imutils.resize(frame, width=600)
105
106
107             # Find QR Codes in Image
108             aprilTags = pyzbar.decode(frame)
109
110             if len(aprilTags) ≥ 1:
111                 aprilTag = aprilTags[0]
112             else:
113                 aprilTag = None
114             if aprilTag is not None:
115                 draw_april_tag_bounding_box(frame, aprilTag, ...
                        self.centered, self.closeEnough)
116             if not self.rc_mode:
117                 self.stageUpdate(len(aprilTags))
118                 self.controller(frame, aprilTag)
119
120             # Prepare image for display in Pygame
121             self.screen.fill([0,0,0])
122             frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
123             frame = np.rot90(frame)
124             frame = np.flipud(frame)
125             frame = pygame.surfarray.make_surface(frame)
126             self.screen.blit(frame,(0,0))
127             pygame.display.update()
128
129             time.sleep(1/FPS)
130
131         print(self.tello.get_battery())
132         self.tello.end()
133         pygame.quit()
134
135     # Functions to update actions
136     def keydown(self, key):
137         """ Update velocities based on key pressed
138         Arguments:
139             key: pygame key
```

```
140         """
141         if key == pygame.K_UP:  # set forward velocity
142             self.forward_back_velocity = SPEED
143         elif key == pygame.K_DOWN:  # set backward velocity
144             self.forward_back_velocity = -SPEED
145         elif key == pygame.K_LEFT:  # set left velocity
146             self.left_right_velocity = -SPEED
147         elif key == pygame.K_RIGHT:  # set right velocity
148             self.left_right_velocity = SPEED
149         elif key == pygame.K_w:  # set up velocity
150             self.up_down_velocity = SPEED
151         elif key == pygame.K_s:  # set down velocity
152             self.up_down_velocity = -SPEED
153         elif key == pygame.K_a:  # set yaw counter clockwise velocity
154             self.yaw_velocity = -SPEED
155         elif key == pygame.K_d:  # set yaw clockwise velocity
156             self.yaw_velocity = SPEED
157         # Toggle Mode between RC and Autonomous
158         elif key == pygame.K_m:
159             self.rc_mode = not self.rc_mode
160             print(self.rc_mode)
161             self.forward_back_velocity = 0
162             self.left_right_velocity = 0
163             self.up_down_velocity = 0
164             self.yaw_velocity = 0
165         # Begin Hover
166         elif key == pygame.K_h:
167             self.rc_mode = True
168             self.forward_back_velocity = 0
169             self.left_right_velocity = 0
170             self.up_down_velocity = 0
171             self.yaw_velocity = 0
172             print(self.rc_mode)
173             print("Hovering")
174 
175         # Reset state of autonomous mode and set drone to hover
176         elif key == pygame.K_r:
177             self.stage = HAS_NOT_SEEN_QR_CODE
178             self.rc_mode = True
179             self.forward_back_velocity = 0
180             self.left_right_velocity = 0
181             self.up_down_velocity = 0
182             self.yaw_velocity = 0
183             self.e_integral = np.array([0,0,0,0])
184             self.e_derivative = np.array([0,0,0,0])
185             self.land = False
186             self.finished_controlling_actuators = False
187             print(self.rc_mode)
188             print("Hovering")
189 
190 
191 
192     def keyup(self, key):
193         """ Update velocities based on key released
194         Arguments:
195             key: pygame key
196         """
197         if key == pygame.K_UP or key == pygame.K_DOWN:  # set zero ...
                forward/backward velocity
198             self.forward_back_velocity = 0
```

```
199         elif key == pygame.K_LEFT or key == pygame.K_RIGHT:  # set ...
                zero left/right velocity
200             self.left_right_velocity = 0
201         elif key == pygame.K_w or key == pygame.K_s:  # set zero ...
                up/down velocity
202             self.up_down_velocity = 0
203         elif key == pygame.K_a or key == pygame.K_d:  # set zero yaw ...
                velocity
204             self.yaw_velocity = 0
205         elif key == pygame.K_t:  # takeoff
206             self.tello.takeoff()
207             self.send_rc_control = True
208             self.land = False
209         elif key == pygame.K_l:  # land
210             self.tello.land()
211             self.land = True
212             self.send_rc_control = False
213
214     def update(self):
215         """ Update routine. Send velocities to Tello."""
216         # if self.rc_mode:
217         command = "rc {} {} {} {}".format(self.left_right_velocity, ...
                self.forward_back_velocity, self.up_down_velocity,
218                                           self.yaw_velocity)
219         printInfo = False
220         self.tello.send_command_without_return(command, printInfo)
221         # self.tello.send_rc_control(self.left_right_velocity, ...
                self.forward_back_velocity, self.up_down_velocity,
222         #                           self.yaw_velocity)
223
224     def controller(self, frame, aprilTag):
225         if self.stage == HAS_NOT_SEEN_QR_CODE:
226             self.rotate()
227         if self.stage == TRACKING_QR_CODE and aprilTag is not None:
228             self.left_right_velocity, self.forward_back_velocity, ...
                    self.up_down_velocity, self.yaw_velocity, ...
                    self.e_integral, self.e_prev, self.land = ...
                    get_pid_control_inputs(frame,aprilTag, ...
                    self.e_integral, self.e_prev)
229         if self.stage == LOST_QR_CODE:
230             self.backup()
231         if self.stage == Found_and_Centered_on_QR_Code:
232             command = "rc {} {} {} {}".format(0, 0, 0, 0)
233             printInfo = False
234             self.tello.send_command_without_return(command, printInfo)
235             self.tello.land()
236             self.stage = landed
237             self.land = True
238         if self.stage == landed:
239             command = "rc {} {} {} {}".format(0, 0, 0, 0)
240             printInfo = False
241             self.tello.send_command_without_return(command, printInfo)
242         if self.land and not self.finished_controlling_actuators:
243             # Move long actuator, shift dynamixel by 200, move short ...
                    actuator, send takeoff command
244             self.unoConnect.sweepLongPusher()
245             time.sleep(19)
246             self.dxl.moveTo(366)
247             self.unoConnect.sweepShortPusher()
248             self.finished_controlling_actuators = True
```

```
249
250
251
252
253     # Tracks what autonomous stage tello is currently at (e.g. has it ...
            seen the QR code yet?)
254     # And updates its control algorithms based on these milestones
255     def stageUpdate(self, numQRCodes):
256         # print("Number of QR Codes is: {}".format(numQRCodes))
257         # print("Stage in stageUpdate is: {}".format(self.stage))
258         if self.stage == HAS_NOT_SEEN_QR_CODE and numQRCodes ≥ 1:
259             self.stage = TRACKING_QR_CODE
260
261         elif self.stage == TRACKING_QR_CODE and numQRCodes == 0:
262             self.stage = LOST_QR_CODE
263
264         elif self.stage == LOST_QR_CODE and numQRCodes ≥ 1:
265             self.stage = TRACKING_QR_CODE
266         elif self.stage == TRACKING_QR_CODE and self.land == True:
267             self.stage = Found_and_Centered_on_QR_Code
268
269
270
271
272     def rotate(self):
273         print("Rotating")
274         self.left_right_velocity = 0
275         self.forward_back_velocity = 0
276         self.up_down_velocity = 0
277         self.yaw_velocity = 20
278
279
280     def backup(self):
281         print("Backing up")
282         self.left_right_velocity = 0
283         self.forward_back_velocity = -15
284         self.up_down_velocity = 0
285         self.yaw_velocity = 0
286
287
288
289 test_drone = Drone()
290 test_drone.tello.takeoff()
291 test_drone.tello.emergency()
```

Listing 4: Python Code to Communicate with Arduino

```
1  import serial
2  import time
3
4  class ArduinoConnect:
5      def __init__(self):
6          # Fill in the string to be the port that the arduino is ...
               plugged into
7          print("Making Serial Connection")
8          self.ser = serial.Serial('/dev/cu.usbmodem14201', 9600)
9          print("Finished Making Serial Connection")
10         # Wait while arduino resets.
```

```
11          time.sleep(3)
12          # clear the serail buffer?
13          # self.ser.readline()
14
15      def __del__(self):
16          self.ser.close
17
18      def sweepLongPusher(self):
19          command = "<stepperx,sweep,0,0>".encode('utf-8')
20          self.ser.write(command)
21      def extendLongPusher(self):
22          command = "<stepperx,extend,0,0>".encode('utf-8')
23          self.ser.write(command)
24      def goHomeLongPusher(self):
25          command = "<stepperx,goHome,0,0>".encode('utf-8')
26          self.ser.write(command)
27
28      def sweepShortPusher(self):
29          command = "<steppery,sweep,0,0>".encode('utf-8')
30          self.ser.write(command)
31
32      def sendCommand(self, command):
33          self.ser.write(command)
34
35      def readArduino(self):
36          return self.ser.readline().decode('utf-8')
37
38      def lightOn(self):
39          self.ser.write("<light,turnOn,0,0>".encode('utf-8'))
40
41      def lightOff(self):
42          self.ser.write("<light,turnOff,0,0>".encode('utf-8'))
43
44  if __name__ == '__main__':
45
46      unoConnect = ArduinoConnect()
47      # unoConnect.sweepShortPusher()
48      # print(unoConnect.readArduino())
49      print("Calling sweep Long Pusher")
50      unoConnect.sweepLongPusher()
51      # print(unoConnect.readArduino())
52      # for i in range(20):
53      #     unoConnect.lightOn()
54      #     print(unoConnect.readArduino())
55      #     time.sleep(0.5)
56      #     unoConnect.lightOff()
57      #     print(unoConnect.readArduino())
```

Listing 5: Arduino Program that controls the linear actuators and communicates with the Master Program

```
1
2  #include <Stepper.h>
3  int led = 13;
4  long currentStepLocationX = 0;
5  long currentStepLocationY = 0;
6
7  const long maxLongStepperPos  = -200000-6500;
8  const long maxShortStepperPos = -100000;
```

```
9
10 // defines pin numbers for our stepper motors
11
12 const int stepperXPin = 2;
13 const int directionXPin  = 5;
14
15 const int stepperYPin = 3;
16 const int directionYPin  = 6;
17
18 const int stepperZPin = 4;
19 const int directionZPin  = 7;
20
21 // Enable Pin
22 const int enPin = 8;
23
24 // Create Stepper Motor Objects
25 // stepperX is the long linear actuator
26 Stepper stepperX = Stepper(200, stepperXPin, directionXPin);
27 // stepperY is the short linear actuator
28 Stepper stepperY = Stepper(200, stepperYPin, directionYPin);
29 // Define constants necessary for string parsing
30 const byte numChars = 64;
31 char receivedChars[numChars];
32 // temporary array for use when parsing
33 char tempChars[numChars];
34
35 // variables to hold the parsed data
36 // char messageFromPC[numChars] = {0};
37 char thingToControl[numChars] = {0};
38 char action[numChars] = {0};
39 int integerInput = 0;
40 long longInput   = 0;
41 float floatInput = 0.0;
42
43 // Status of whether or not a new string was found
44 boolean newData = false;
45
46 //============
47
48 void setup() {
49     Serial.begin(9600);
50     // Serial.println("This demo expects 4 pieces of data — a string, ...
       a string, an integer and a floating point value");
51     // Serial.println("Enter data in this style <Thing To Control ...
       (e.g. stepperX), Action (e.g. setSpeed), Int: 2,  Float: ...
       24.7> ");
52     // Serial.println();
53
54     pinMode(led,OUTPUT);
55
56
57      // Sets the two pins as Outputs
58
59     pinMode(stepperXPin,OUTPUT);
60     pinMode(directionXPin,OUTPUT);
61
62     pinMode(stepperYPin,OUTPUT);
63     pinMode(directionYPin,OUTPUT);
64
65     pinMode(stepperZPin,OUTPUT);
```

```
66      pinMode(directionZPin,OUTPUT);
67
68      pinMode(enPin,OUTPUT);
69      digitalWrite(enPin,LOW);
70
71      digitalWrite(directionXPin,HIGH);
72      digitalWrite(directionYPin,HIGH); // Used to be on LOW rather ...
            than HIGH
73      digitalWrite(directionZPin,HIGH);
74
75      stepperX.setSpeed(5000);
76      stepperY.setSpeed(5000);
77  }
78
79  //=============
80
81  void loop() {
82      parseString();
83
84  }
85
86  //=============
87
88  void recvWithStartEndMarkers() {
89      static boolean recvInProgress = false;
90      static byte ndx = 0;
91      char startMarker = '<';
92      char endMarker = '>';
93      char rc;
94
95      while (Serial.available() > 0 && newData == false) {
96          rc = Serial.read();
97
98          if (recvInProgress == true) {
99              if (rc != endMarker) {
100                 receivedChars[ndx] = rc;
101                 ndx++;
102                 if (ndx ≥ numChars) {
103                     ndx = numChars − 1;
104                 }
105             }
106             else {
107                 receivedChars[ndx] = '\0'; // terminate the string
108                 recvInProgress = false;
109                 ndx = 0;
110                 newData = true;
111             }
112         }
113
114         else if (rc == startMarker) {
115             recvInProgress = true;
116         }
117     }
118 }
119
120 //=============
121
122 void parseData() {      // split the data into its parts
123
124     char * strtokIndx; // this is used by strtok() as an index
```

```
125
126     strtokIndx = strtok(tempChars, ","); // get the first part - the ...
           string that says which motor/light
127     strcpy(thingToControl, strtokIndx);  // copy it to thingToControl
128
129     strtokIndx = strtok(NULL, ",");      // this continues where the ...
           previous call left off
130     strcpy(action, strtokIndx);          // copy it to thingToControl
131
132
133     // strtokIndx = strtok(NULL, ",");
134     // integerInput = atoi(strtokIndx);    // convert this part to an ...
           integer
135
136     strtokIndx = strtok(NULL, ",");
137     longInput = atol(strtokIndx);    // convert this part to an long
138
139     strtokIndx = strtok(NULL, ",");
140     floatInput = atof(strtokIndx);     // convert this part to a float
141
142 }
143
144 //============
145
146 void showParsedData() {
147     Serial.print("Thing To Control:");
148     Serial.println(thingToControl);
149     Serial.print("Action:");
150     Serial.println(action);
151     Serial.print("Integer Input:");
152     Serial.println(integerInput);
153     Serial.print("Long Input:");
154     Serial.println(longInput);
155     Serial.print("Float Input:");
156     Serial.println(floatInput);
157     Serial.println();
158 }
159
160 void parseString()
161 {
162     recvWithStartEndMarkers();
163     if (newData == true) {
164         strcpy(tempChars, receivedChars);
165             // this temporary copy is necessary to protect the ...
                   original data
166             //   because strtok() used in parseData() replaces the ...
                   commas with \0
167         parseData();
168         // showParsedData();
169         actionHandler();
170         newData = false;
171     }
172 }
173
174 void updateStepCounter(long steps, char stepper)
175 {
176     if (stepper == 'x')
177     {
178         currentStepLocationX += steps;
179         if (currentStepLocationX % 10000 == 0)
```

```
180         {
181             // Serial.print("Current X Step Location:");
182             // Serial.println(currentStepLocationX);
183         }
184     }
185 
186     if (stepper == 'y')
187     {
188         currentStepLocationY += steps;
189         if (currentStepLocationY % 10000 == 0)
190         {
191             // Serial.print("Current Y Step Location:");
192             // Serial.println(currentStepLocationY);
193         }
194     }
195 
196 }
197 
198 void moveSteps(long steps, char stepper)
199 {
200     int stepSize = 500;
201     long stepsRemaining = steps;
202     Stepper curStepper = stepperX;
203     if (stepper == 'x')
204     {
205         // Serial.println("\nI've picked StepperX\n");
206         curStepper = stepperX;
207     }
208     if (stepper == 'y')
209     {
210         // Serial.println("\nI've picked StepperY\n");
211         curStepper = stepperY;
212     }
213 
214     // Max step size stepper motor can accept
215     while (abs(stepsRemaining) > stepSize)
216     {
217         if (stepsRemaining % 1000 == 0)
218         {
219             // Serial.print("Steps Remaining:");
220             // Serial.println(stepsRemaining);
221         }
222         if (stepsRemaining > 0)
223         {
224             curStepper.step(stepSize);
225             stepsRemaining = stepsRemaining - stepSize;
226             updateStepCounter(stepSize, stepper);
227         }
228         else if (stepsRemaining < 0)
229         {
230             curStepper.step(-1*stepSize);
231             stepsRemaining = stepsRemaining + stepSize;
232             updateStepCounter(-1*stepSize, stepper);
233         }
234     }
235     // Have only a number of steps within the size of an int left to take
236     // Serial.print("Steps Remaining:");
237     // Serial.println(stepsRemaining);
238     curStepper.step(stepsRemaining);
239 
```

```
240     updateStepCounter(stepsRemaining, stepper);
241     stepsRemaining = stepsRemaining - stepsRemaining;
242     // Serial.print("Steps Remaining:");
243     // Serial.println(stepsRemaining);
244 
245 }
246 void returnToStart(char input)
247 {
248     if (input == 'x')
249     {
250         moveSteps(-1*currentStepLocationX, input);
251     }
252 
253     if (input == 'y')
254     {
255         moveSteps(-1*currentStepLocationY, input);
256     }
257 }
258 // x is long actuator, y is short
259 void extendToEnd(char input)
260 {
261     // About -230000 steps to get from one end to the other
262     if (input == 'x')
263     {
264         moveSteps(maxLongStepperPos - currentStepLocationX, input);
265     }
266 
267     if (input == 'y')
268     {
269         moveSteps(maxShortStepperPos - currentStepLocationY, input);
270     }
271 }
272 
273 void extendAndRetract(char input)
274 {
275     extendToEnd(input);
276     returnToStart(input);
277     if (input == 'x')
278     {
279         Serial.println("Long Sweep Complete");
280     }
281     else if (input == 'y')
282     {
283         Serial.println("Short Sweep Complete");
284     }
285 }
286 void actionHandler()
287 {
288     // Serial.println("In Action Handler");
289     if (strcmp(thingToControl, "light") == 0)
290     {
291         // Serial.println("Controlling Light");
292         // digitalWrite(led, HIGH);
293         if (strcmp(action, "turnOn") == 0)
294         {
295             // Serial.println("Turning Light On");
296             digitalWrite(led, HIGH);
297 //            delay(1000);
298             Serial.println("Finished Turning Light On");
299         }
```

```
300         if (strcmp(action, "turnOff") == 0)
301         {
302             // Serial.println("Turning Light Off");
303             digitalWrite(led, LOW);
304             Serial.println("Finished Turning Light Off");
305         }
306     }
307 
308     else if (strcmp(thingToControl, "stepperx") == 0)
309     {
310         if (strcmp(action, "setSpeed") == 0)
311         {
312             stepperX.setSpeed(longInput);
313         }
314         if (strcmp(action, "step") == 0)
315         {
316             moveSteps(longInput, 'x');
317             // stepperX.step(longInput);
318 
319         }
320         if (strcmp(action, "goHome") == 0)
321         {
322             returnToStart('x');
323 
324         }
325         if (strcmp(action, "extend") == 0)
326         {
327             extendToEnd('x');
328         }
329         if (strcmp(action, "sweep") == 0)
330         {
331             extendAndRetract('x');
332         }
333     }
334 
335     else if (strcmp(thingToControl, "steppery") == 0)
336     {
337         if (strcmp(action, "setSpeed") == 0)
338         {
339             stepperY.setSpeed(longInput);
340         }
341         if (strcmp(action, "step") == 0)
342         {
343             moveSteps(longInput,'y');
344             // updateStepCounter(longInput);
345             // stepperY.step(longInput);
346         }
347         if (strcmp(action, "goHome") == 0)
348         {
349 
350             returnToStart('y');
351         }
352         if (strcmp(action, "extend") == 0)
353         {
354             extendToEnd('y');
355 
356         }
357         if (strcmp(action, "sweep") == 0)
358         {
359             extendAndRetract('y');
```

```
360
361                     }
362             }
363     }
```