B9: Hawkeye

Authors: Alvin Shek, Siddesh Nageswaran, Vedant Parekh
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—We created a drone capable of tracking a target in real time as they move across the ground. The drone has a camera that streams its live video feed to a display.**

***Index Terms*–computer vision, state estimation, motion planning, color filtering, kalman filter**

1 INTRODUCTION

Over the last few years, drones have become increasingly popular for both recreational and professional videography. Vloggers can take breathtaking aerial shots of their escapades through mountains and fjords. And during search-and-rescue missions, rescuers can use drone footage to identify victims and monitor their own surroundings to call for backup if needed. Yet, despite the numerous use-cases, one bottleneck stands in the way when it comes to capturing drone footage: manual control. Maneuvering a drone takes skill and concentration. For vloggers, this can be an inconvenience while for mission critical operations, this can prove to be a fatal distraction. Moreover, any task involving user control always has the potential for user error. Thus, the goal of Hawkeye is to be an autonomous drone videographer that prioritizes video quality and tracking accuracy. Since users are putting full faith into Hawkeye's tracking algorithm, the most critical requirements are to update the user position frequently and ensure that the user is centered to the frame at all times. Hawkeye aims to:

- Capture video of a target to identify a target's current position and future motion path
- Use this data for motion planning to follow the target. The target center should be within the frame for 90% of all frames and the drone should maintain reasonable stability (defined in the next section) across 90% of all 30 frame windows
- Stream video of the target to a display

This takes all the hassle out of the user's hands and enables them to focus on the task at hand. The drone will automatically shoot footage and users can either monitor the feed in real-time or select their best frames afterwards.

There are other drones that have recently arrived in the market which also aim to provide autonomous tracking of a target, such as the Skydio 2 and DJI Mavic Air 2 [1]. However, these drones have a few shortcomings.

1. **Price**. These drones go for around $800 - $1,400 and come with incredibly sophisticated 45MP+ cameras. Hawkeye uses a simple 8MP camera and relatively inexpensive compute to make drone videography cheap and accessible to all.
2. **User Convenience**. The competitors rely on a smartphone app to control the drone's tracking, which goes against our hands-free, minimally intrusive philosophy by forcing the user to live on their phone rather than in the moment. On the other hand, our design is controlled entirely by two simple and intuitive switches.

2 DESIGN REQUIREMENTS

Target Detection uses computer vision to predict first whether or not a target is present at all and second the center pixel location of the target a given image. Our entire target tracking and motion planning stacks depend on this initial detection, so this requirement is to guarantee its accuracy. We define performance in terms of false positive (FP) and false negative (FN) rates. Specifically:

$$FN = \frac{Detections}{Negatives}$$
$$FP = \frac{No\ detections}{Positives}$$

where *Negatives* are images with no target present, and *Positives* are images with targets. We aimed to satisfy a 2% false positive rate and 10% false negative rate. The reason we prioritized a lower false positive rate is because while the drone can rely on its previous motion plan in the case where the target is not detected for a frame, its path estimates get completely thrown off when a random point is misidentified as the target.

For our second target detection requirement, we define performance in terms of mean distance between predicted and true target pixel position:

$$\frac{1}{N}\sqrt{(\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2}$$

where the true position (x_i, y_i) for the ith image was hand-labeled by us. To test this, the drone was flown manually with remote control, allowing us to test computer vision independently of motion planning.

From our tests, targets often represent only around a 15 x 30 pixel rectangle from our altitude. We define error with respect to this rectangle's hypotenuse of 33.5 pixels. We expected to correctly identify the target's center pixel within 50% of the hypotenuse, which is 16.75 pixels.

Target Tracking involves both target detection and the drone's motion planning. Performance here is simply measured by the percentage of frames where the target is located in the frame. We aimed for the target to be present in 90% of all frames across three testing conditions: walking target, running target and mixed motion.

Drone Stability also involves both target detection and the drone's motion planning. The key is for drone to remain at fixed positions as long as possible so as to reduce camera jitter. We define stability by taking the standard deviation of the target's x and y pixel position across 30 frame windows. The window is deemed stable if:

$$std(x) \leq \frac{\text{frame width}}{6} \quad \text{AND} \quad std(y) \leq \frac{\text{frame height}}{6}$$

Where frame width is 426 pixels and frame height is 240 pixels. We aimed for 90% of all 30 frame windows to be deemed stable. Once again, this is across three testing conditions: walking target, running target and mixed motion.

Communication Bandwidth will have hard limits since we have offloaded our computation to a ground computer. Our transmitted data must not exceed our Realtek RTL8188CUS-GR wifi module's limit of 300 Megabits per second at either 2.4 or 5GHz frequency [11]. Our Pixhawk flight controller expects control commands to stream in at a minimum of 2Hz, so the latency between successive flight commands cannot exceed 0.5 seconds for a given timestep. This means we need a bare minimum processing FPS of 2 FPS, but we aimed for an FPS of 5.

Power Consumption and Flight Time Our requirement for flight time is 10 minutes as a lower bound, and we picked a 5100 mAH battery to achieve this. We calculate projected power consumption and overall flight time based on the following information:

- **Motors** 4 motors consuming 170W to lift 1kg (Model: AC 2830, 850 kV) [5]
- **Mass** 1282g for bare drone and battery and 300g for sensors and compute [5]
- **Battery** 5100 mAH battery at 11.1V

Assuming we can only use 80% of the battery for safety reasons, we calculate the following [3]:

$$\begin{aligned}\text{Amp draw} &= \frac{170W}{1kg} * \frac{1282 + 300g}{1000g/kg} * \frac{1}{11.1V} \\ &= 24.23A \\ \text{Flight time} &= \frac{\text{capacity}}{\text{Amp draw}} \\ &= \frac{.8 * 5100mAH}{24.23 * 1000mAH} * \frac{60 \text{ min}}{H} \\ &= 10.10 \text{ min}\end{aligned}$$

Our flight time estimate is a lower bound since the compute and sensors will most likely not weigh 300g, and the 170Wkg assumes constant ascent (not hovering).

2.1 Scope

In order to meet these requirements within the time period, these are the simplifying assumptions we made:

- The target will be wearing a red shirt
- The drone will operate in open field without obstacles
- There will be WiFi access in test environment
- Daytime conditions with minimal wind
- The drone will be limited to tracking one person
- Flight height no more than 30 feet

3 ARCHITECTURE OVERVIEW

Our system, which is shown in Fig. 1, is broken into three groups:

1. Drone Compute
2. User Compute
3. Display Device

The drone compute contains all the hardware that will be on the drone to help navigate it. The RaspiCamV2 on the drone collects a live video feed and sends it to an RPi 4 through the MIPI Camera Serial Interface 2. The RPi then sends the video it receives to the Jetson TX1 (over WiFi using ROS) for processing.

The TX1 uses the video as input for its algorithms to keep track of the position of the target, predict their future path, and create a motion plan for the drone. Based on this, the TX1 returns the flight position commands back to the RPi over WiFi using ROS. The RPi receives these position commands and can forward it to the flight controller of the drone via serial. The flight controller can then use these to guide the drone to the correct position. We do the bulk of the processing on the ground rather than on the drone in order to reduce power consumption on the drone and maximize flight time.

The video that the TX1 receives is sent to a display via HDMI for the user to view. We have also hooked up switches to the TX1 for user input that which are processed through GPIO interface. The display and the switches are powered by the TX1 while the TX1 is powered by a rechargeable LiPo battery while the drone is powered by a 5100 mAH rechargeable battery. A high-level description of the division of computation between the important subsystems is as follows:

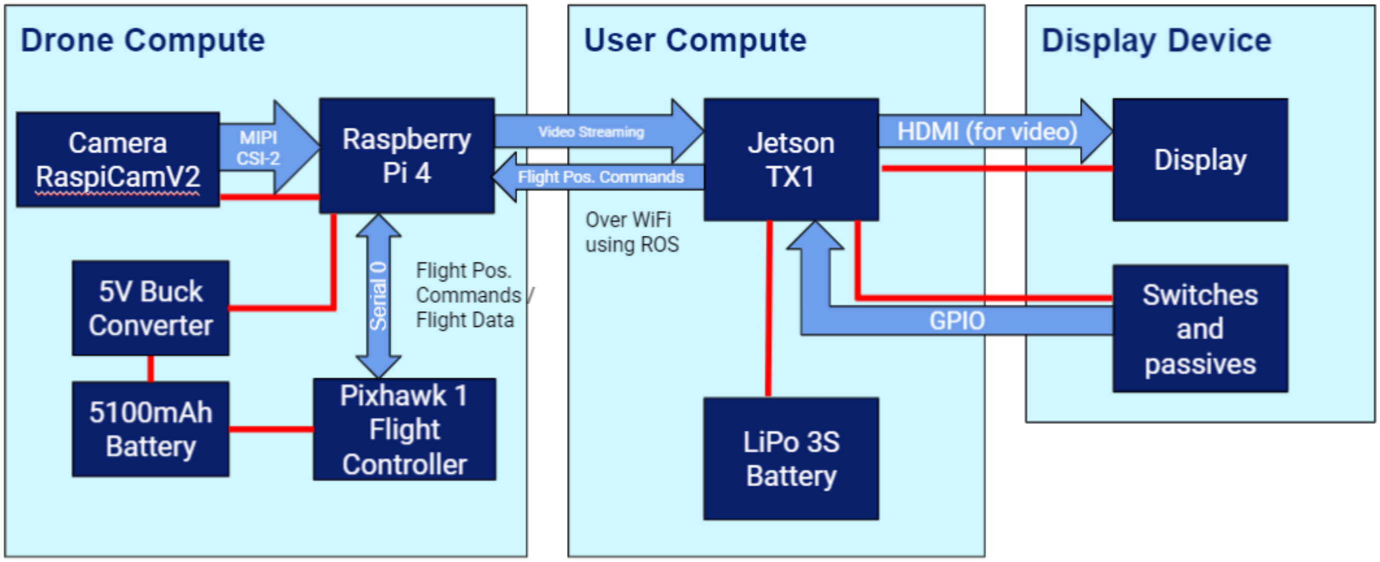


Figure 1: System Block Diagram

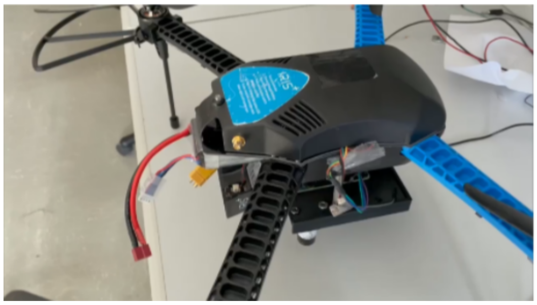


Figure 2: Drone Compute

3.1 RPi

Used as a middleman between the drone flight controller, camera and the TX1. Forwards updates in position from the flight controller to the TX1, new image frames from the camera to the TX1 and updated motion plans from the TX1 back to the drone flight controller.

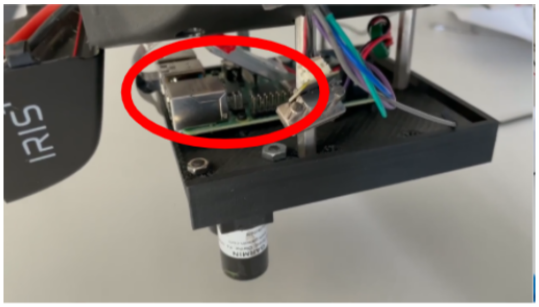


Figure 3: Raspberry Pi Location

3.2 Iris Drone Flight Controller

Takes high-level position/velocity/orientation motion planning waypoints that have been computed by the TX1 and forwarded through the RPi. Uses these to determine the speed of the motors required to achieve the desired path.

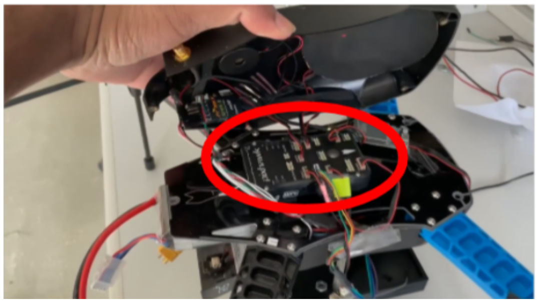


Figure 4: Flight Controller Location

3.3 Jetson TX1

Used for video processing and processing user inputs via the switches. The TX1 use OpenCV library to process the captured video from the camera for the computer vision algorithm, and then feed the output of this into state estimation and motion planning algorithms. The implementation of the algorithms run on the TX1 will be discussed in detail in the System Description section.

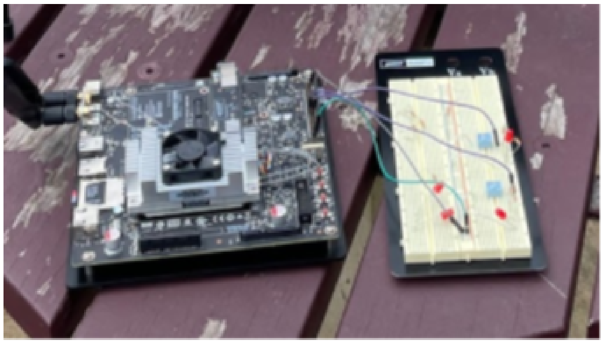


Figure 5: Jetson TX1 and Switches

4 SYSTEM DESCRIPTION

Now that we have gone over the high-level design of our solution, we will go through the specifics on how each of the components work and interact:

4.1 Onboard Compute

The drone itself has three broad categories of compute: flight controller, sensors, and RPi. The flight controller is a Pixhawk that uses the PX4 API with ROS to broadcast state of the drone and listen to motion commands. The Pixhawk can take a variety of control commands, from high level position (x, y, z) in local or global space to low level desired orientation in roll, pitch, and yaw. The Pixhawk has built-in PD controllers that control speed of the motors, and we only need to provide high level motion commands.

The Pixhawk already is built with several sensors for drone odometry: gyroscope, accelerometer, magnetometer (compass), and barometer. It also contains a GPS that provides global localization within 1-2 meters. We also have a downward-facing Lidar Lite V3 rangefinder that provides accurate height estimate as well as an optical flow PX4Flow camera that points downwards. All of these sensors are fused together within the Pixhawk's internal Extended Kalman Filter.

Connected to the Pixhawk via UART, the RPi listens to these drone state estimates and forwards it down to the ground TX1. The RPi also records video from a RaspiCam V2 and stream down the frames to the TX1 via ROS WiFi. The TX1 simultaneously publishes motion commands to the RPi, which forwards these to the Pixhawk flight controller.

4.2 On-User Compute

The on-user compute is performed entirely on the Jetson TX1, which interfaces with the wearable display using HDMI and the switches via GPIO. The control flow for the TX1's operation is shown in Fig. 7 where light blue boxes are switch inputs and red boxes are steps we only execute in simulation (more on this in section 4.4). Here is a labelling of the switches we use, along with a description of the steps from Fig. 7:

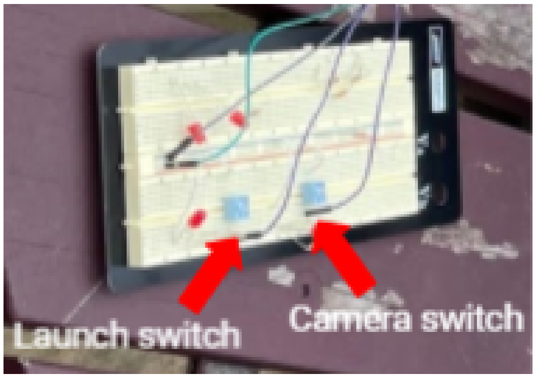


Figure 6: Purpose of Switches

1. When the launch switch is turned on, the TX1 initializes ROS and sends a "launch" event to the RPi via ROS signalling it to launch communications with the flight controller. The RPi then tells he flight controller to take off.

2. When the camera switch is turned on, the TX1 sends a "cam_start" event to the RPi. The RPi starts streaming image frames to the TX1, they go through our image processing stack, and the resulting motion plans are sent back to the flight controller.

3. When the camera switch is turned off, the TX1 sends a "cam_stop" event to the RPi. The RPi closes the camera and tells the flight controller to land.

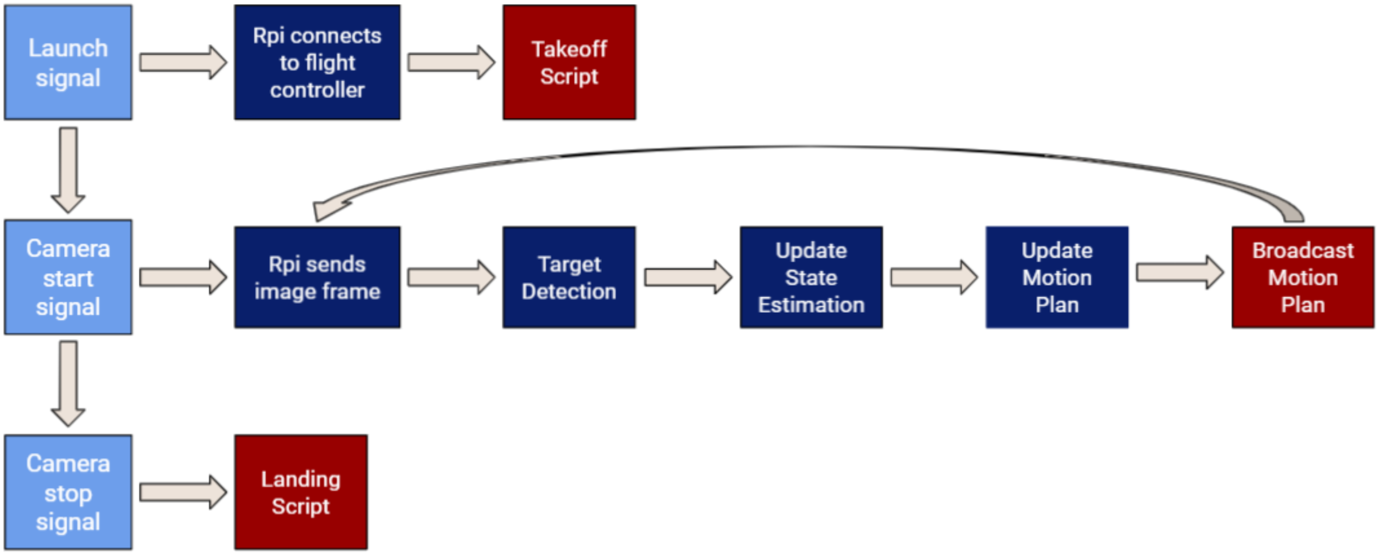


Figure 7: Software Control Flow

In the next subsections, we will elaborate further on the computer vision and state estimation.

4.2.1 Target Detection

The computer vision algorithm to detect the target reads frames captured by the camera, extracts the pixel location of the target, and projects these into 3D world space. To accomplish this, the captured frames go through the following stages:

1. **RGB Color Thresholding**

   The image read from the camera is an RGB image. In our scope, we mentioned we are tracking a target wearing a red colored shirt. Thus, the first step is to perform color thresholding on the RGB image to narrow down the red channel of the image. We predetermine the ranges for each of the 3 channels, blue(B), green(G), red(R) by manually adjusting them such that only the red pixels of the shirt would pass through. At first, we tried using HSV color space, but during testing, we realized it was too dependent on the lighting conditions. However, the RGB filter worked well in bright and dull lighting conditions.

2. **Noise Filter**

   The color thresholding by itself still allows some noisy pixels to pass through that are not the color of the shirt, which leads to the algorithm not detecting the target. To remove those, the frame is eroded using OpenCV which shrinks each group of pixels. This removes the noisy pixels that were getting past the color thresholding and affecting the detection of the target.

3. **Object Detection**

   Once the noise is removed, the OpenCV findContours function is used to detect the red shirt. Then, the moments function is used to find the center of the object detected. This gives us the detected target position in pixel coordinates.

4. **2D to 3D Conversion** The 2D pixel representing the center of the target needs to be converted into 3D world position since this enables the drone to plan its own motion in 3D. We apply a classical technical known as back-projection, which has several key steps. First, a 2D pixel can be mapped to 3D space with respect to the camera light sensor using the camera's intrinsic matrix:

   $$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$
   $$\mathrm{X}_{sensor} = K^{-1}[p_x, p_y, 1]$$

   where (p_x, p_y) is the 2D pixel and K can be generated using chessboard image calibration.

   Second, this 3D sensor position needs to be transformed into 3D world position in the following order:

   - Image → Camera
   - Camera → Drone
   - Drone → World

   Each of the steps above can be represented as an affine transformation composed of rotation **R** and translation **T**:

   $$\mathbf{M_{3x4}} = \mathbf{R_{3x3}}[\mathbf{I_{3x3}}|\mathbf{T_{3x1}}]$$
   $$\vec{X}_i = \mathbf{M_{3x4}}[\vec{X}_j \;\; 1]$$

   where $\vec{X}_i$ is the newly transformed 3D position. Steps 1 and 2 have static transforms that we defined using Solidworks and a custom-designed CAD model:

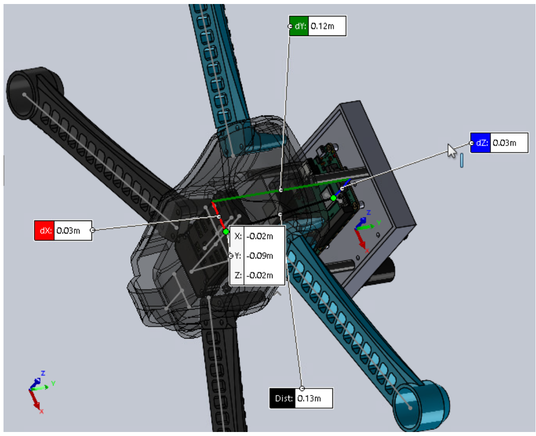
   

   Figure 8: Custom Solidworks Model with distance measuring

   The last step of transformation is dynamic, and is computed using the drone's own position and orientation estimate, which is calculated by the drone's flight controller.

   Now that we have the estimated 3D position of the pixel, we can calculate a vector from the drone's camera center to this pixel:

   $$\vec{r} = X_{pixel} - X_{camera}$$

   where X_{pixel} is our final 3D pixel coordinate in world space and X_{camera} is the position of the camera in world space.

   Using this vector, we can project this vector onto the ground plane, and the point of intersection provides us the 3D estimate of the target's position.

   We make a key assumption: the target is standing on flat ground represented by a plane $z = 0$. To avoid making this assumption, we could use deep learning techniques to predict the normal vector of surfaces, but our assumptions works well for our test environment. Overall, our predicted 3D target position will serve as an input to the state estimator.

4.2.2 State Estimation

State estimation is used to keep track of a model of the target's state at all time, enabling us to predict future motion. We model target state in terms of the following variables:

- (x, y, z) positions
- (x, y, z) velocities
- (x, y, z) accelerations
- estimated covariance between these parameters

Notice how given these positions, velocities and accelerations, we can already predict future motion using kinematics equations. Of course our model may not necessarily remain accurate to the target's actual state and we need to constantly update it using the results of our target detection. We use an algorithm called Kalman Filtering in order to do this. Observe Fig. 8:

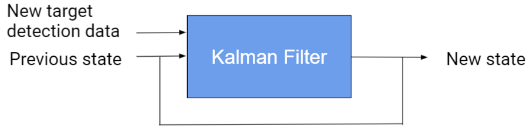


Figure 8: Kalman Filtering

Our new state is a probabilistic "best guess" based on the previous state and the newly detected 3D position of the target. Ideally, the two inputs to the filter agree. However, when they disagree, it's up to the filter to balance between the two, smooth out any noise in the detected positions, and come up with a model that most closely matches the target's actual motion.

The inner workings of the filter consist of the following state transition matrices:

- **F**: Models how the user's state deterministically transitions from one period to the next. It assumes the user does not change acceleration and takes in Δt as a parameter.
- **Q**: Models random fluctuations in user state (due to user changing acceleration). Takes in assumed std. dev. in user accceleration as a parameter.
- **H**: Models how target detection results relate to user's current state, Ideally, the detected (x, y, z) position of the target matches the (x, y, z) position modeled in our current state.
- **R**: Models random fluctuations in target detection data (noisy observations). Takes in assumed std. dev. in observed user position as a parameter.

Using these matrices, the Kalman Filter probabilistically determines the most likely future state ($\hat{x}'_k$) and covariance in future state (P'_k) given the user's current state ($\hat{x}'_{k-1}$), current covariance in state (P'_{k-1}) and current (x, y, z) position outputted by target detection ($\vec{z}_k$). Here are the equations for doing so [4]:

$$\begin{gathered}\hat{x}_k = \mathbf{F}\hat{x}'_{k-1}\\ P_k = \mathbf{F}P'_{k-1}\mathbf{F}^T + \mathbf{Q}\\ \mathbf{K'} = \mathbf{H}P_k\mathbf{H}^T(\mathbf{H}P_k\mathbf{H}^T + \mathbf{R})^{-1}\\ \hat{x}'_k = \hat{x}_k + \mathbf{K'}(\vec{z}_k - \mathbf{H}\hat{x}_k)\\ P'_k = P_k - \mathbf{K'}\mathbf{H}P_k\end{gathered}$$

4.2.3 Motion Planning

Motion Planning is composed of three key steps:

1. **Drone Dynamics Model** We first need to define a model of the drone's state and how control inputs affect this state. We represent the drone's state as simple 2D position $[x, y]$ with an assumed fixed height of 8m. We define control input as horizontal velocity :

$$\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}v_x\\v_y\end{bmatrix}dt$$

   This means that we cannot rotate the drone to face a target, but must move horizontally. While these assumptions indeed limit our control of the drone, this behavior fits our use-case. These simple dynamics also allow our computation to meet the required rate specified by the drone's flight controller.

2. **Define Desired Behavior** Next, we need to define some behavior for the drone. We firstly want the drone to keep the camera focused on the target, meaning the target should stay in the center of the image. This objective can be represented as a maximization of the dot-product between two vectors:

$$\max_{\mathbf{v}_1} C_1 = (\mathbf{v}_1 \cdot \mathbf{v}_2)$$

   where $\mathbf{v}_1$ is the fixed vector from the camera center to the center pixel on the image plane and $\mathbf{v}_2$ is the dynamic vector from the drone's camera to the 3D target position, obtained from the target detection pipeline.

   Second, we optimize for smooth, stable video footage, and this can be achieved by minimizing the motion of the drone represented in terms of velocity control input:

$$\min_{\mathbf{v}} ||\mathbf{v}||_2^2$$

   where $\mathbf{v} = [v_x, v_y]$ is the velocity control input. We optimize these two objectives over a time horizon of 1 second with a stepsize of $dt = 0.1$ seconds. We use the predicted target trajectory inside this optimization. We have tuned weights for the above two costs to achieve desired behavior.

3. **Solve for Behavior** Now that we have defined our formulation, we can solve this using an optimization package. In our case, we use the Scipy Optimization package. We use a Model Predictive Control formulation to optimize over multiple timesteps, but only execute the first step of this plan. Every iteration, we generate an entire trajectory, but execute only the first step. This ensures our motion is not only optimal, but is reactive to position drift and external disturbances like wind due to the feedback loop planning.

4.3 Switches and Passives

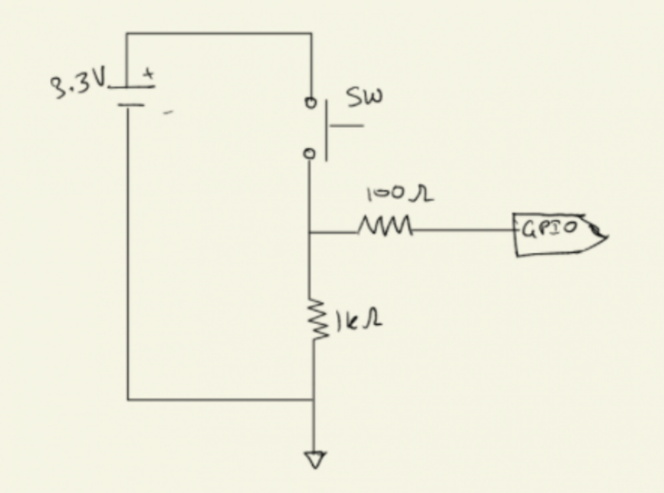


Figure 9: Switch Circuitry

As mentioned above, switches are used to interact with the drone. They are assembled on a breadboard and powered by 3.3V GPIO (pin 1 on the J21 header on the TX1). Fig. 9 shows a simple circuit to connect a switch to the TX1. The switches are connected to J21 header pins 15 and 18 on the TX1 which all have a pull down network that completes the circuit when the switches are pressed. We are able detect changes in the switch state through interrupt-based I/O, as the TX1 receives an interrupt on both the rising/falling edges. This is less wasteful to the CPU than polling-based I/O since user inputs are infrequent.

4.4 Simulation-Only Processes

From Figure 7, you may notice that some boxes are highlighted in red, which we briefly mentioned were steps conducted in simulation only. This is because the more we tested our drone's flight controller, we noticed errors in its internal local position estimates:

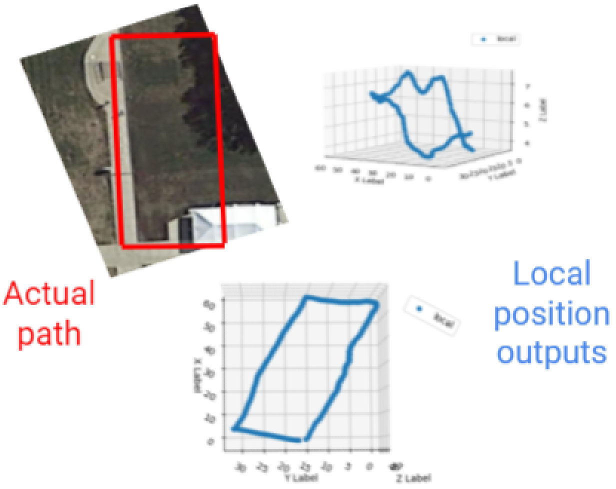


Figure 9: Errors in Local Position

Unfortunately, these errors did not seem to be systematic and differed from test to test seemingly at random. Because of these local position errors, the drone acted chaotically when we forwarded flight position commands to the flight controller. For example, sending a signal to hold position would cause it to circle wildly rather than hold position (possibly due to drift in position estimates).

These errors are most likely due to our flight controller being an older model. Our model only contains 1 MB flash memory, which is technically not enough to contain the entire firmware. Thus, when building and flashing the firmware, we were forced to experiment with removing certain "non-essential" sensor drivers, possibly affecting the positional estimates.

For these reasons, we determined that while we would demonstrate the calculation of motion plans in real-time, we would not be sending these motion plans back to the drone to follow due to safety concerns. Instead, full autonomy would be tested on a Software in the Loop simulated drone with the same interface as the actual drone. The same code that is used to forward commands to our actual drone can be used to forward commands to this simulated drone.

5 DESIGN TRADE STUDIES

5.1 On-Drone vs. On-User Compute

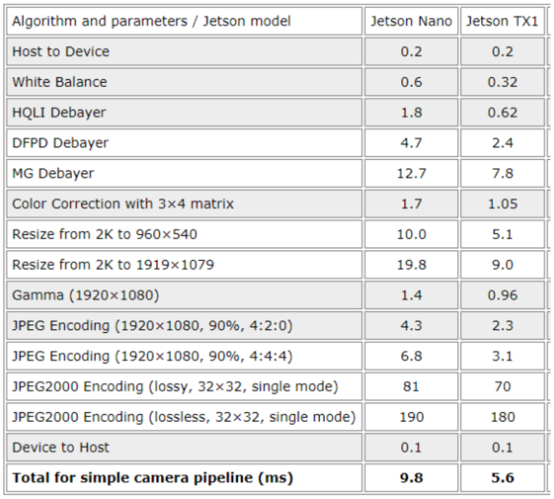
Originally, rather than have a Jetson TX1 located on the user and a RPi on the drone to interface with peripherals, we intended to just have a single Jetson Nano located on the drone that performed all the necessary software tasks. We figured that any extra compute on the user would be cumbersome and hoped to perform as much computation as possible on the drone itself. However, looking into the issue further, we realized that for our use-cases, most users would be carrying a backpack anyways and that performing computation on the drone severely limited us. The main points are:

1. **Power Consumption**

   Powering the Jetson Nano using the same battery as the drone would severely eat into the drones runtime given that the battery is only 5100 mAH. Considering that the drone by itself only runs for about 20-30 minutes on a single charge this would put us in danger of missing our flight time requirement. Not to mention that the additional weight of a Jetson Nano would force the drone to use up even more power to stay in flight. We would be forced to invest in a bigger battery, which once again would weigh the drone down even further.

2. **Performance**

   Moving compute to the drone forces us to use a Jetson Nano rather than a TX1 (for size/weight reasons). However, a Nano contains only 128 GPU cores while the TX1 contains 256. For basic color filtering / blob detection this may be okay, but since we wanted to meet our FPS requirement and have the flexibility of using more sophisticated approaches should we need to, the TX1 was a much safer approach. Here is a table [2] which benchmarks the Nano vs. TX1 for the same image processing task:



Algorithm and parameters / Jetson model	Jetson Nano	Jetson TX1
Host to Device	0.2	0.2
White Balance	0.6	0.32
HQLI Debayer	1.8	0.62
DFPD Debayer	4.7	2.4
MG Debayer	12.7	7.8
Color Correction with 3×4 matrix	1.7	1.05
Resize from 2K to 960×540	10.0	5.1
Resize from 2K to 1919×1079	19.8	9.0
Gamma (1920×1080)	1.4	0.96
JPEG Encoding (1920×1080, 90%, 4:2:0)	4.3	2.3
JPEG Encoding (1920×1080, 90%, 4:4:4)	6.8	3.1
JPEG2000 Encoding (lossy, 32×32, single mode)	81	70
JPEG2000 Encoding (lossless, 32×32, single mode)	190	180
Device to Host	0.1	0.1
Total for simple camera pipeline (ms)	**9.8**	**5.6**

Figure 10: Benchmarks for Nano vs. TX1

Of course, there also some benefits to handling compute on the drone rather than the user. Streaming the video down from the drone to the user over WiFi creates additional latency. Because of this, once the Jetson receives a frame, the user may have already moved slightly. However, given that we are able to compute new motion plans around 3 times per second (as shown in the following Testing section), we felt this would not be significant.

5.2 Image Compression

Our image capture pipeline is composed of two steps: image capture and image conversion into a ROS format that can be streamed. Since this includes conversion, we decided to compare image capture throughput between compressed and uncompressed image formats. While image compression requires extra computation, time can be saved from image conversion. We noticed the following results, computed over a video of two minutes, or 360 frames:

	Image Capture FPS
Uncompressed	2.91
Compressed	2.94

Table 1: Image Capture FPS Comparison

Looking above, notice how image compression actually **increases** the throughput of our image capture, and this is likely due to greater time savings in the image conversion step. Though not mentioned, image compression also increases image streaming FPS, but since image capture is the bottleneck of our system (see Testing for more details), we analyze this specifically.

However, even though image compression slightly increases FPS, we deemed the increase from 2.91 to 2.94 to be negligible when compared to the loss in quality from compression.

5.3 Motion Planning Control Cost

For our next study, we examined the effect of motion planning cost function weights on behavior of the drone. Our motion planning is a balance between optimal target tracking (keeping target in the center of image) and video smoothness. The latter is achieved using a control cost that discourages excessive drone motion, we compare tracking and stability performance while varying the weight of this control cost.

	Ours (1x)	High (10x)	Low(0.1x)
Tracking	97%	47%	84.58%
Stability	93.75%	100%	43.75%

Table 2: Comparing original, high, and low control cost weights in stability and tracking performance

The above results were computed in simulation. Notice above how a high control cost leads to perfect stability of 100%, but poor tracking of 47%. Low control cost has the opposite effect. Also notice how a low control cost does not achieve 100% tracking as we might expect. Due to the low control cost, the drone often changes position violently, causing sharp changes in roll and pitch that cause the camera to lose sight of the target for a few frames. Overall, we can see that optimal performance is achieved by balancing control cost and viewpoint cost.

6 TESTING AND VALIDATION

6.1 Target Detection

To test target detection, we manually flew the drone and had a target wearing a red shirt walk in a variety of different motions. We flew the drone in a variety of angles to capture frames where the target was large in the frame, small in the frame, centered to the frame, and on the edges of the frame. Across a sample size of 157 frames, we calculated the following results:

	FP Rate	FN Rate	Avg. Pixel Err.
Actual	0%	14.78%	11.87
Desired	2%	10%	16.75

Table 3: Target Detection Results

We had no false positives and our average pixel error was well within what we specified. Our false negative rate was slightly higher than desired, but this is because we optimized our target detection to provide no false positives. As mentioned in the requirements section, lowering false positives is critical because a couple outlier points could permanently throw off our Kalman Filter and lead to bogus state estimation. However, false negatives are less of an issue because in the case we miss a frame, we can just fall back on our currently modeled state of the target from the state estimator.

6.2 Tracking and Stability

To test drone stability and tracking, we ran our code in simulation in order to track a simulated target. The simulated target followed one of three behaviors:

1. **Walking-** Slower movement and limited turning, probabilistically transitions between stopped and walking

2. **Running-** 2x speed movement and more frequent turning, probabilistically transitions between running and walking

3. **Mixed** Probabilistic transitions between stopped, walking and running states

Here are the results:

	Tracking	Stability
Walking	100%	100%
Running	88%	100%
Mixed	97%	93.75%
Desired	90%	90%

Table 4: Drone Tracking and Stability Results

For the most part, we easily met our desired metrics across these conditions. Intuitively, it makes sense that the slowest target (purely walking) would be easiest to track while the fastest target (purely running) would be harder to track since it's harder to keep up with.

Stability wise, it makes sense that mixed would perform worse than walking and running. In an environment without wind, what stability really measures is the smoothness of our motion plans. Smoother plans achieve higher stability whereas ones involving drastic changes in velocity that jerk the drone around achieve lower stability. Since a mixed target is constantly shifting movement speeds, the motion plans are less smooth since they keep adjusting to the changes in speed. Thus, stability is lower.

As mentioned in the previous section, we also did a trade-off study by comparing motion planners with higher and lower control costs. This was done for the mixed movement condition. Here are the results once again:

	Ours (1x)	High (10x)	Low(0.1x)
Tracking	97%	47%	84.58%
Stability	93.75%	100%	43.75%

Table 5: Comparing original, high, and low control cost weights in stability and tracking performance

Again, it makes sense that penalizing movement speed causes tracking inaccuracies (can't keep up) but increases stability (slower movement = smoother movement). However, allowing the drone to move as fast as it wants causes awful stability from the jerky motion and actually gives medicore tracking. As mentioned before, this is because when the drone moves fast, it rolls/pitches, causing the camera to lose the target. Our control cost optimized for both tracking and stability.

6.3 Communication Bandwidth

To measure this, we found the average latency/FPS of every component of our image processing pipeline. We then took the FPS of the bottleneck as the overall FPS of the system:

Operation	**Latency (s)**	**FPS**
Image Capture	0.344	2.91
Image Streaming	0.25	4
Target Detection	3.38e-4	2,959
State Estimation	6.50e-4	1,538
Motion Planning	0.0135	74.07
Overall	**0.344**	**2.91**
Desired		5
Minimum Req.		2

Table 6: Latency Across Image Processing

We met the bare minimum requirement of 2 FPS in order for the flight controller to be operational. However, we still fell short of our desired 5 FPS. The main bottleneck was the time it took to capture a single frame rather than any of our code. This time is dependent on the camera itself. Since at this stage we were already on our third camera due to the previous ones malfunctioning, we decided to stick to this one since it still met the minimum requirement.

One idea we had to increase FPS was compression. We tried this before we realized that the image capture was the bottleneck. Using compression, we get:

	Image Capture FPS
Uncompressed	2.91
Compressed	2.94

	Image Streaming FPS
Uncompressed	4
Compressed	6.67

Table 7: Compression FPS Comparison

Image streaming is greatly improved, but this is irrelevant since it isn't the bottleneck. Counter-intuitively, image capture is also improved. This is because the image capture phase also includes a conversion of the image to a ROS image format. This is able to be done faster on a compressed image. That being said, the benefits are still negligible and not worth the loss of image quality.

6.4 Flight Time

Across multiple tests our drone was able to operate for around 12 minutes under a single battery charge.

7 PROJECT MANAGEMENT

7.1 Schedule

Our schedule (Fig. 11 in the Appendix) was composed of four phases: **design**, **pre-integration**, **integration**, and **performance testing**. In design, we proposed various methods for building the system with various contingency plans in case certain ideas fail. We researched and purchased materials that would suite our requirements, and verified that already-owned materials would meet our requirements. In pre-integration, we implemented and debugged various subsystems independently of one another. For instance, target detection was implemented and tested using a fixed camera without involving the drone. Other tasks included target state estimation and prediction, drone motion planning in simulation, and the hardware communication protocol for the wearable display. After this phase integration involved combining all these independent subsystems together to produce a functional product. After this, performance testing involved us testing the system as a whole and recording performance through our various test metrics.

7.2 Team Member Responsibilities

Vedant Parekh worked on target detection in an image as well as the hardware communication protocols and circuitry for the wearable device.

Siddesh Nageswaran worked on target state estimation and prediction as well as the CAD design for mounting all the compute and sensors onto the drone.

Alvin Shek worked on inferring 3D motion of the target from its 2D motion and the 2D image as well as generating drone trajectories from this information.

7.3 Risk Management

Here are the contingency plans we came up with for certain failure cases:

1. **Noisy Target Detection** During actual flight, target detection could have been noisy and unreliable if the image quality was too poor or the drone was too unstable. While drone acceleration and jerk are be minimized in trajectory optimization, if detection still wasn't effective, we planned to buy a shirt with higher contrast. In addition, we thought we could switch to other detection methods: April Tags [9] or CNN-based models that can be trained with domain randomization to be robust. However, we didn't run into any issues with target detection.

2. **Noisy Target State Prediction** Target state prediction becomes noisy if the target detection is too noisy. In this case, we could bias more towards the historical model of target motion rather than frame by frame data if the incoming observations are too noisy. We can even set a threshold to throw away observations if they seem unrealistic. If the target suddenly ”jumps” a large distance in pixel space that would translate to unrealistic motion in the real world, we can ignore this observation. We ended up employing all these techniques

3. **Lost Sight of Target** The drone continues to use the predicted trajectory of the target for a short amount of time, even if the target becomes occluded by some trees or obstacles. If the target is unseen for a prolonged duration, the drone immediately flies back to home or land, and the user receives a warning message on the display.

4. **Wifi Issues** As mentioned in the scope, we assume a strong WiFi signal in the environment (on-campus). If the WiFi connection ends up unstable during flight and motion commands do not stream at the minimum 2Hz, the drone automatically lands.

5. **Unable to Safely Fly Drone** In the case that we were not able to safely fly the drone autonomously, we planned to do as much as we can in real-time (calculate all the motion plans without sending them back to the drone), and then demonstrate the rest of the autonomy in simulation. We ended up having to use this approach.

7.4 Budget

The bill of materials is shown in Fig. 12 of the Appendix. Any components that we ordered through our $600 available budget are highlighted in yellow with their prices stated. For any components that were reused, we still stated their market price (although this does not count against our budget).

8 ETHICS

Some of the ethical issues involved in our project include when the drone is in the air tracking and recording, the surrounding people in the frame may not consent to being in the video. This may either be done maliciously or unintentionally (inadvertently filming someone else in the background while filming yourself). The way we can try and mitigate this is to use facial recognition to recognize the user and blur out all other faces. If someone else wants to be included, the user can have control to unblur their face upon which the person who wants to be included needs to make eye contact with the camera and shake his/her head up and and down (to signify a yes). The same way we can use facial recognition to ensure the user does not spy on other people by making sure the user is always in the frame.

Some other issues with our project may be running into things like transmission lines or crashing into other things like traffic signals, etc. We need to make sure our autonomy is safe and does not interfere with the outside environment. One way to mitigate this is to incorporate object avoidance algorithms to make sure we avoid these objects while flying.

Another ethical issue could be that the product could be used by the government to keep tabs of people outside of legal jurisdiction. This product is not intended for this purpose so we would not be selling it to the government, but again, we could add an extra precaution by ensuring the user is in the frame at all times, making it very hard for the government or anybody else to spy on people.

An additional issue could be two people wearing the same colored shirt (since we track a specific color), so the drone could get confused as to who to follow and record the wrong person who may not want to be recorded. Again, the way we could solve this is before going into flight, capture the face of the user and then use facial recognition while in flight to ensure that the right person is being tracked.

None of these edge cases adversely affect a specific group of people, but rather anyone could be a victim of these issues if they are not carefully addressed and mitigated.

9 RELATED WORK

[8] provided us inspiration for our motion planning by introducing the idea of maximizing the dot-product between the camera's center view vector and the camera to 3D target vector. Their drone dynamics model is nonlinear along with their objective function, which is why they use a proprietary optimization solver known as FORCES PRO. We initally tried using this solver with a more complex 7D system for the drone, but found poor performance and switched to our current simple 2D system.

The open-source Instructables project, [7] provided us a useful reference for the necessary steps to take to connect our Raspberry Pi to the drone's flight controller. This project only focuses on this communication aspect and demonstrates successful offboard control, but does not implement any autonomy.

[10] describes a drone that hones in on red balloons and pops them like targets. While the general idea of detecting a red-colored target is similar to our project, they do not perform any 2D to 3D state estimation or solve for optimal viewpoints. We do not want our drone to simply chase after the target's own position, but rather want the drone to only keep the target in the center of the video.

We drew inspiration from CMU's own Computer Vision course, 16385, to implement 2D to 3D position estimation. Specifically, we referenced lecture 11 of the Spring 2017 course [6], which describes the image backprojection process that we use. We applied this theoretical algorithm to our physical drone using Solidworks to estiamte the fixed transform from camera to drone as well as the dynamic transform that depends on drone pose.

Lastly, our original plan was to actually use Deep Learning to estimate the ground plane's normal vector. [12] specifically uses a CNN to aggregate pixels in a 2D image into surfaces, each with a normal vector. This is similar to common depth estimation networks that predict disparity maps from 2D image, but instead predicts plane normals using an RGBD dataset. Rather than use this, however, we realized in our test environment that the ground is essentially flat and can be assumed to have a normal vector pointing straight up to the sky.

10 SUMMARY

10.1 Lessons Learned

The first big lesson we learned was to really think about all that could go wrong with our project: to understand the worst case and see given the resources if we could avoid the worst case. For us, we never thought through the worst case, which in hindsight is something we really should have thought of while choosing this project: the Pittsburgh weather. To test our drone, we had to fly it. Unfortunately since the budget was low, we could not afford a high quality drone. Thus, our drone was extremely sensitive to wind conditions. Additionally, our drone was not waterproof, so the days it rained (and it rains a lot in Pittsburgh), we could not test our drone. Further, our drone required GPS for navigation, which inhibited us from testing the drone in the few large indoor places.Therefore, the only way we could test our system was on sunny days with low wind speed, which were only a handful of days during the whole semester. The way we combatted that was to try to do as much as we could in simulation (testing our object detection, state estimator, and motion planning in simulation). However, simulation is an ideal world and it usually never translates in the real world, so we missed out on the feedback we could have gotten by testing our drone in the real world multiple times. The times we did test it, we learned so much of what was going wrong that if we had more chances to do those tests, it would have been really helpful.

Another lesson we learned was that in the final 3 weeks, we realized a key limitation with our older flight controller model. Given a limited flash memory size of 1MB, we could only flash the default firmware binaries onto the flight controller. These default binaries do not include the firmware for two of our sensors, an optical flow camera and a single point Lidar for height estimation. Normally, this can be resolved by simply specifying the desired firmware packages in an XML-type file and building custom firmware. However, given our 1MB flash limit, we could not include our two sensors, and as a result, our drone only had onboard IMU and GPS for pose estimation, leading to the large noise in position estimation. We most likely would have been able to fly the drone if we had these two additional sensors since pose estimation error would drastically reduce. Overall, we learned that remaining too stubborn to buy new parts can waste time and even lead to an incomplete project as in our case.

We learned another valuable lesson while implementing our motion planning. We initially designed a more complicated planner that had to optimize for full 3D position and yaw angle. While this allows for more fine-tuned control of the drone's motion, this also significantly increases runtime in the planning computation and optimization. Solving over a horizon of 10 timesteps took upwards of 12 seconds, much too slow for our needs. We eventually stuck with our final, simplified 2D state model in x and y only. While this prevents us from controlling the drone's height and orientation, this resulted in much faster motion planning and overall faster reaction times to the target's motions. Overall, these simplifying assumptions also still work well for our use-case, and in fact seem to lead to more stable video naturally. The valuable lesson here is to always start simple and only add complexity where necessary. Research papers often criticize every assumption and seek the most general solution, but it's important to recognize that assumptions aren't evil and may even be necessary. The key is to determine which assumptions are fine for a specific application.

10.2 Future Work

For future work, we would like to first and foremost buy a new flight controller, enable our new sensors, and run our autonomy on the physical drone. Second, we would like to explore other interesting flight modes for the drone. Currently, the drone only optimizes to keep the target in the center of the video. This doesn't allow for much diversity in footage when compared to the possible panoramic sweeps that professional videographers perform. A cool extension would be to let users wave their arms in a sweeping motion to direct the motion of the drone, so the drone not only keeps the target in its camera's view, but also follows a specific path that can view the target from different angles.

Glossary

API Application Programming Interface. 4

CAD Computer-Aided Design. 5

CNN Convolutional Neural Network. 11

CPU Central Processing Unit. 7

FN False Negatives. 1, 9

FP False Positives. 1, 9

FPS Frames per Second. 2, 8–10

GHz Gigahertz. 2

GPIO General Purpose Input/Output. 2, 4, 7

GPS Global Positioning System. 4

GPU Graphics Processing Unit. 8

HDMI High Definition Multimedia Interface. 4

HSV Hue, Saturation, Value (image representation). 5

I/O Input/Output. 7

J21 An array of GPIO pins on the Jetson. 7

kV Kilovolts. 2

LiPo Lithium Polymer. 2

mAH Milliamp-hours. 2, 8

MIPI Mobile Industry Processor Interface. 2

PD Proportional, Derivative (feedback control). 4

PX4 Type of flight controller firmware. 4

RGB Red, Green, Blue (image representation). 5

RGBD Red, Green, Blue, Depth (image representation). 11

ROS Robot Operating System. 2, 4, 8, 10

RPi Raspberry Pi. 2–4, 8

TX1 Nvidia Jetson TX1. 2–4, 7, 8

UART Universal Asynchronous Receiver-Transmitter. 4

Wkg Watt-kilograms. 2

References

[1] *12 Best Follow Me Drones And Follow You Technology Reviewed.* 2020. URL: `https://www.dronezon.com/drone-reviews/best-follow-me-gps-mode-drone-technology-reviewed/`.

[2] *Benchmarks for Jetson Nano, TX1, TX2 and AGX Xavier.* 2019. URL: `https://fastcompression.medium.com/benchmarks-for-jetson-nano-tx1-tx2-and-agx-xavier-6c3b7105421d`.

[3] *Drone Flight Time Calculator.* 2018. URL: `https://www.omnicalculator.com/other/drone-flight-time`.

[4] *How a Kalman Filter Works, in pictures.* 2015. URL: `https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/`.

[5] *Iris - The Ready to Fly UAV Quadcopter.* URL: `http://www.arducopter.co.uk/iris-quadcopter-uav.html`.

[6] Kris Kitani. URL: `http://www.cs.cmu.edu/~16385/s17/`.

[7] Artyom Maxim. *Control a Pixhawk Drone Using ROS and Grasshopper.* URL: `https://www.instructables.com/Control-a-Pixhawk-Drone-Using-ROS-and-Grasshopper/`.

[8] Tobias Nägeli et al. "Real-Time Motion Planning for Aerial Videography With Dynamic Obstacle Avoidance and Viewpoint Optimization". In: *IEEE Robotics and Automation Letters* 2.3 (2017), pp. 1696–1703. DOI: `10.1109/LRA.2017.2665693`.

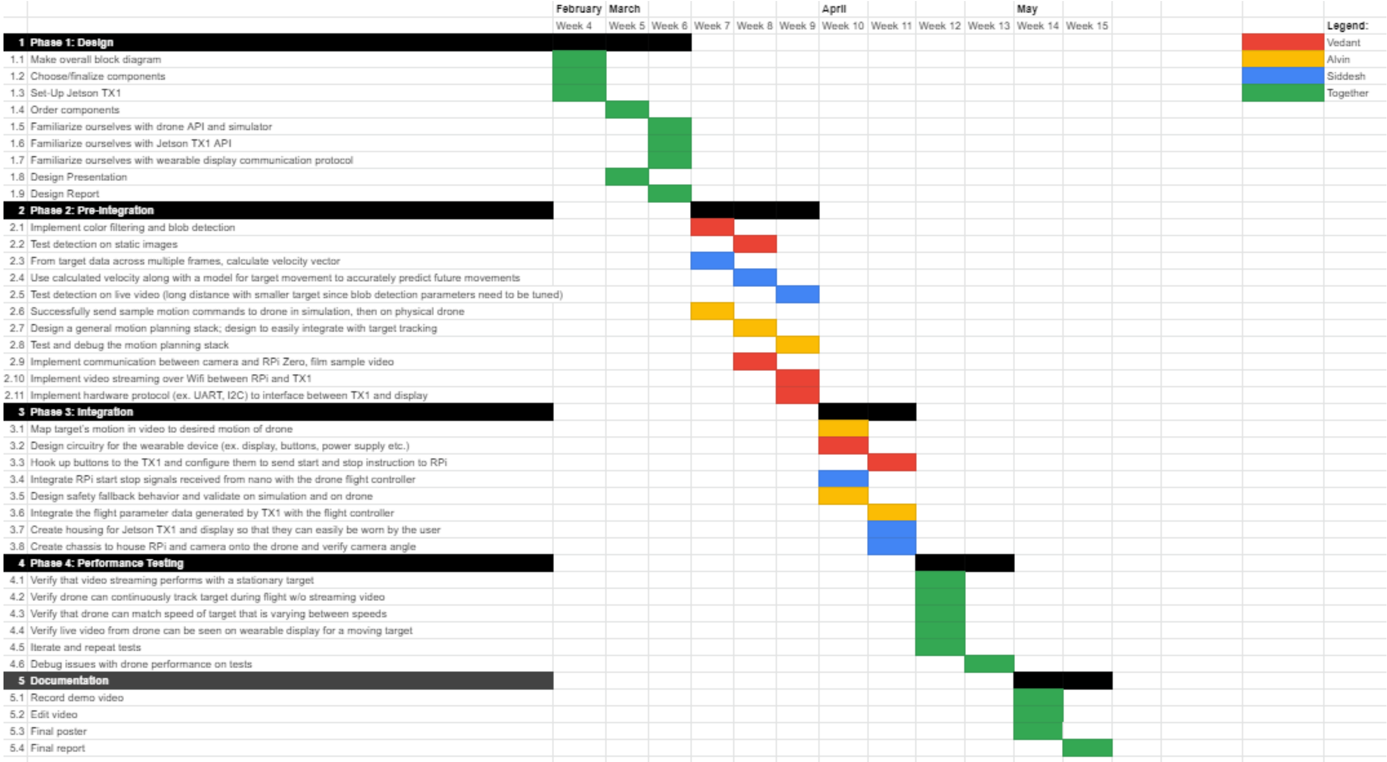
[9] E. Olson. "AprilTag: A robust and flexible visual fiducial system". In: *2011 IEEE International Conference on Robotics and Automation.* 2011, pp. 3400–3407. DOI: `10.1109/ICRA.2011.5979561`.

[10] Randy. *Red Balloon Finder.* URL: `https://diydrones.com/profiles/blogs/red-balloon-finder`.

[11] *USB Stick - WiFi Realtek 8188 (rtl8188cus).* 2015. URL: `http://domoticx.com/usb-stick-wifi-realtek-8188-rtl8188cus/`.

[12] Fengting Yang and Zihan Zhou. "Recovering 3D Planes from a Single Image viaConvolutional Neural Networks". In: *IEEE Robotics and Automation Letters* (2018).

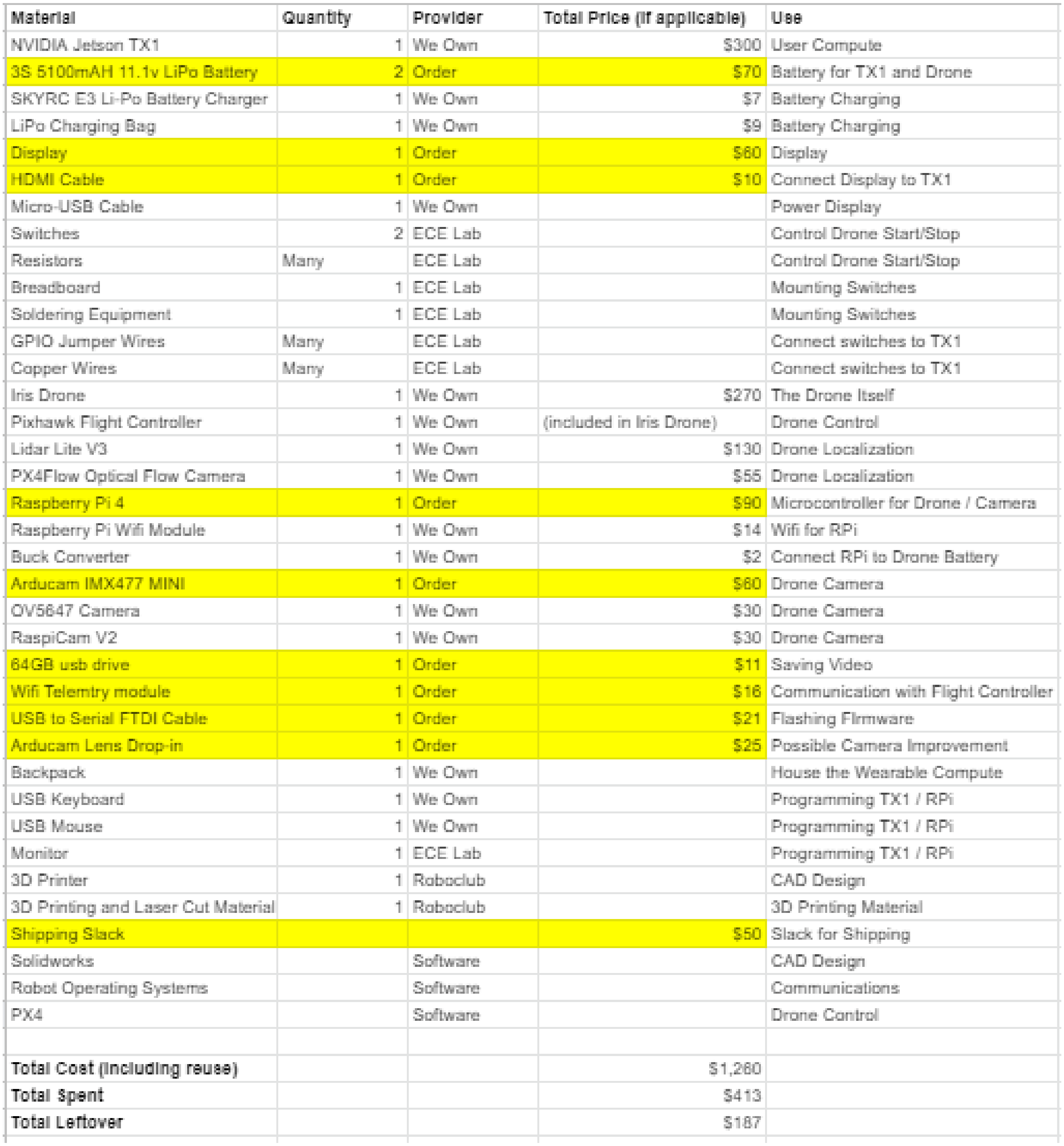
11 APPENDIX



		February	March					April				May	
		Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15
1	**Phase 1: Design**	⬛	⬛	⬛									
1.1	Make overall block diagram	🟩											
1.2	Choose/finalize components	🟩											
1.3	Set-Up Jetson TX1	🟩											
1.4	Order components		🟩										
1.5	Familiarize ourselves with drone API and simulator			🟩									
1.6	Familiarize ourselves with Jetson TX1 API			🟩									
1.7	Familiarize ourselves with wearable display communication protocol			🟩									
1.8	Design Presentation		🟩										
1.9	Design Report			🟩									
2	**Phase 2: Pre-Integration**				⬛	⬛	⬛						
2.1	Implement color filtering and blob detection				🟥								
2.2	Test detection on static images					🟥							
2.3	From target data across multiple frames, calculate velocity vector				🟦								
2.4	Use calculated velocity along with a model for target movement to accurately predict future movements					🟦							
2.5	Test detection on live video (long distance with smaller target since blob detection parameters need to be tuned)						🟦						
2.6	Successfully send sample motion commands to drone in simulation, then on physical drone				🟨								
2.7	Design a general motion planning stack; design to easily integrate with target tracking					🟨							
2.8	Test and debug the motion planning stack						🟨						
2.9	Implement communication between camera and RPi Zero, film sample video					🟥							
2.10	Implement video streaming over Wifi between RPi and TX1						🟥						
2.11	Implement hardware protocol (ex. UART, I2C) to interface between TX1 and display						🟥						
3	**Phase 3: Integration**							⬛	⬛				
3.1	Map target's motion in video to desired motion of drone							🟨					
3.2	Design circuitry for the wearable device (ex. display, buttons, power supply etc.)							🟥					
3.3	Hook up buttons to the TX1 and configure them to send start and stop instruction to RPi								🟥				
3.4	Integrate RPi start stop signals received from nano with the drone flight controller							🟦					
3.5	Design safety fallback behavior and validate on simulation and on drone							🟨					
3.6	Integrate the flight parameter data generated by TX1 with the flight controller								🟨				
3.7	Create housing for Jetson TX1 and display so that they can easily be worn by the user								🟦				
3.8	Create chassis to house RPi and camera onto the drone and verify camera angle								🟦				
4	**Phase 4: Performance Testing**									⬛	⬛		
4.1	Verify that video streaming performs with a stationary target									🟩			
4.2	Verify drone can continuously track target during flight w/o streaming video									🟩			
4.3	Verify that drone can match speed of target that is varying between speeds									🟩			
4.4	Verify live video from drone can be seen on wearable display for a moving target									🟩			
4.5	Iterate and repeat tests									🟩			
4.6	Debug issues with drone performance on tests										🟩		
5	**Documentation**											⬛	⬛
5.1	Record demo video											🟩	
5.2	Edit video											🟩	
5.3	Final poster											🟩	
5.4	Final report												🟩

	Legend:
🟥	Vedant
🟨	Alvin
🟦	Siddesh
🟩	Together

Figure 11: Final Gantt Chart



Material	Quantity	Provider	Total Price (if applicable)	Use
NVIDIA Jetson TX1	1	We Own	$300	User Compute
3S 5100mAH 11.1v LiPo Battery	2	Order	$70	Battery for TX1 and Drone
SKYRC E3 Li-Po Battery Charger	1	We Own	$7	Battery Charging
LiPo Charging Bag	1	We Own	$9	Battery Charging
Display	1	Order	$60	Display
HDMI Cable	1	Order	$10	Connect Display to TX1
Micro-USB Cable	1	We Own		Power Display
Switches	2	ECE Lab		Control Drone Start/Stop
Resistors	Many	ECE Lab		Control Drone Start/Stop
Breadboard	1	ECE Lab		Mounting Switches
Soldering Equipment	1	ECE Lab		Mounting Switches
GPIO Jumper Wires	Many	ECE Lab		Connect switches to TX1
Copper Wires	Many	ECE Lab		Connect switches to TX1
Iris Drone	1	We Own	$270	The Drone Itself
Pixhawk Flight Controller	1	We Own	(included in Iris Drone)	Drone Control
Lidar Lite V3	1	We Own	$130	Drone Localization
PX4Flow Optical Flow Camera	1	We Own	$55	Drone Localization
Raspberry Pi 4	1	Order	$90	Microcontroller for Drone / Camera
Raspberry Pi Wifi Module	1	We Own	$14	Wifi for RPi
Buck Converter	1	We Own	$2	Connect RPi to Drone Battery
Arducam IMX477 MINI	1	Order	$60	Drone Camera
OV5647 Camera	1	We Own	$30	Drone Camera
RaspiCam V2	1	We Own	$30	Drone Camera
64GB usb drive	1	Order	$11	Saving Video
Wifi Telemtry module	1	Order	$16	Communication with Flight Controller
USB to Serial FTDI Cable	1	Order	$21	Flashing Firmware
Arducam Lens Drop-in	1	Order	$25	Possible Camera Improvement
Backpack	1	We Own		House the Wearable Compute
USB Keyboard	1	We Own		Programming TX1 / RPi
USB Mouse	1	We Own		Programming TX1 / RPi
Monitor	1	ECE Lab		Programming TX1 / RPi
3D Printer	1	Roboclub		CAD Design
3D Printing and Laser Cut Material	1	Roboclub		3D Printing Material
Shipping Slack			$50	Slack for Shipping
Solidworks		Software		CAD Design
Robot Operating Systems		Software		Communications
PX4		Software		Drone Control
Total Cost (including reuse)			$1,260	
Total Spent			$413	
Total Leftover			$187	

Figure 12: Bill of Materials