FALCON

CP2106 Independent Software Development Project

NUS Orbital 2025

Apollo 11

Team No. 7003

By:

Joel Ku and Kenneth Wong

Contents

1. Overview

1. About

FALCON is a vision-based dynamic drone landing system that allows Unmanned Aerial Vehicles (UAVs) to land autonomously on moving platforms. It presents users with a Python-based GUI that abstracts backend algorithms for users to adapt FALCON's capabilities into any drone, fine tune parameters and simulate in real time. At its core, FALCON uses AprilTags for real-time visual localization, sensor fusion for motion estimation, and a Proportional-Integral-Derivative (PID) controller with noise filtering for precise trajectory correction. As a modular and reusable software library, FALCON aims to be easily extensible for all users with various hardware and thus benefit the open-source robotics ecosystem.

2. Problem Statement

Most commercial and hobbyist drones today are capable of landing only on static surfaces. However, real-world use cases such as maritime rescue, autonomous delivery, or mobile base transfers demand landings on moving targets (e.g., a boat, ground vehicle, or even another drone). This introduces challenges like uncertain platform motion, variable environmental conditions, and the need for continuous visual feedback, to achieve a robust and real-time trajectory estimation.

3. Posters, Proof-of-Concept, FALCON & Project Log

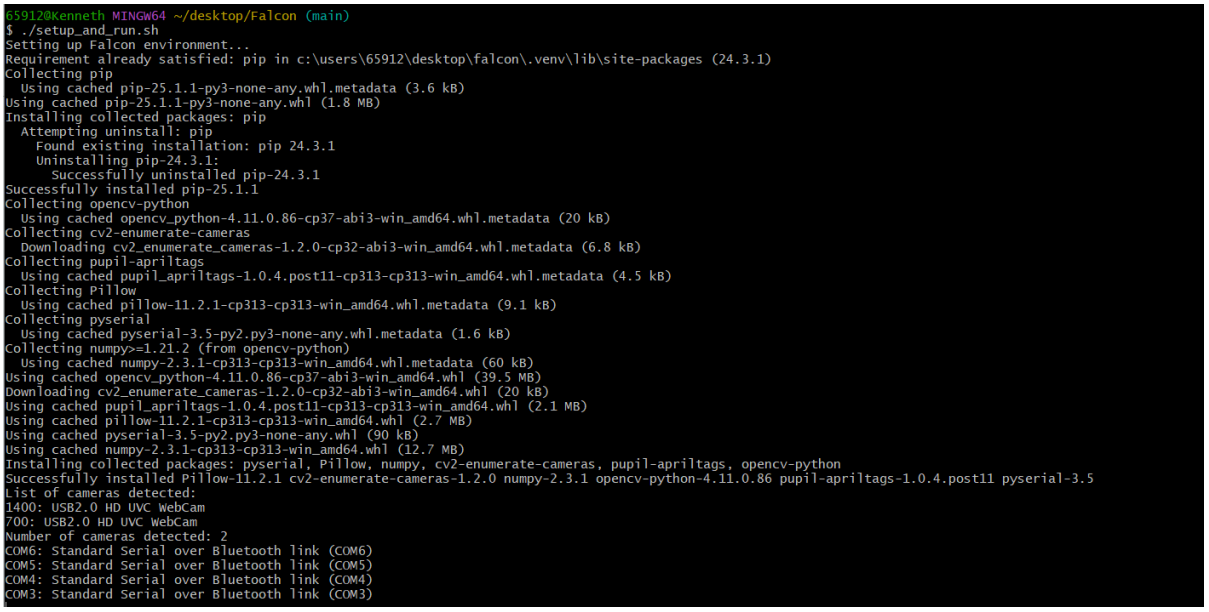
Posters for FALCON	Poster
Proof-of-Concept Video Demonstration	Video (past)
FALCON is available for all on this Github Repository To run Falcon, please refer to the README on the github page, or here	GitHub
Link to GUI	GUI
Our Project Log	Project Log

4. Running and Testing FALCON for Users

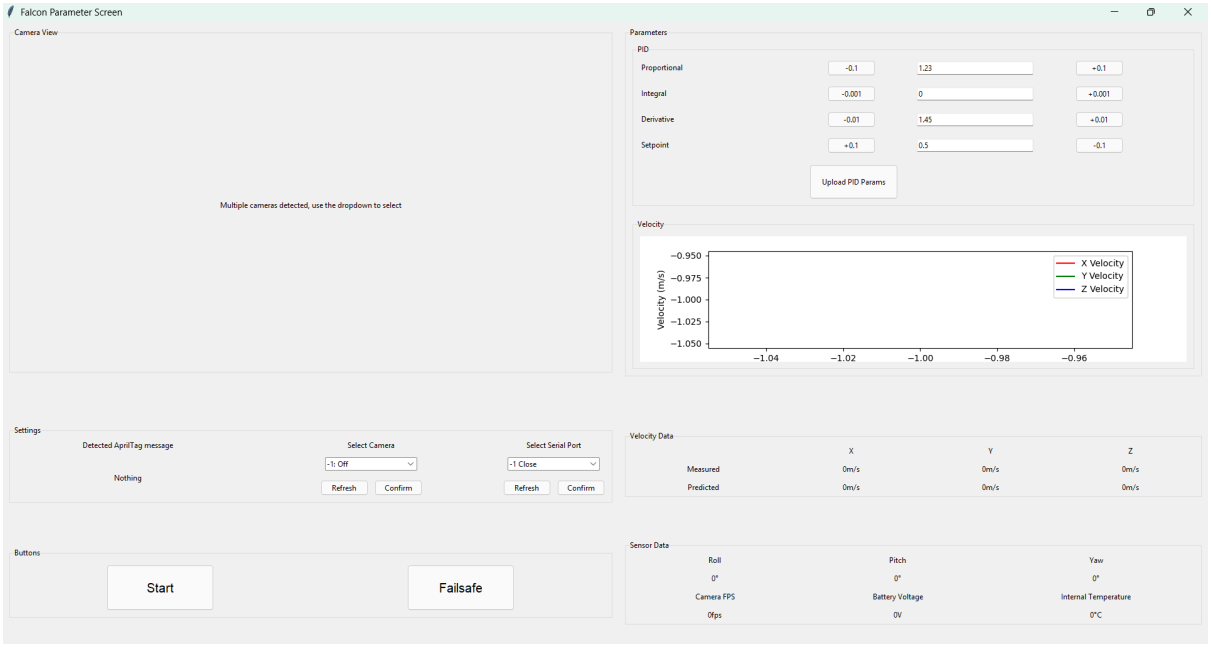
Users may refer to GitHub's README to run FALCON locally, with an included .pdf as an example apriltag to test and calibrate with. In summary, users will need to:

1. Download the project or clone the repository locally, such as on their desktop
2. Setup a Telegram bot and paste the HTTP API key into `creds.json`
3. Simply run the `setup_and_run.sh` script on bash terminal (e.g. Git Bash for Windows Users). The script installs necessary modules and runs the GUI.
4. Under 'Select Camera' dropdown, select cam starting with '700' and 'Confirm'
5. Open 'recursive_tag_0_1_2.pdf' on another screen and put it in front of the webcam.

On the Bash terminal, users can expect to see the lines shown in the screenshot below. The FALCON GUI will be generated shortly after.



Users may then refer to our Features page to test FALCON's implemented features.



Example of the main FALCON GUI

2. User Stories

As passionate individuals in robotics and integrated systems, we have had many interactions from users that range from drone hobbyists to full-time robotics engineers in the robotics community. We collated our findings below:

1. As a UAV operator, I want the drone to **autonomously detect** and **land on a moving platform** so that I do not have to manually pilot it during landing operations.
2. As a system integrator, I want to **fuse sensor data from the Inertial Measurement Unit (IMU) and vision module** so that the drone can make accurate trajectory corrections in real time.
3. As a software developer, I want to **adjust the PID gains from a user interface** so that I can fine-tune the drone's responsiveness without editing the code manually.
4. For testing purposes, I want to **simulate tag detection and IMU data using mock inputs** so that I can test the system without requiring the full hardware setup.
5. As a flight control engineer, I want the system to **initiate a fail-safe hover** if the tag is lost or tracking becomes erratic so that I can prevent crashes.
6. As a robotics researcher, I want access to a **logging system** that records sensor data, tag detections, and control outputs so that I can analyze and debug the drone's behavior post-flight.
7. For visualization, I want a **dashboard** that shows the drone's real-time status, predicted trajectory, and tag tracking confidence so that I can monitor the system’s performance during flight.
8. As a developer, I want to run **unit and integration tests on individual modules** (vision, sensor fusion, control) so that I can ensure each component works reliably in isolation and together.

9. As an electrical engineer, I want to **verify sensor connections** (IMU, camera) through a diagnostics panel so that I can detect hardware malfunctions early.
10. As a team collaborator, I want to work on **modular components** using GitHub branches and pull requests so that multiple contributors can develop features in parallel without conflicts.

These gave us meaningful insights into the useful features and capabilities that FALCON should incorporate.

3. Tech Stack

Falcon integrates a carefully chosen set of software libraries and hardware components that together form a powerful, portable, and modular system for dynamic drone landings. Each item in the stack is selected based on real-world relevance, performance, and its ability to scale with future system extensions.

1. OpenCV and AprilTag

We use OpenCV, an industry-standard open-source library for real-time computer vision tasks, due to its robust support for image processing, matrix math and camera calibration. OpenCV allows FALCON to perform detection and tracking of AprilTags, which are placed on moving platforms, and rectify imaging distortions. AprilTag is a popular camera-based technology in robotics fields, with a working principle similar to QR codes, but designed specifically as fiducial markers for visual detection and localisation. They have a lower resolution than other markers, and can therefore be detected at longer distances. Integrating the well-documented OpenCV and the reliable AprilTag into FALCON enables the system to be easily integrated into other Python modules.

2. Custom MCU Sensor Board

We chose to manufacture a custom Printed Circuit Board, complete with various sensors to report the state of the drone. For roll, pitch and yaw, we have integrated a 6-axis accelerometer, and a 3-axis magnetometer. These sensors, coupled with an Extended Kalman Filter (EKF), give a reliable estimation of the current state of the drone. The GUI shows the values of these, along with other sensor data, for the operator to easily visualize the current state of the drone while it is landing. The MCU of choice, a Teensy 4.1, sends the data to Python through the PySerial library, and the MCU is connected to the host computer through USB UART.

3. Tkinter-based GUI

FALCON’s graphic user interface is built from Tkinter. It serves as a desktop GUI for tuning, visualising and interacting with the drone landing system in real-time or

during post-analysis. Tkinter is Python's standard GUI library, which makes it fast to prototype and easy to integrate, particularly with OpenCV and Matplotlib. Together, they provide a lightweight alternative to heavy web-based dashboards, ideal for local use on low-power devices like the Raspberry Pi. It provides real-time feedback on what FALCON sees (i.e., AprilTag detection visualization), adjustable control parameters (i.e., PID gains, Kalman tuning) through sliders or input fields, logging control (i.e., start/stop recording telemetry), etc.

4. Matplotlib

We chose this tool for both our simulation and real-time logging and visualisation of FALCON's metrics by plotting AprilTag detection success/failure rate over time, position prediction vs actual tag position, sensor fusion drift over time and PID output vs error. This is useful for our offline analysis, debugging, testing simulated data and generating reports, to ensure that our AprilTag detection, PID controller and noise filtering, and drone trajectories are accurate. This is integrated into FALCON's Tkinter-based GUI.

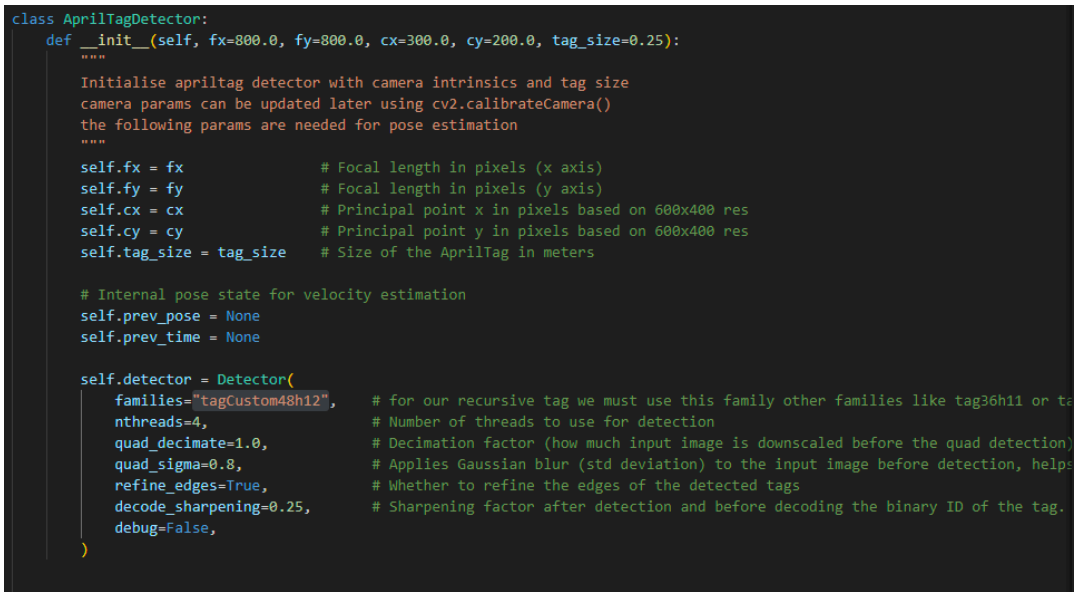
5. Raspberry Pi 4B, Raspberry Pi Camera and Teensy4.1

This hardware trio forms one of the many possible main compute units for FALCON. The RPi acts as the main processor, running the vision pipeline, trajectory prediction, and sensor fusion. It's powerful, lightweight, Linux-based, and widely supported by the community. The Raspberry Pi Camera captures high-resolution video at low latency and integrates seamlessly with OpenCV for AprilTag detection. The Teensy4.1 is a versatile microcontroller that allows FALCON to interface with peripherals that may be on the drone. Together, this hardware setup is a drone-compatible, affordable, and modular compute environment for anyone to run FALCON.

4. Features

1. Real-Time AprilTag Recognition and Recursive Tag Design

FALCON achieves robust and real-time AprilTag-based visual navigation by leveraging a modular software pipeline that incorporates a live camera feed, Python-based image processing using OpenCV, and precise pose estimation using AprilTags. Instead of using readily available wrappers, we followed the logic presented in the original C-based AprilTag detection system and papers developed by APRIL Lab at the University of Michigan [1].. The entire AprilTag recognition logic is encapsulated within our `AprilTagDetector` class, implemented in its own wrapper in the `AprilTagDetection.py` module. We expose the parameters in the constructor for users to input key information like camera intrinsics, which differs for cameras used, as explained by the Pinhole camera model. [2]



```python
class AprilTagDetector:
    def __init__(self, fx=800.0, fy=800.0, cx=300.0, cy=200.0, tag_size=0.25):
        """
        Initialise apriltag detector with camera intrinsics and tag size
        camera params can be updated later using cv2.calibrateCamera()
        the following params are needed for pose estimation
        """
        self.fx = fx                # Focal length in pixels (x axis)
        self.fy = fy                # Focal length in pixels (y axis)
        self.cx = cx                # Principal point x in pixels based on 600x400 res
        self.cy = cy                # Principal point y in pixels based on 600x400 res
        self.tag_size = tag_size    # Size of the AprilTag in meters

        # Internal pose state for velocity estimation
        self.prev_pose = None
        self.prev_time = None

        self.detector = Detector(
            families="tagCustom48h12",    # for our recursive tag we must use this family other families like tag36h11 or ta
            nthreads=4,                   # Number of threads to use for detection
            quad_decimate=1.0,            # Decimation factor (how much input image is downscaled before the quad detection)
            quad_sigma=0.8,               # Applies Gaussian blur (std deviation) to the input image before detection, helps
            refine_edges=True,            # Whether to refine the edges of the detected tags
            decode_sharpening=0.25,       # Sharpening factor after detection and before decoding the binary ID of the tag.
            debug=False,
        )
```

Class definition of our `AprilTagDetector`

---

[1] https://docs.wpilib.org/en/stable/_downloads/e72e01c5464f1a0838751a5cb158087e/krogius2019iros.pdf

[2] https://en.wikipedia.org/wiki/Pinhole_camera_model

```python
def _process_detection(self, detection):
    """
    Process a single detection to extract pose and velocity information.
    """
    # Extract the pose of the tag
    pose_t = detection.pose_t #3x1 translation vector
    pose_R = detection.pose_R #3x3 rotation matrix

    # Pose estimation: compute the camera position relative to the tag
    # using the pinhole camera model: camera position is the negative of the rotation matrix transposed multiplied
    pose = -np.matmul(pose_R.T, pose_t)

    # Compute velocity of the tag
    current_time = time.time()
    velocity = np.zeros(3) # Initialize velocity to zero

    # Calculate the velocity if we have a previous pose
    if self.prev_pose is not None and self.prev_time is not None:
        dt = current_time - self.prev_time
        if dt > 0:
            velocity = (pose - self.prev_pose).flatten() / dt

    # Update state for the next detection
    self.prev_pose = pose
    self.prev_time = current_time
```

Code snippet of the detection process

AprilTag Detection: Our Study of the Industrial Standard

Our implementation adapts the advanced and optimized AprilTag Version 3 (v3) detection algorithm as described by Olson et al. (Wang & Olson, 2016; Krogius et al., 2016)[3]. The key innovations of AprilTag v3, such as reduced false positive rates, faster quad fitting, and improved pose estimation accuracy, make it particularly well-suited for drone localization and landing tasks under challenging real-world conditions.

AprilTags are visually encoded fiducial markers that rely on a lexicode design to achieve a guaranteed minimum Hamming distance between tags within the same family, enabling robust detection even under conditions of partial occlusion, perspective distortion, or poor lighting. As described by Wang and Olson (2016), the AprilTag detection algorithm proceeds through the following major stages:

1. **Preprocessing**: The input frame is first converted to grayscale to reduce computational load.

[3] J. Wang and E. Olson, "AprilTag 2: Efficient and robust fiducial detection," 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Daejeon, Korea (South), 2016, pp. 4193-4198, doi: 10.1109/IROS.2016.7759617.

2. **Segmentation**: A thresholding technique is used to segment the image into light and dark regions.

3. **Component Clustering**: Union-find algorithms group connected components, and adjacent light-dark pairs are formed.

4. **Quad Fitting**: Candidate quads are formed by evaluating all possible corner permutations and fitting lines using principal component analysis (PCA). We then iterate all permutations of four candidate corners, fitting lines to each side of the candidate quad and select the four corners that result in the smallest mean squared line fit errors.

5. **Decoding & Pose Estimation**: The contents of the quad are decoded using a codebook with built-in error correction. If decoding is successful, a 3D pose is estimated based on known tag size and calibrated camera intrinsics.

In other words, this approach allows the system to reliably distinguish it from other tags in the set even when a tag is viewed from different angles or partially obscured. This built-in error correction capability dramatically reduces the likelihood of false positives, especially in complex or cluttered environments where natural textures, shadows, or patterns might otherwise be misinterpreted as valid tags.

<u>Recursive Apriltag: Our own Implementation for FALCON</u>

In the early stages of FALCON development, we experimented with linear arrays of AprilTags, placed sequentially in a runway-style layout. This design was meant to guide the drone progressively along a path. However, such a configuration proved to be space-inefficient and not well-aligned with the vertical takeoff and landing (VTOL) capabilities of multirotor drones. The linear setup also introduced challenges in maintaining consistent detection across varying altitudes and angles of approach.

After researching on the various AprilTag implementations and family IDs, we adopted a recursive tag approach, where multiple AprilTags of decreasing size are

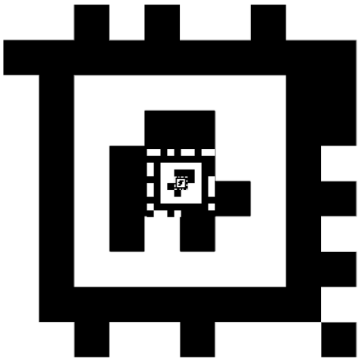
nested concentrically within each other. We found an AprilTag family that is able to implement recursive implementation, with an AprilTag placed within an AprilTag. This implementation significantly increases the space efficiency of our landing algorithm, where the size of the largest, outermost tag is the overall size of the entire landing platform. Going with the assumption that the drone is able to land vertically, the idea of having AprilTags placed within AprilTags works well for drones to navigate towards and land on the middle of the platform. The tag family `"tagCustom48h12"` is specifically chosen to allow the use of a recursive AprilTag, as it has an empty white square at the center of the tag, allowing the placement of another AprilTag within a tag. Most importantly, unlike other families (e.g., `tag36h11` or `tagStandard41h12`), this layout allows us to maintain strong visual separation between adjacent tag borders, which prevents mutual occlusion or decode corruption. This recursive layout enhances robustness in two ways:

- Redundant Detection: If the outer tag is partially out of frame or occluded, the inner tags are still detectable, ensuring continuity in localization.
- Progressive Accuracy: As the drone descends, smaller inner tags dominate the frame, naturally leading to finer-grain pose estimation closer to touchdown.

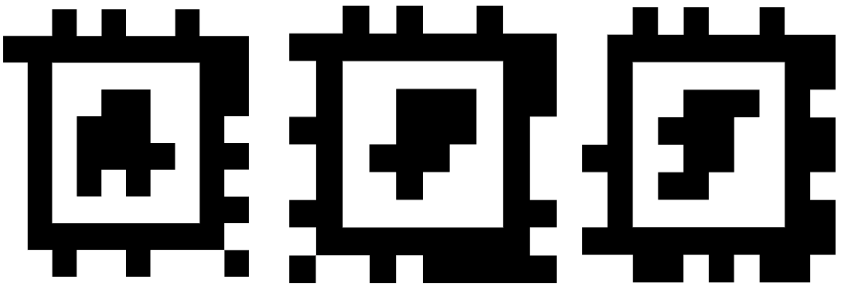
The recursive design enables hierarchical tracking where different tags serve different purposes depending on the drone's altitude and field of view.

Illustration below:

- A large tag (ID 0) contains a medium tag (ID 1), which itself contains a small tag (ID 2).
- The drone can detect all three tags at different stages of descent, providing robust fallback detection and dynamic precision scaling.



Custom recursive tag, with ID2 in ID1 in ID0



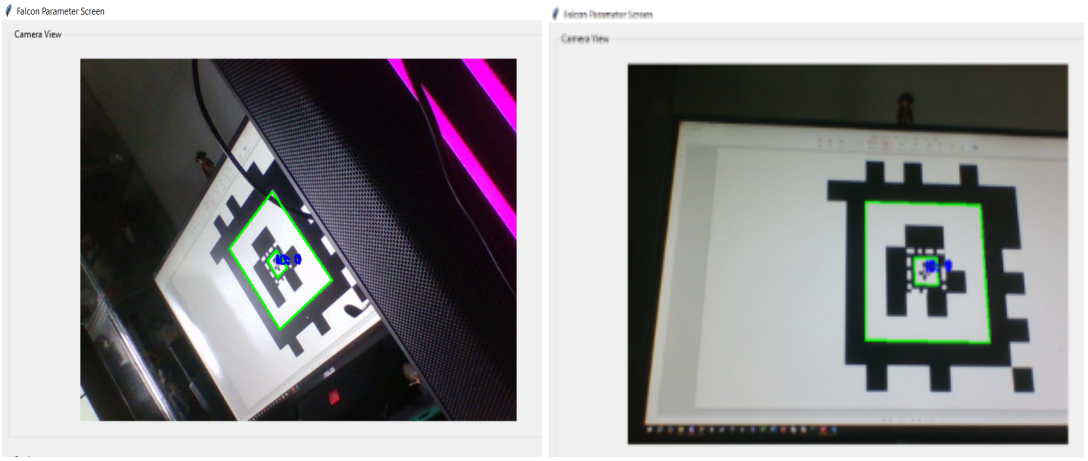
tagCustom48h12 tags, from left to right: ID 0, ID 1, ID 2

FALCON allows for Greater Optimization for Robustness and Accuracy

FALCON exposes all detector parameters (such as focal length, decimation, blur, sharpening, and resolution center) via the constructor of AprilTagDetector, allowing full control over performance and accuracy trade-offs:

- Decimation (quad_decimate): Adjusted to balance speed and detail. Low decimation maintains full resolution for pose fidelity.
- Gaussian Blur (quad_sigma): Applied to reduce image noise and stabilize edge detection.
- Sharpening (decode_sharpening): Tuned to maximize decoding reliability for smaller, inner tags.
- Camera Intrinsics: Manually calibrated or derived via OpenCV to improve the accuracy of 6-DoF pose estimation.

Through empirical testing (e.g. demonstrated in the pictures below), we validated this setup against varied lighting, distance, and orientation conditions. The recursive AprilTag system consistently maintained robust detection across altitudes and during lateral camera motion.

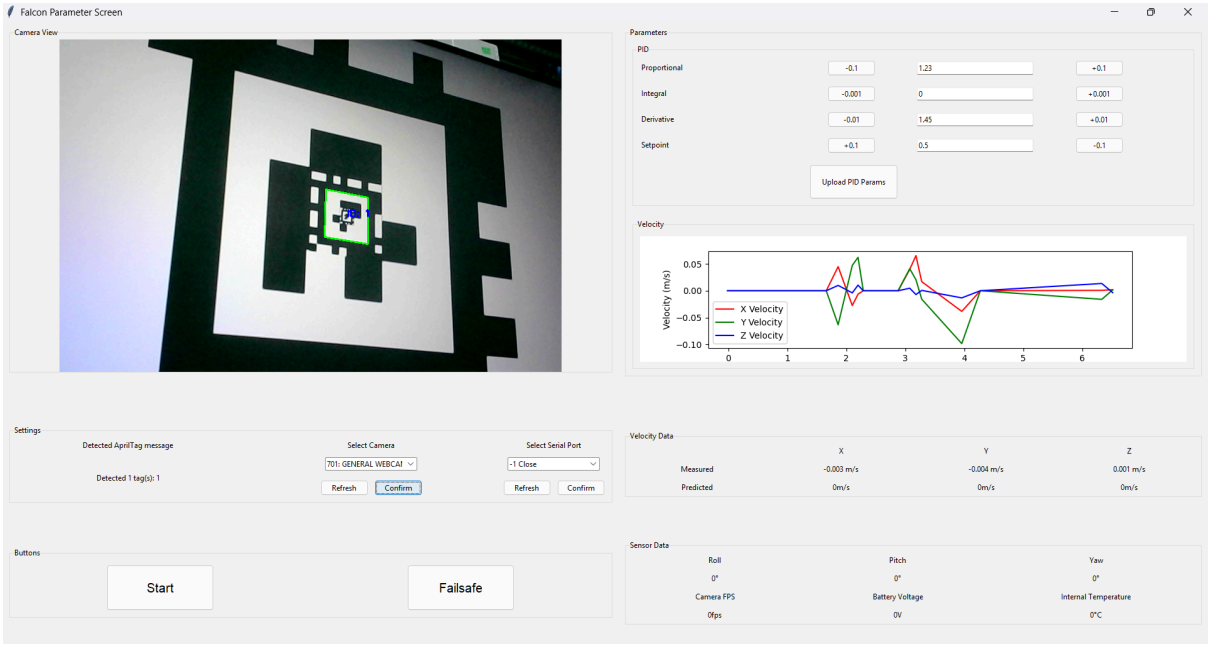


Configurable live feed from Camera with Overlay

The real-time AprilTag recognition pipeline runs continuously on the live camera feed, even when the drone's control logic is inactive. This ensures that the GUI can immediately display detected tags, including:

- ID overlays,
- tag boundaries,
- and detection confirmation text.

This continuous detection loop supports smooth user feedback and simplifies debugging of visual tracking behaviors.



Sample of FALCON GUI, with a detected AprilTag

2. Motion approximation using AprilTag frame sequences

By tracking the sequence of AprilTag positions over time, FALCON is able to estimate the velocity and direction of the moving platform. This temporal information allows the algorithm to model the platform's trajectory and anticipate its future position, which is critical for smooth and successful landing. Data is then returned in a structured format for the controls algorithm and Tkinter GUI.

```python
def new_detection(self, detection):
    """
    Process a new detection and update the pose and velocity
    :param detection: Detection object containing pose_R and pose_t
    :return: None
    """
    if self._last_time is None:
        self._last_id = detection.tag_id
        self._last_time = time.time()
        self._last_corners = detection.corners
    elif self._last_corners is not None:
        if detection.tag_id != self._last_id:
            self._last_time = time.time()
            self._last_id = detection.tag_id
            self._last_corners = detection.corners
        else:
            # Velocity logic here
            curr_time = time.time()
            dt = curr_time - self._last_time
            if dt > 0:
                if (detection.tag_id == 0):
                    scale = self.tag_size_0
                elif (detection.tag_id == 1):
                    scale = self.tag_size_1
                elif (detection.tag_id == 2):
                    scale = self.tag_size_2

                curr_xy = np.mean(detection.corners, axis=0)
                last_xy = np.mean(self._last_corners, axis=0)

                curr_width = self._tag_width_pixels(detection.corners)
                last_width = self._tag_width_pixels(self._last_corners)

                # X/Y velocity based on center movement
                delta_xy = curr_xy - last_xy
                self.velocity[:2] = delta_xy * (scale / curr_width) / dt

                # Z velocity based on tag size change
                width_change = (curr_width - last_width) / last_width
                self.velocity[2] = width_change * scale / dt

                self._last_time = curr_time
                self._last_id = detection.tag_id
                self._last_corners = detection.corners
```

Code snippet of our final velocity estimation process

Initial attempts

For our velocity estimation, we first attempted to do pose estimation, which performed matrix multiplication on `pose_R` and `pose_t` for every detection, which

gives the estimated pose of the AprilTag, relative to the camera lens. Pose refers to the position and orientation of an object in 3D space, and with such information about the AprilTag from one frame to another, estimations can be made about the velocity. `pose_R` and `pose_t` from the AprilTag detector refer to the rotational pose and translational pose respectively.

First, we computed the pose by performing matrix multiplication on `pose_R` (transposed) and `pose_t`. This rotates the translation vector into the tag's coordinate space, and thus provides the 3D position of the camera in the coordinate frame of the AprilTag. Next, we subtract the pose of the AprilTag within the current frame from the pose of the AprilTag of the previous frame, ensuring that the AprilTag ID is consistent across both frames. Lastly, we divide the difference by the elapsed time between each frame, which provides the velocity data.

The original velocity estimation performed this calculation within the detector class, which resulted in poorer detection, as the detection process was slowed down due to the additional calculations needed to be done. Furthermore, the detection class was now also burdened with storing the previous detection's pose.

To alleviate the processing load of the detector class, we created a new estimator class to handle the estimations using the data returned from the detector class, which not only sped up the detection processing, but also allowed us to test and modify the velocity estimation algorithm much quicker. This also gave us better separation of concerns, where we were able to make modifications to one class without affecting the other.

However, the initial speed estimation algorithm did not take into account the size of the AprilTag, and would be difficult to retroactively implement into, as `pose_R` and `pose_t` are returned from the AprilTag detector class. As the size of the AprilTag is used for scale and thus accurate velocity scaling, we needed another method of velocity estimation from the AprilTag data returned by the detector class.

Descriptor	**Returned data**
tag_family	b'tagCustom48h12'

tag_id	0
hamming	0
decision_margin	117.32499694824219
homography	[[1.96279663e+02 7.00215396e-02 7.00879619e+02] [2.84516104e-01 1.95782316e+02 3.50635899e+02] [8.20551468e-04 -1.00661752e-04 1.00000000e+00]]
center	[700.87961917 350.63589894]
corners	[[505.13531494 546.63726807] [896.5838623 546.30944824] [896.26361084 154.99531555] [504.8934021 154.68041992]]
pose_R	[[9.99999111e-01 -1.02890571e-04 -1.32921475e-03] [1.04725556e-04 9.99999042e-01 1.38050754e-03] [1.32907144e-03 -1.38064552e-03 9.99998164e-01]]
pose_t	[[0.25586079] [0.09613674] [0.51072226]]
pose_err	2.3754452063594288e-08

Table showing 1 AprilTag (ID 0) detection, as returned by the detector class

Reliable velocity estimation

By observing the detection data, our next step was to use either corner or center data, by tracking the positional change of AprilTag from detection to detection. These were primarily considered as they were references to the camera's pixel count, and made it easy to perform velocity scaling based on the accurate, real life size of the AprilTag that is detected. The detection class is able to return the corner detections reliably, as the overlay detection drawing reliably shows the box around the AprilTag. As we would require velocity in all 3 directions (x, y and z) from the detection, we decided against using center tracking, as it would not provide enough information to calculate the velocity in the z-axis.

To check the number of pixels that make up the length and breadth of the detected AprilTag, we calculate the Euclidean distance between the corners. With these 2 measurements, we would also be able to calculate both the skew of the AprilTag in

relation to the camera position, which gives us another frame of reference for positioning. The ID of the detected AprilTag is then used to reference the scaling, where the number of pixels are scaled to meters, based on the actual size of the AprilTag.

Tag ID	Intended size of AprilTag	Tested size of AprilTag
0	1.25m	0.161m
1	0.25m	0.031m
2	0.05m	0.0055m

Table showing actual size of AprilTags

As we were unable to source a 1.25x1.25 meter AprilTag, we scaled down the recursive testing AprilTag, which allowed it to fit on a 24” 1920x1080 monitor at 100% scaling that was our main test display. By ensuring that we used the same image and monitor for all testing, the size of the AprilTag would remain constant, eliminating a source of error.

To obtain the x and y velocities, the rate of change in the length of the 2 detected frames was calculated using the time difference between the 2 detections. For velocity in the z direction, the width was considered instead.

Preliminary testing showed that this velocity estimation algorithm was much more reliable compared to the previous method. However, there was another issue, where the detector class was not tracking the tag IDs of the 2 detections. This caused velocity spikes due to the different physical sizes of the AprilTags, which is a major safety concern if this were to be deployed on a drone, as it would cause erratic motion. This was fixed by ensuring that the tag ID between 2 detections is the same before any velocity estimation is performed. While we have encountered problems with such simple tag ID comparison, where multiple tags with the same ID are detected together, causing weird velocity readings, we feel that this problem would not be a concern for our intended use case, which is an outdoor environment, where it is extremely unlikely for the camera to pick up any other AprilTag in the surroundings.

3. Custom IMU Sensor Fusion and Yaw Stabilisation for Drift-Free Attitude Estimation

Within `Falcon/Backend/SensorFusion/SensorsBoard/src/` lies a fully custom sensor fusion pipeline to interpret 9-degree-of-freedom (DOF) IMU data, which includes measurements from a 3-axis accelerometer, gyroscope, and magnetometer. The goal is to reliably estimate the UAV's orientation (roll, pitch, and yaw) in real time, minimizing noise, drift, and axis coupling effects.

Sensor Data Acquisition and Calibration: Our Initial Tests

The raw sensor readings, linear acceleration, angular velocity, and magnetic field strength, are streamed over USB UART to the FALCON system. Prior to use, each sensor is individually calibrated: Accelerometer and gyroscope offsets are computed during static initialization while magnetometer is calibrated through environmental scanning to determine hard-iron and soft-iron distortions. Calibration was performed outdoors to minimize magnetic interference, such as from power lines or tables.

Despite this, anomalies were noted during initial experiments, particularly that the yaw angle would shift significantly even during pure roll or pitch movements. This led to an extensive investigation of yaw computation, distortion correction, and filter behavior.

Orientation Estimation and Coupling Effects: The Problem

Our sensor fusion initially employed the Madgwick filter, which is known for combining IMU data to estimate quaternion-based orientation. This filter uses gyroscope integration, accelerometer direction vectors, and magnetometer heading to compute orientation.

However, during pitch or roll, unexpected variations in yaw were observed: even when no actual yaw motion occurred. This phenomenon is known as roll-pitch-yaw coupling, where tilt-induced changes in sensor alignment with the Earth's magnetic field cause yaw readings to become unstable or incorrect.

We found these problems:

- The acceleration vector, used to determine the "up" direction (gravity), was too noisy to reliably correct magnetometer tilt using traditional projection methods.
- The magnetometer heading changes during roll and pitch, because the magnetic field vector is no longer aligned with the horizontal plane in the sensor's rotated body frame.
- Attempts to correct this using Madgwick's internal gravity projection (`getGrav()`) showed inconsistency, especially in the Z-component.

Quaternion-Based Correction and Rotational Decoupling: Our Solution

After working on the trigonometry of 2 planes of roll and pitch to see how x y and z changes, and using polar coordinates to represent tilt and resultant coordinates, we implemented a quaternion-based tilt compensation system for magnetometer correction. This involved computing a 3×3 rotation matrix from the current roll and pitch and applying this matrix to rotate the magnetometer vector into the world (level) frame before computing yaw. The implementation of our solution can be found in the snippet below. The gist is that there are 3 variables: theta_xz, theta_yz, theta_xy. The first changes with roll, since we rotate about y, which will change magX and magZ. Similarly for theta_yz it changes with pitch, since we rotate about x, which will change magY and magZ. yaw is computed by atan2(magX, magY). Thus, we concluded that roll or pitch changes magX and magY. So even if the sensors do not yaw, if magX and magY change from roll and pitch, yaw will change when it should not.

```
void getCorrectedYaw(float r, float p)
{
        // convert roll and pitch to radians
        r *= M_PI / 180.0f;
        p *= M_PI / 180.0f;
        float original_magX = magX;
        float original_magY = magY;
        float original_magZ = magZ;

        //for roll
        float polar_r_xz = sqrt(magX*magX + magZ*magZ);
        float theta_xz_init = atan2(magX,magZ);
        float polar_angle_after_roll = theta_xz_init - r;
        magX = polar_r_xz * cos(r);
        magZ = polar_r_xz * sin(r);

        //for pitch
        float polar_r_yz = sqrt(magY*magY + magZ*magZ);
        float theta_yz_init = atan2(magY,magZ);
        float polar_angle_after_pitch = theta_yz_init - p;
        magY = polar_r_yz * cos(p);
        magZ = polar_r_yz * sin(p);

        float deltaX = magX - original_magX;
        float deltaY = magY - original_magY;

        yaw = atan2(magX - deltaX, magY - deltaY) * 180/M_PI;

        // Serial.printf("ori_magX: , %f, ori_magY: %f, ori_magZ: %f, polar_r_xz: %f, theta_xz_init: %f, polar_angle_after_roll: , %f
        // Serial.printf("dumbX:, %f, dumbY, %f, dumbZ, %f, beebeeyawyaw: %f\n", magX, magY, magZ, yaw);
}
```

In other words, the magnetometer readings were transformed as follows:

$$\begin{bmatrix} m'_x \\ m'_y \\ m'_z \end{bmatrix} = \mathbf{R}_{pitch,roll} \cdot \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}$$

This decouples the roll and pitch effects from the heading calculation. The yaw angle is then computed using:

$$\text{yaw} = \text{atan2}(m'_y, m'_x)$$

This approach was far more effective than simply applying Madgwick's quaternion outputs directly, particularly for static or slow dynamic conditions. Empirical regression analysis was briefly considered but discarded due to lack of generalizability. The rotational matrix method, rooted in geometric transformation, proved to be the mathematically correct and general solution.

Extended Kalman Filtering for Sensor Fusion: Optimisation

We integrated an Extended Kalman Filter (EKF) to combine gyroscope and accelerometer data for roll and pitch estimation, while magnetometer-corrected yaw was fused separately. This two-pronged approach allowed:

- Gyroscope: High short-term precision but subject to drift.
- Accelerometer: Long-term stability, useful for drift correction in roll and pitch.
- Magnetometer: Used cautiously only for yaw, and only after tilt-compensation.

The EKF maintained internal state for each angle and bias term, correcting based on sensor variance. Kalman tuning (via Q_angle, Q_bias, and R_measure parameters) was iteratively adjusted to achieve critically damped performance.

```
void getRollPitchYawK()
{
        double dt = (double)(micros() - timer) / 1000000;
        timer = micros();
        kalRoll = kalmanX.getAngle(roll, gyroX, dt);
        kalPitch = kalmanY.getAngle(pitch, gyroY, dt);
        getCorrectedYaw(kalRoll, kalPitch);
        kalYaw = kalmanZ.getAngle(yaw, gyroZ, dt);
        PRINTLN(" kalYaw: " + String(kalYaw));
        // Serial.println(kalYaw);

        yawDepth[0] = kalYaw;

        Serial.printf("kalRoll: %.7f, kalPitch: %.7f, kalYaw: %.7f\n", kalRoll, kalPitch, kalYaw);
        //Serial.println("Orientation: " + String(kalRoll) + ", " + String(kalPitch) + ", " + String(kalYaw));


        rollPitch[0] = kalRoll;
        rollPitch[1] = kalPitch;
}
```

Stable Orientation Tracking: Result

After implementing these filtering and correction strategies, yaw drift was virtually eliminated during roll/pitch-only movements, **roll and pitch** remained accurate and low-noise under dynamic conditions and the system performed consistently in real-world test environments, even with minor magnetic anomalies. By combining classical estimation theory (EKF), 3D geometry (rotation matrices), and sensor fusion principles, we were able to overcome the limitations of consumer-grade IMU sensors. This rigorous approach allows the FALCON navigation algorithm to

confidently rely on fused IMU data for feedback control, path estimation, and AprilTag alignment. This is particularly so for orientation, which is one of the two components in determining the UAV's pose.

4. PID-based Control logic and Configurable Parameters

In FALCON's landing and approach sequence, precise motion control is critical to ensure that the UAV responds to visual cues (AprilTags in this case) in a stable, smooth, and accurate manner. To achieve this, we implemented a Proportional-Integral-Derivative (PID) controller, which is a closed-loop control feedback mechanism to control process variables in driving a system towards a desired state. This control algorithm, as its name implies, involves 3 key factors: proportional, integral and derivative. The crux of this controller is to utilise deviation from desired value, or error e(t) = $d_{desired}$ −d_{actual} to feedback into the input of the system. The Proportional Controller (= KP * error) feedbacks a correcting input proportional to its current error. For certain situations, steady state error can occur, where regardless of the gain by KP, the error will never reach zero. The Integral Controller eliminates this by correcting the system according to the cumulative error resulting from the proportional factor. Lastly, due to the high possibility of overshooting during correction which can lead to an issue of oscillation, the Derivative Factor (= KD * (curr_error - prev_error)), in contrast, uses the current rate of change of error to determine the rate at which the system is approaching the desired value. It prematurely reduces the gain in input to act as a damping factor.

The PID output is then:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{d}{dt}e(t)$$

Cascading PID Implementation in FALCON

To improve precision and stability during landing maneuvers, especially in vertical descent, FALCON uses a cascaded PID controller structure. This method employs two layers of control loops:

- Outer loop (Position PID): Controls the position error, outputs a velocity setpoint. This is how FALCON computes how fast the drone should move to correct its position.
- Inner loop (Velocity PID): Takes that velocity setpoint, compares it with the drone's actual velocity, and outputs the control effort (e.g., motor commands or acceleration). This ensures the drone follows the desired velocity trajectory accurately.

Position Error (x_target - x_actual) -> Position PID Controller (outer loop) -> Velocity Setpoint (v_desired) -> Velocity PID Controller (inner loop) -> Motor/Thrust Commands

Configurable Parameters via FALCON GUI

To empower developers and users with on-the-fly tuning, the system exposes key parameters to the UI, which are the constants Kp, Ki, Kd, and the desired setpoint.

This enables live debugging and real-world PID gain tuning without redeploying the software, dramatically accelerating the development cycle. Fine-tuning during flight tests was essential in adjusting the system's responsiveness to varying lighting, tag sizes, and IMU feedback latency.

We did a simple value check such that only float data type is acceptable as a valid input, else, an error will be thrown to the user.

5. Python GUI to Interface with FALCON in Real-Time

To facilitate seamless interaction with the FALCON system during testing and deployment, we designed and implemented a **Graphical User Interface (GUI)** in Python using Tkinter. The goal of this GUI is to provide real-time configurability, visualization, and feedback for key subsystems, most notably, the AprilTag detection output, the PID controller and velocity estimation, which are outputs of the FALCON algorithm that runs on the backend.

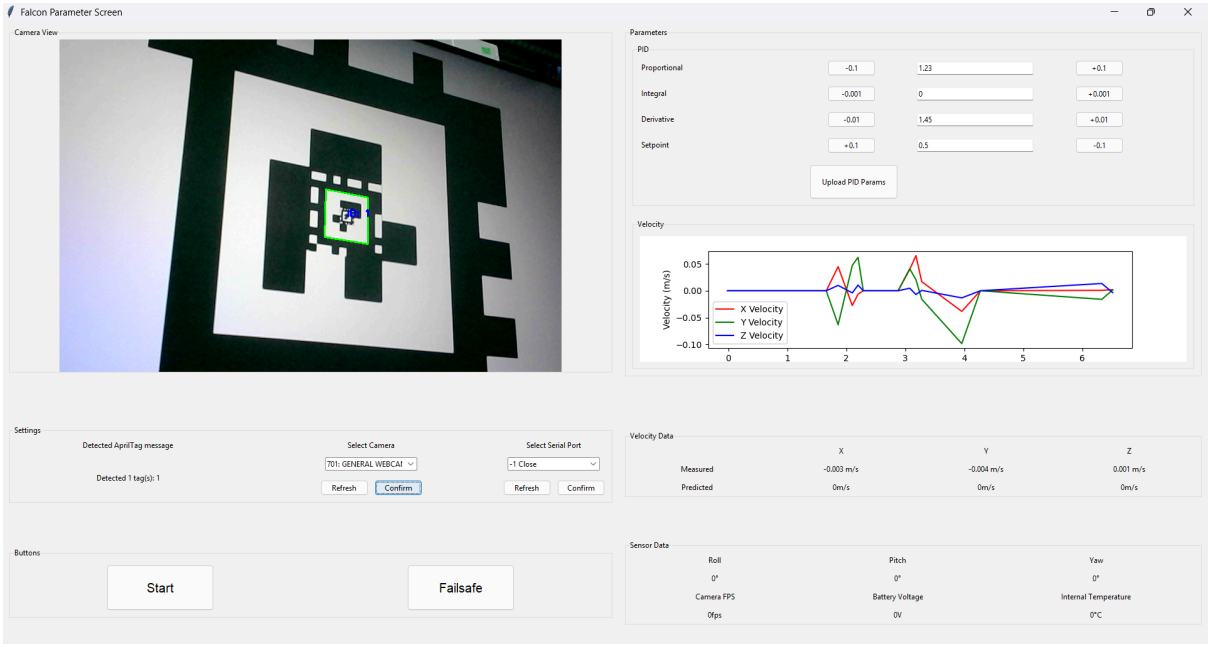
Motivation and Design Goals

While we had in mind the need for a GUI during deployment for users to receive feedback and know the state of the UAV, early testing revealed a huge inconvenience in maintenance and tuning due to hardcoded parameters and the need to restart the system after each tuning change. This was inefficient, particularly when fine-tuning PID gains and evaluating vision system feedback under changing conditions (lighting, movement, noise, etc.). The need for a proper GUI was also reaffirmed during our external robotics projects where we used GUIs like Foxglove or QGroundControl, which were essentials to interface with robots efficiently.

Thus, we designed a modular GUI layer with the following objectives. It must allow real-time modification of control parameters (e.g. PID gains), visualize AprilTag detection results (tag IDs, position, computed velocity), provide immediate feedback on the effect of parameter changes, and serve as an interface for monitoring key telemetry such as computed desired velocities and relative tag positions.

Key Features

The GUI has multiple panels for the operator to view important information and control important parameters.



1. AprilTag Camera Detection

The largest panel on the top left side is for the live feed from the camera with a OpenCV detection overlay of the AprilTags. We have a green bounding box drawn around and the corresponding ID indicated for the successfully detected AprilTags. This real-time display is complemented with a label below the camera feed that also reflects the value of the detected AprilTag in text, if any, as well as two dropdown boxes for selecting the camera and the Serial Port for communication with the MCU and obtaining sensor data. To eliminate the need to restart FALCON when plugging in new devices, we have added buttons to refresh the list of devices available to the operator.

2. Start and Emergency Stop Buttons

At the bottom left are two buttons for starting the landing algorithm and manually activating the failsafe hover mode, in case of any emergency during the landing process. The failsafe hover button is a key safety kill switch that sets the predicted speed to 0, which will cause the drone to come to a stationary hover immediately.

3. PID Constants Tuning Parameters

The input fields on the top right are for the operator to easily tune the PID parameters during operation. The operator can also easily set the setpoint, which dictates the

desired speed. This is an important feature as PID requires live tuning, since the values are unique to each and every drone, and environments. Furthermore, this feature allows operators to perform PID tuning methodologies like Zieger Nichols that require tuning each constant in a specific order.

4. Velocity Estimation

Right below the PID control tuning is the real-time graphical visualisation of the drone's velocity output using matplotlib. It is difficult for operators to keep track of the minute changes in position and orientation when it is flying in the air. Thus, it is important for operators to know the x, y and z velocities of the drone with a form of visualisation. This feature can be paired with the previously mentioned PID control tuning (e.g. Zieger Nichols tuning), where operators will need to observe the drone's oscillations in the air, and see whether the PID values have dampened them.

Numerical values of the velocity outputs are displayed below the graph, so that operators can observe the changes in velocity down to a high resolution.

5. Sensor Data Telemetry

The bottom right contains the various flight parameters and sensor outputs, including the XYZ velocities, roll, pitch, yaw from the IMU, and other derived outputs from the onboard UAV sensors. As we want operators to treat the backend algorithms as a blackbox, we abstract them such that users will only need to observe the changes in the sensor output data to know whether their sensors are accurate and suitable for use. It will also help with analysis since these values provide the localisation and orientation of the drone in flight.

6. Telegram bot telemetry integration

From our user interviews during user testing, we found out from seniors in the robotics field that having remote telemetry, accessible by someone other than the operator and the GUI screen, would make the monitoring and debugging process much simpler. In our search for a simple, web/internet-based platform that offers easy integration with Python, the Telegram bot API stood out as a beginner-friendly and user-accessible method to broadcast telemetry data remotely. Furthermore, a

Telegram bot can broadcast messages to a group, which further increases its utility in sharing telemetry data with a large group of people. Also, Telegram allows anyone with a valid Telegram account to create and host a bot, and has an extremely simple 3-step setup process to set up a new API key to host a bot. To keep our API key secure, we stored it in a JSON file and excluded it from the Git repository, while providing instructions to users on the GitHub README document on the steps to generate their own.

As our camera is running at around 30 FPS, and assuming that there are 3 AprilTags within the camera frame, there would be a lot of detection data that would be sent through the Telegram bot. As such, we decided against sending all the data collected by the detector, choosing instead to send alerts when the data values cross a threshold. This reduces the amount of unnecessary messages received, as well as reduces the load on the computer running FALCON. Additionally, the Telegram notifications could serve as a reminder for the operator and other team members who may not necessarily have line-of-sight with the drone about any urgent and important telemetry data, which could potentially prevent an accident from happening.

A popular, open-source Python library commonly used for simple Telegram bot operations is `python-telegram-bot`, which provides multiple functions for sending messages and parsing user input. We chose this library as it has all the features that we require, and has a large support community that was able to provide much insight into making a functional bot. As this Telegram bot would have to run in parallel with the rest of FALCON in order to receive live telemetry data, it was also paramount that we implemented multi-threaded operation, which was handled by Python's built-in `threading` library. As we maintained good separation of concerns and a modular design, it was easy to integrate the Telegram bot into the existing code.

```python
async def botloop_routine(API_KEY):
    print("Bot started")
    application = ApplicationBuilder().token(API_KEY).build()
    application.add_handler(CommandHandler('start', start_command))

    await application.initialize()
    await application.updater.start_polling()
    await application.start()
    try:
        while True:
            # Check the message queue for outgoing messages
            try:
                item = message_queue.get_nowait()
                if isinstance(item, tuple) and len(item) == 2:
                    chat_id, message = item
                await application.bot.send_message(chat_id=item[0], text=item[1])
            except queue.Empty:
                pass
            await asyncio.sleep(1)
    finally:
        await application.updater.stop()
        await application.stop()
        await application.shutdown()
```

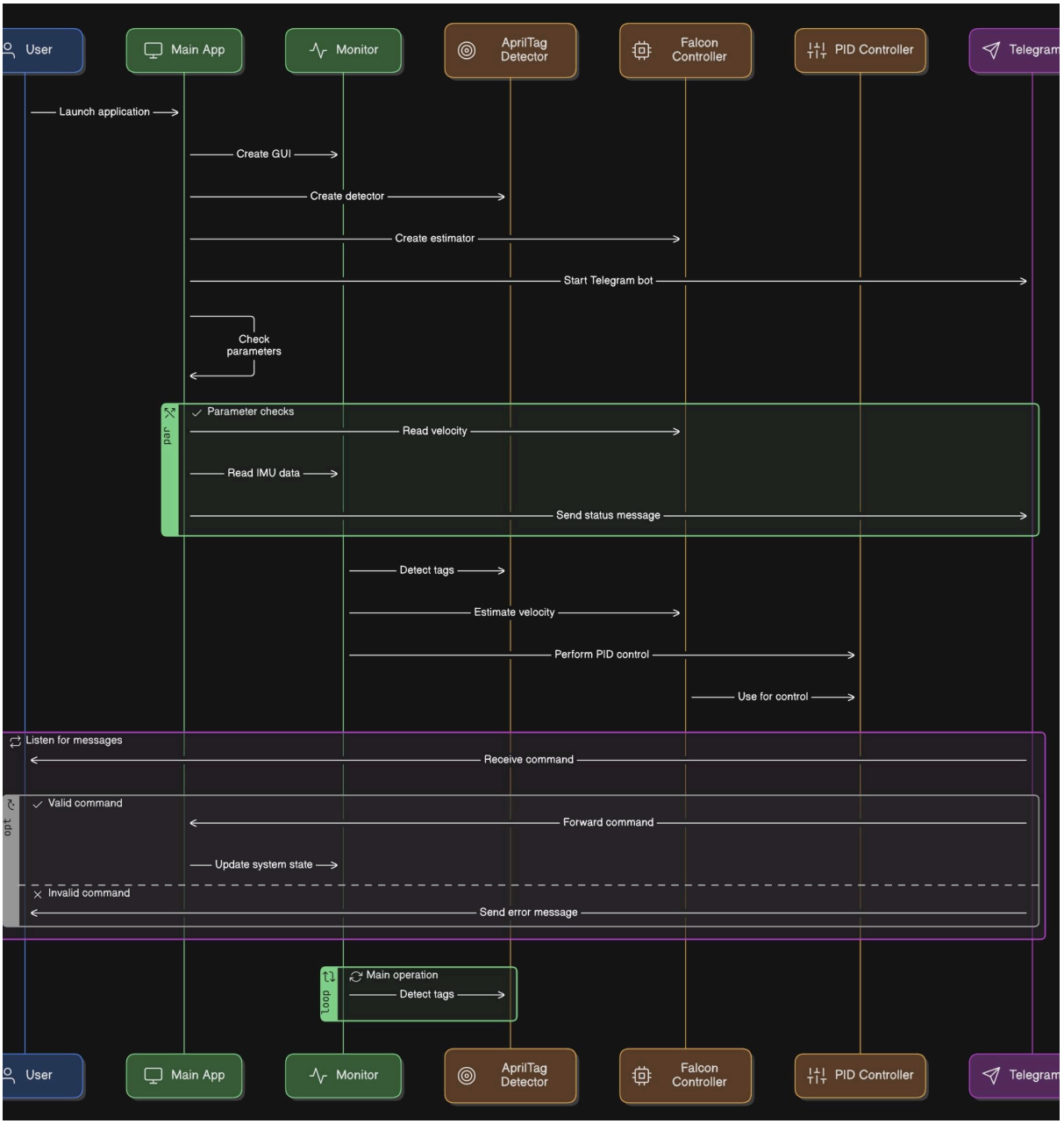
Code snippet of the main function to host the Telegram bot

As the threads for the detector/estimator and the Telegram backend are handled separately within the same runtime, we used Python's built-in queue library to send the information from one to another, allowing the Telegram backend to receive the telemetry data as soon as possible. Helper functions were made to sort the incoming data to ensure that the bot does not spam the user with unnecessary information. Using a first-in-first-out queue system, the detector and estimator parses sorted, relevant data into the queue, and the Telegram backend will take the data from the queue and parse it as a Telegram message to the user and other subscribers to the bot.

5. Software Architecture Diagram

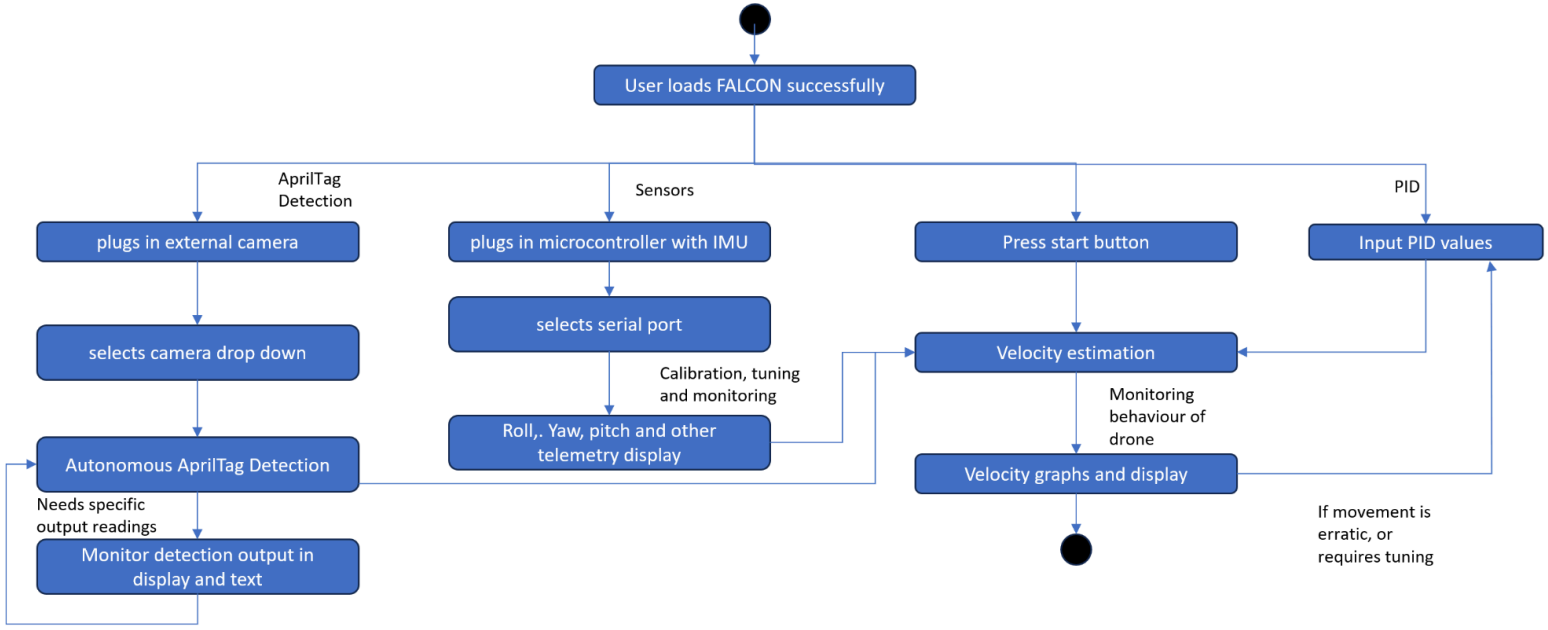
1. FALCON-hardware Architecture Diagram

The following skeletal text-based software architecture diagram shows how the FALCON system functions through the flow of data and tech stack:



2. User Activity Diagram

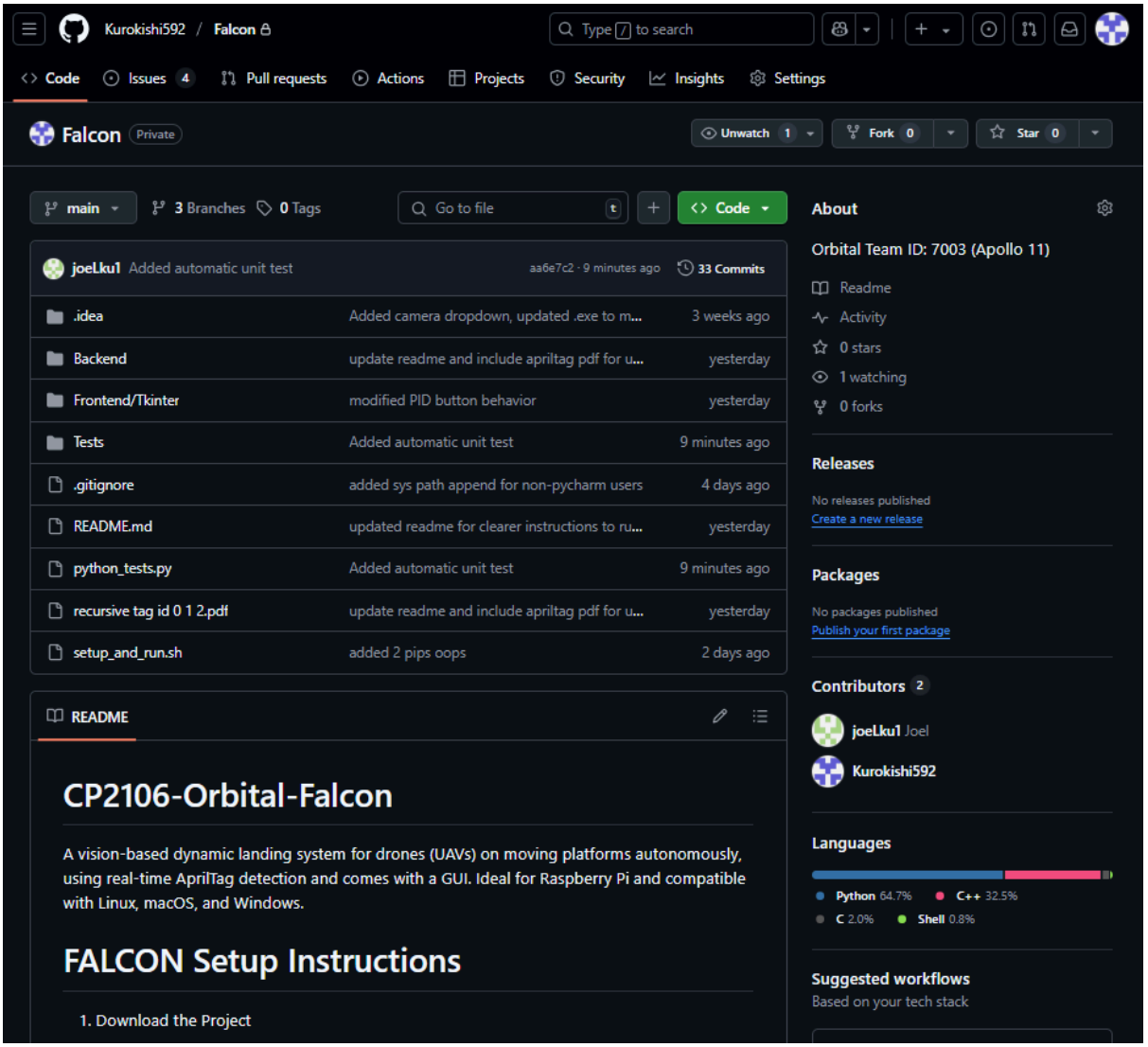
This user activity diagram outlines the typical user flow for an operator using FALCON's GUI to run the backend algorithms.


User loads FALCON successfully
AprilTag Detection
Sensors
PID
plugs in external camera
plugs in microcontroller with IMU
Press start button
Input PID values
selects camera drop down
selects serial port
Velocity estimation
Calibration, tuning and monitoring
Monitoring behaviour of drone
Autonomous AprilTag Detection
Roll,. Yaw, pitch and other telemetry display
Velocity graphs and display
Needs specific output readings
Monitor detection output in display and text
If movement is erratic, or requires tuning

6. Software Engineering Principles

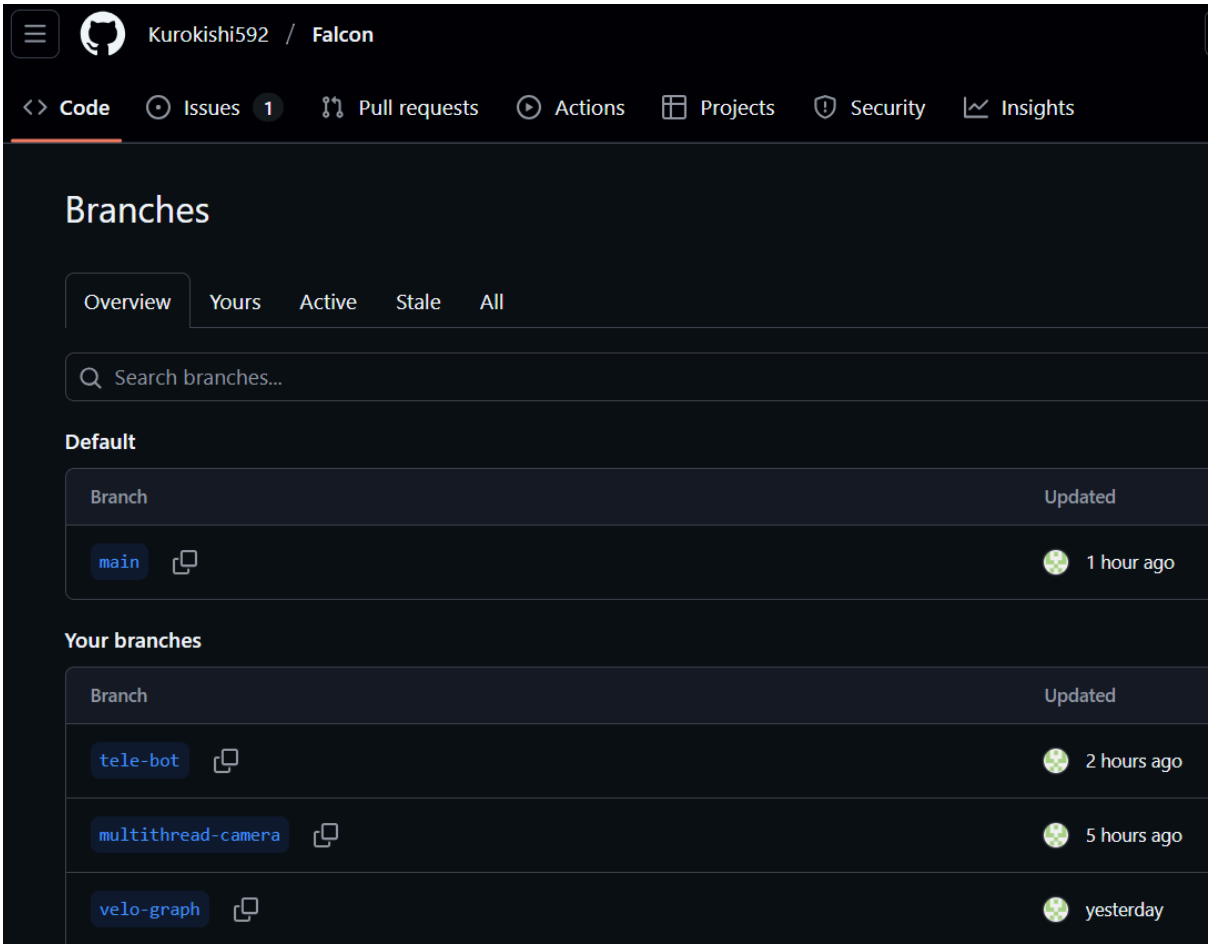
1. Version Control and Collaborative Development

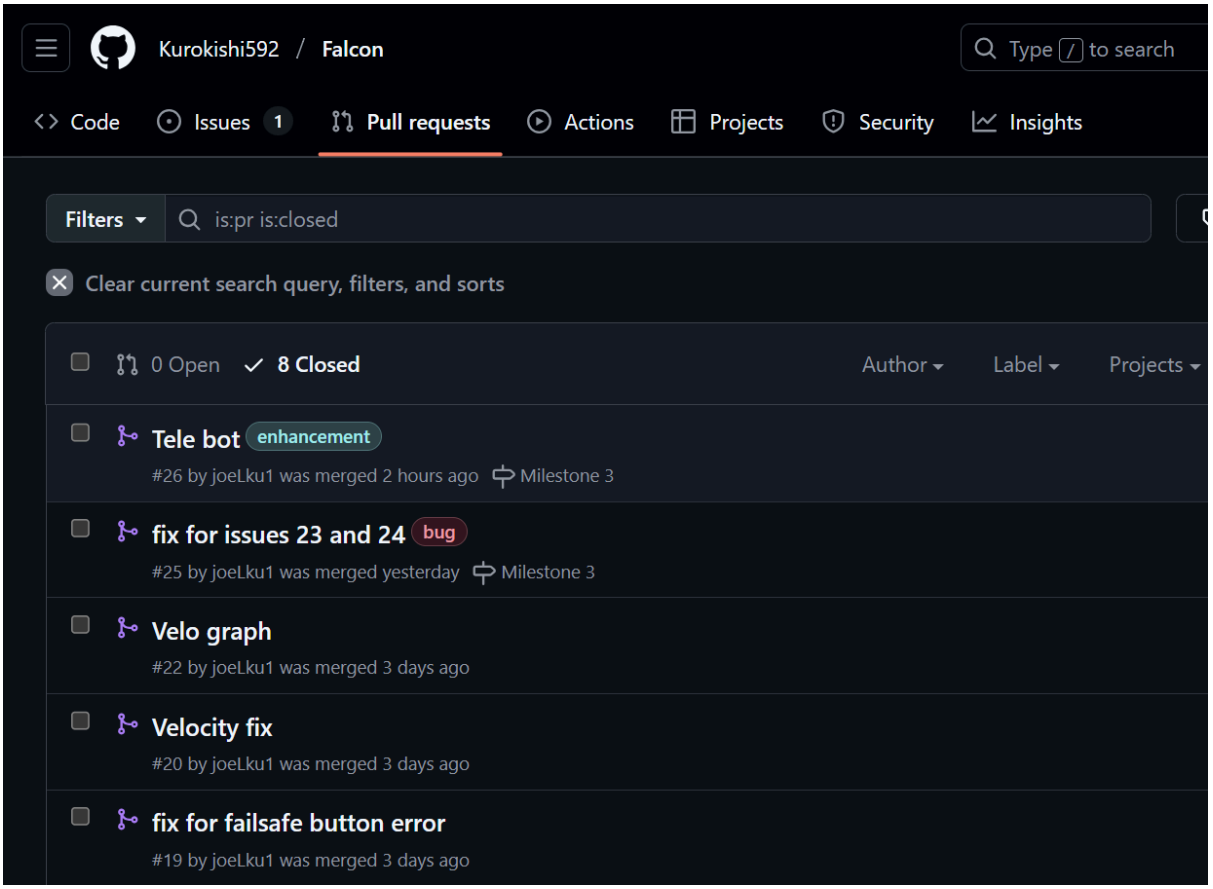
To support effective team collaboration and ensure code integrity throughout the development lifecycle, we adopted robust version control practices using Git and GitHub. All source code was maintained in a shared GitHub repository, providing a centralised and transparent platform for code management and team coordination. We used features like Branches, Pull Requests (PR), issues, milestone labels, tags, assignees for our project.



FALCON's Github Repository and its Snippet of the README

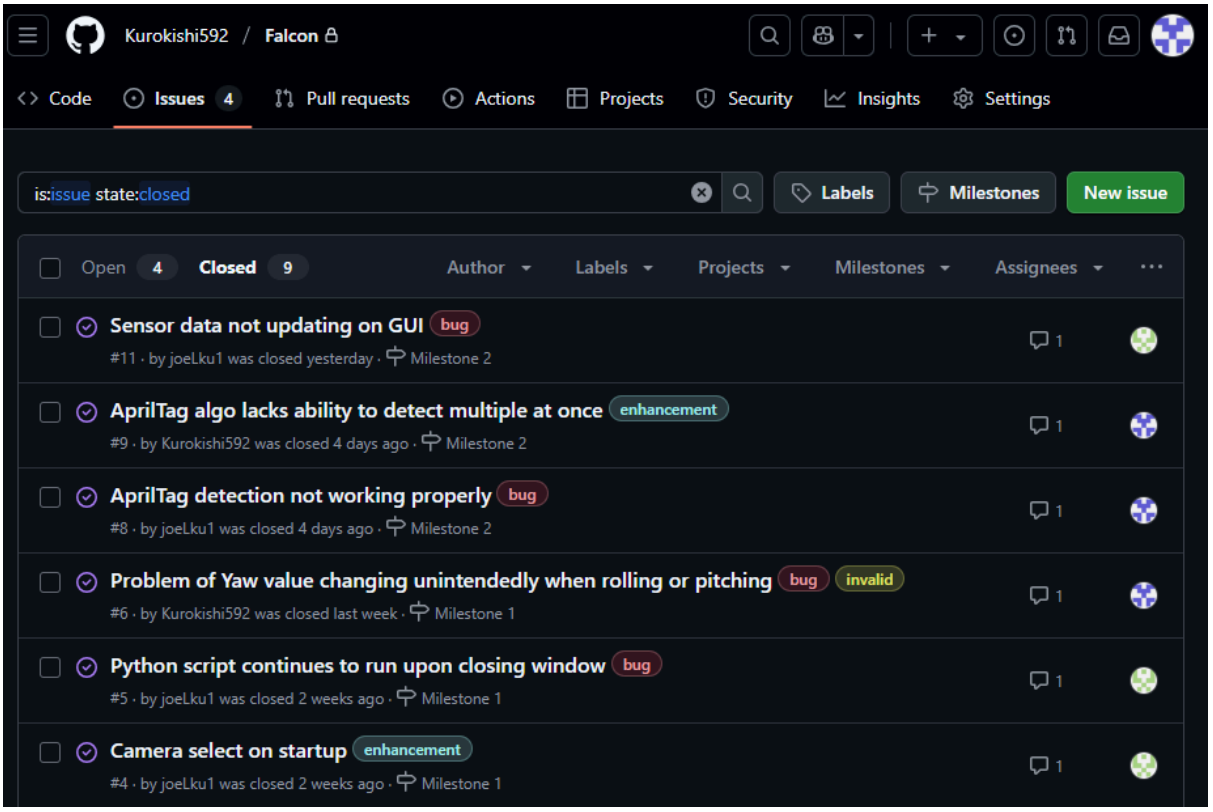
We followed a **branch-based development** model where each team member worked on isolated local branches derived from the main branch (`main`). This allowed us to implement and test features independently, without interfering with the production-ready code. Integration into the main branch was carried out through **Pull Requests**, which were subject to peer review to ensure code quality, maintainability, and correctness. This approach also enforced a separation of concerns and facilitated modular development, which we will discuss later.





Our use of branches - main and various working branches - and pull requests

During development, to organize our workflow and track progress systematically, we used **GitHub Issues** for task management and bug tracking, assigning each issue to the relevant **assignees** with appropriate **labels** and **milestone tags**, such as identification by milestones and bug or enhancement tags. This made it clear what kind of changes are needed and its specific purposes. Using issues also enabled us to link code changes directly to problems that we have discovered and maintain traceability between requirements and implementation.

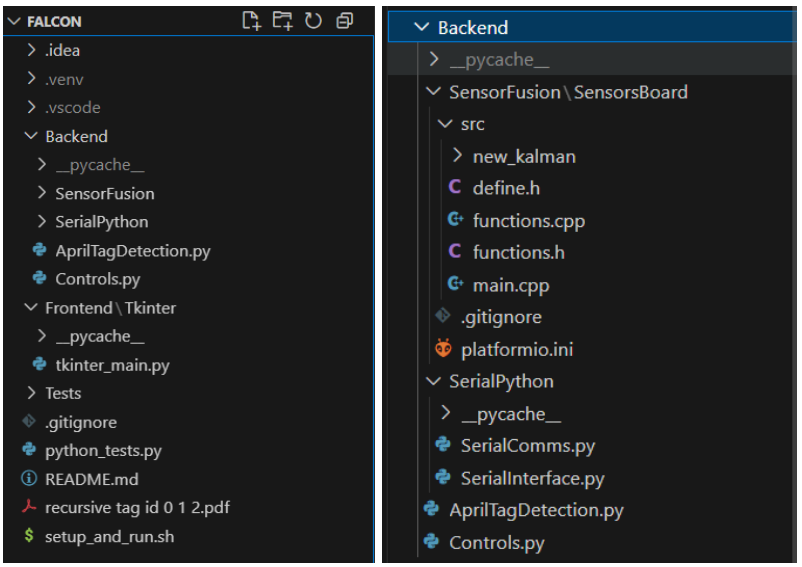


Snippet of our use of issues, tags, labels and assignees

2. Modular Design and Separation of Concerns (SoC)

We employed a modular design approach by architecturing FALCON as a collection of distinct modules, each responsible for a well-defined subset of functionality within the broader FALCON algorithm.

The application was divided into frontend and backend components, with the Frontend implemented in Python Tkinter to manage the graphical user interface. It operates independently of the backend logic, using functions to accept incoming data from the various Backend modules, like the IMU sensor data in `SensorFusion\SensorsBoard\src`, pose estimation and velocity estimation in `Controls.py`, and AprilTag detection results in `AprilTagDetection.py`. These data are sent over serial communication USB UART, which itself is another module within `SerialPython.`



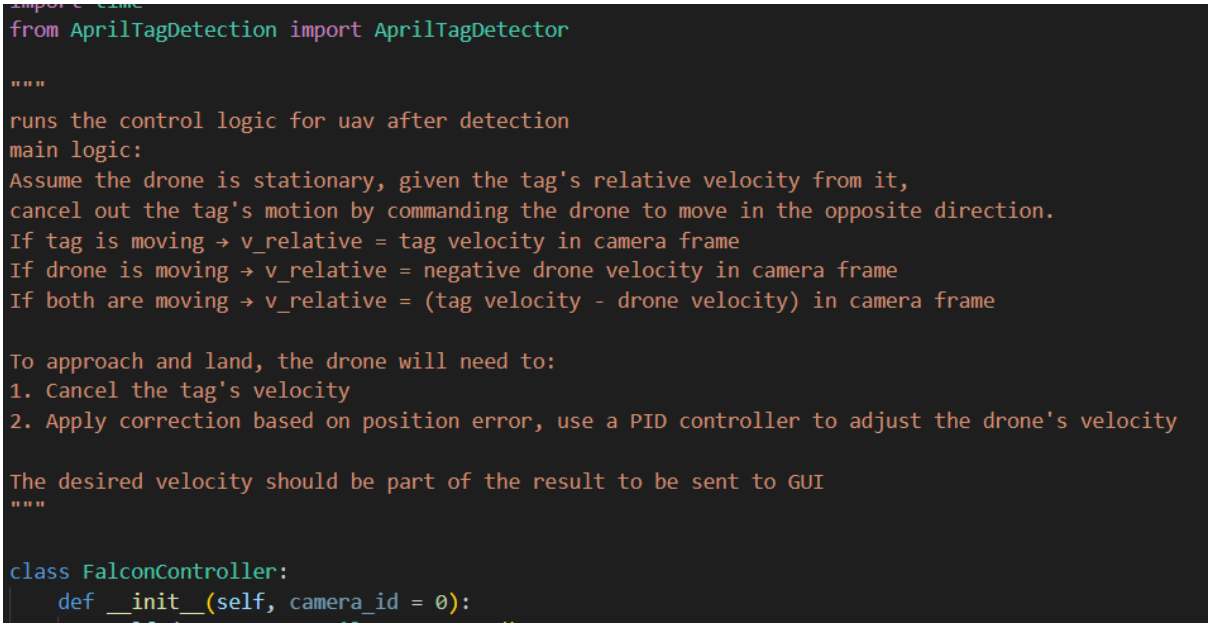
Separation of concerns and modularisation of the various modules

Each module can be run independently and was encapsulated within its own module or class, adhering to the principle of separation of concerns. This architectural decision ensures that the logic for each is self-contained and minimally dependent on other parts of the system. As a result, individual modules can be tested, debugged and extended in isolation without risking unintended side effects in other components. This modular architecture not only simplifies the development process but also enables parallel development among team members. Furthermore, it enhances the reusability of components and supports more robust error handling and future scalability of the system.

3. Documentation:

We were consistent with comprehensive documentation during the FALCON software development process. We maintained internal and external documentation to ensure clarity and readability in code.

At the code level, we adopted Python docstring conventions and inline comments to describe the purpose, inputs, outputs, and behavior of functions, classes, and critical logic blocks. This documentation style made the codebase more readable and self-explanatory, particularly for modules with complex algorithms such as AprilTag detection, velocity estimation, and UI data binding.



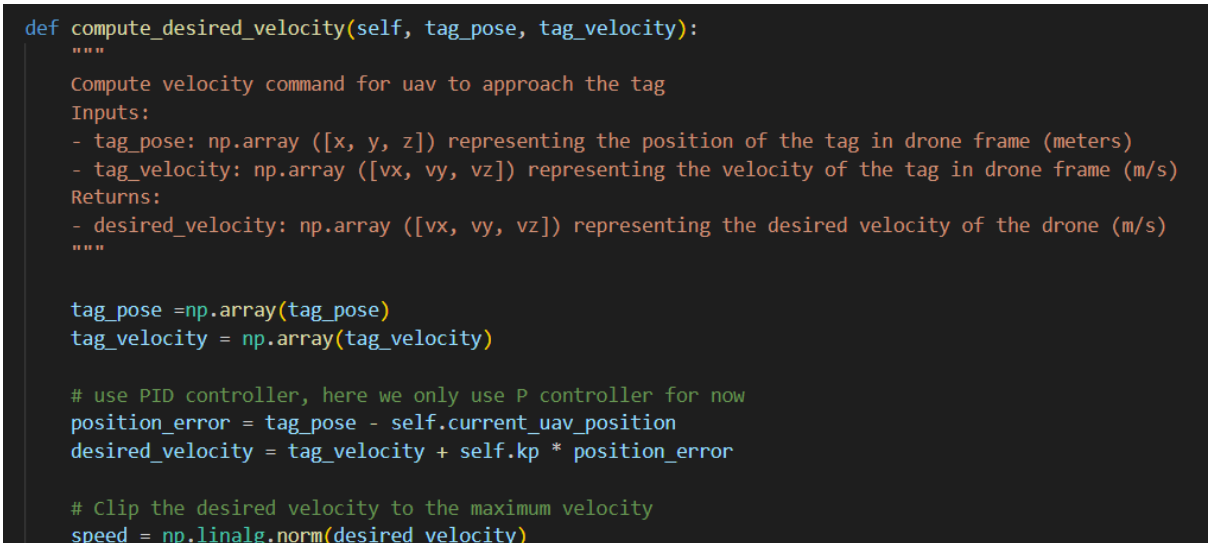
```
from AprilTagDetection import AprilTagDetector

"""
runs the control logic for uav after detection
main logic:
Assume the drone is stationary, given the tag's relative velocity from it,
cancel out the tag's motion by commanding the drone to move in the opposite direction.
If tag is moving → v_relative = tag velocity in camera frame
If drone is moving → v_relative = negative drone velocity in camera frame
If both are moving → v_relative = (tag velocity - drone velocity) in camera frame

To approach and land, the drone will need to:
1. Cancel the tag's velocity
2. Apply correction based on position error, use a PID controller to adjust the drone's velocity

The desired velocity should be part of the result to be sent to GUI
"""

class FalconController:
    def __init__(self, camera_id = 0):
```

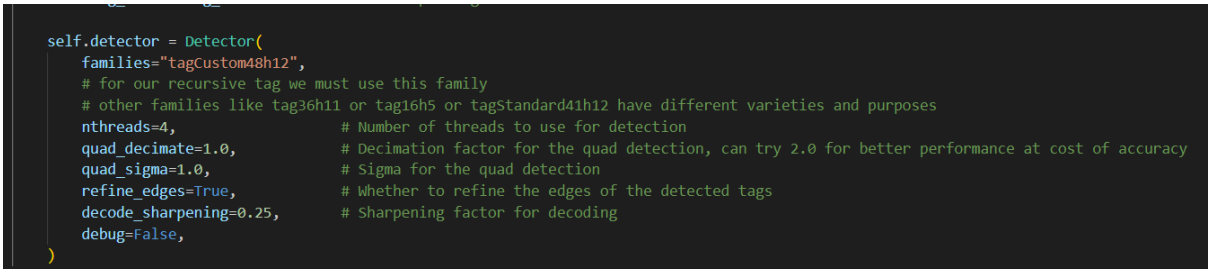


```
def compute_desired_velocity(self, tag_pose, tag_velocity):
    """
    Compute velocity command for uav to approach the tag
    Inputs:
    - tag_pose: np.array ([x, y, z]) representing the position of the tag in drone frame (meters)
    - tag_velocity: np.array ([vx, vy, vz]) representing the velocity of the tag in drone frame (m/s)
    Returns:
    - desired_velocity: np.array ([vx, vy, vz]) representing the desired velocity of the drone (m/s)
    """

    tag_pose =np.array(tag_pose)
    tag_velocity = np.array(tag_velocity)

    # use PID controller, here we only use P controller for now
    position_error = tag_pose - self.current_uav_position
    desired_velocity = tag_velocity + self.kp * position_error

    # Clip the desired velocity to the maximum velocity
    speed = np.linalg.norm(desired_velocity)
```

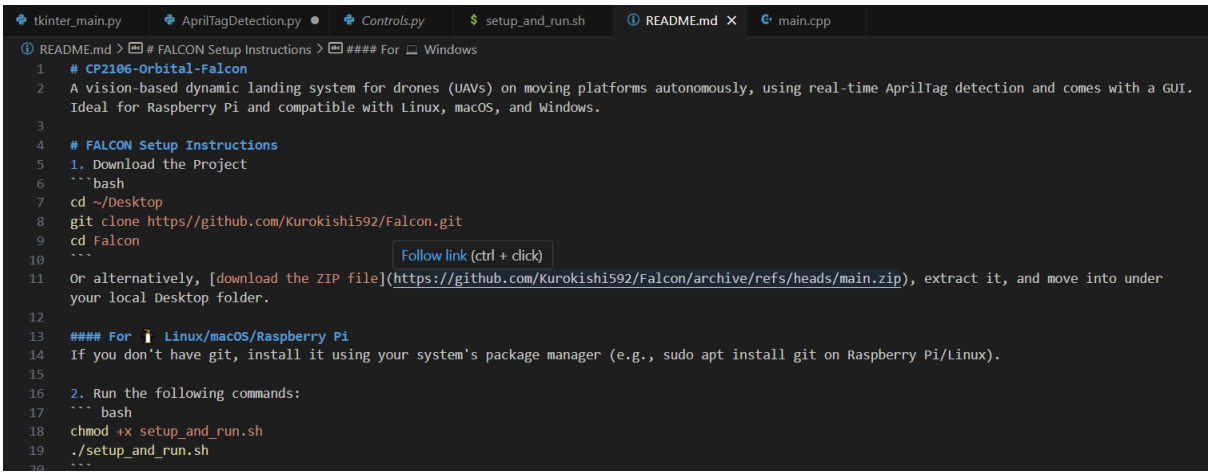


```
self.detector = Detector(
    families="tagCustom48h12",
    # for our recursive tag we must use this family
    # other families like tag36h11 or tag16h5 or tagStandard41h12 have different varieties and purposes
    nthreads=4,                 # Number of threads to use for detection
    quad_decimate=1.0,          # Decimation factor for the quad detection, can try 2.0 for better performance at cost of accuracy
    quad_sigma=1.0,             # Sigma for the quad detection
    refine_edges=True,          # Whether to refine the edges of the detected tags
    decode_sharpening=0.25,     # Sharpening factor for decoding
    debug=False,
)
```

Examples of FALCON's documentations on the code level

Beyond inline documentation, we created a README markdown file within the GitHub repository to provide high-level explanations of the system architecture, setup instructions, module dependencies, and expected behavior. These documents

served as a guide for both developers and users to understand how to install, configure, and run the system.

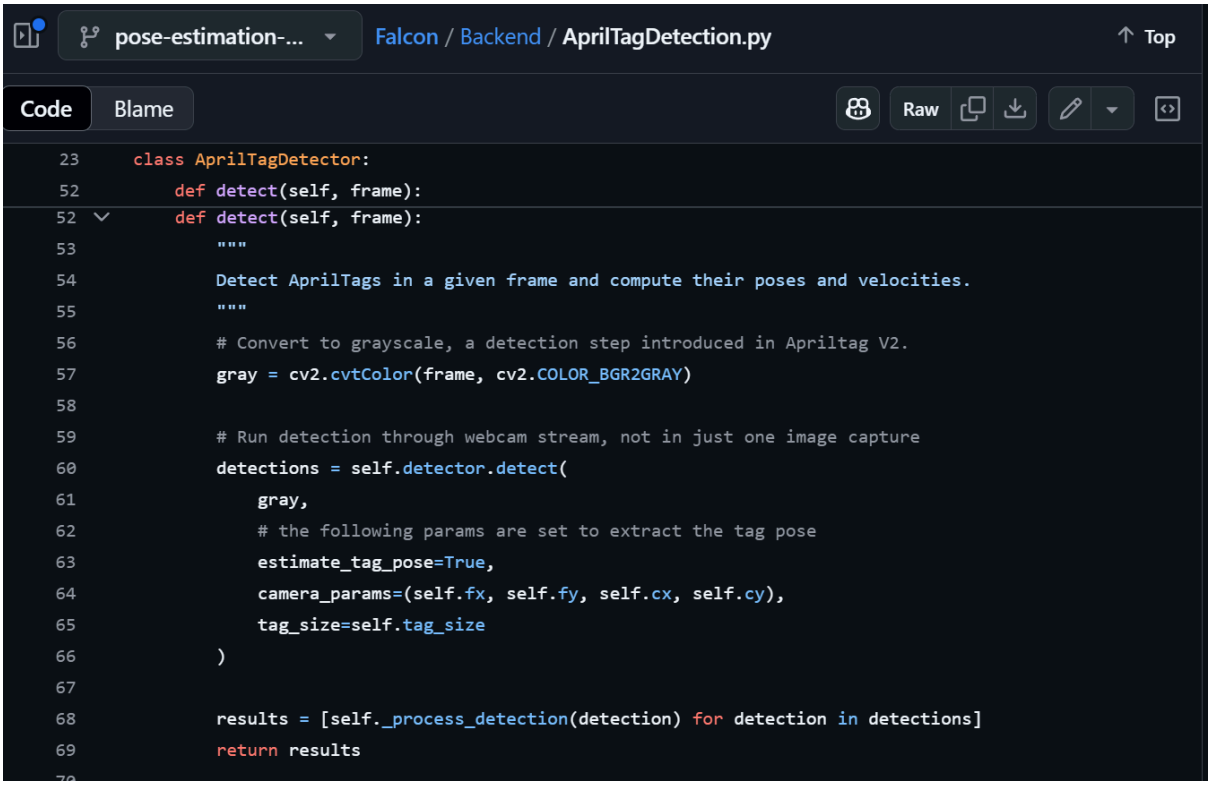


FALCON's README markdown file that is reflected on the Github repository

4. OOP Principles

Core object-oriented programming (OOP) principles (in particular encapsulation, abstraction and not so much of inheritance and polymorphism) were applied to structure the software in a modular, scalable and maintainable way.

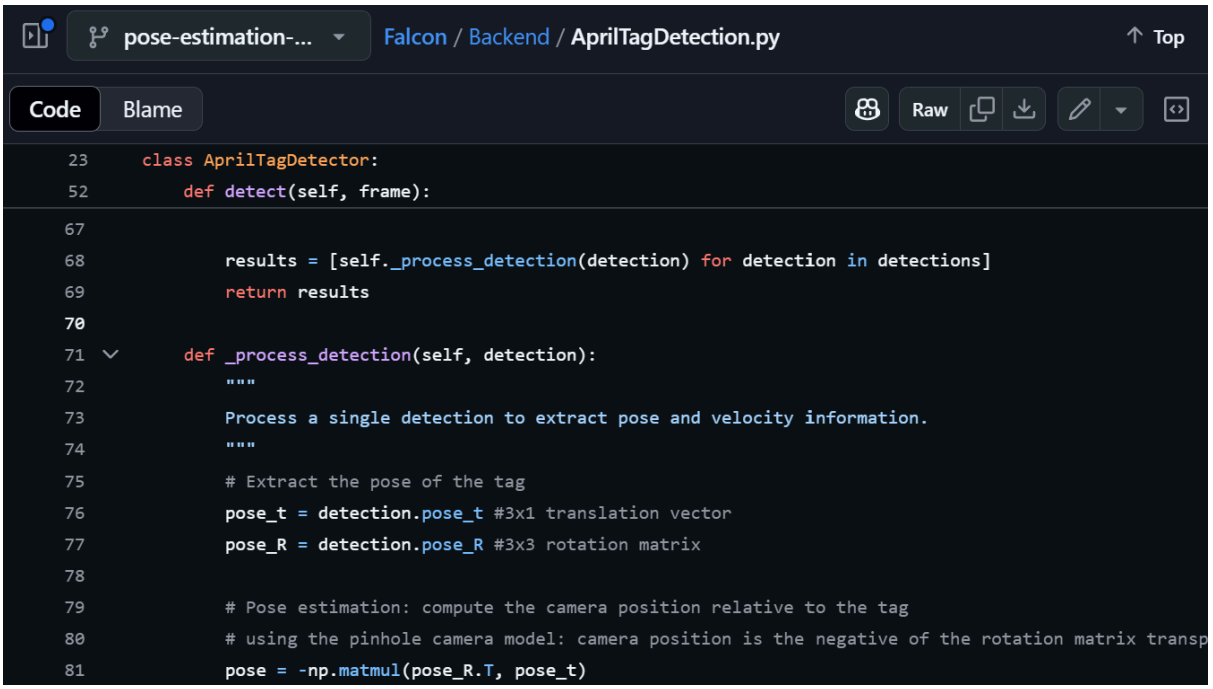
Encapsulation was achieved by grouping related data and functionality into classes such as `AprilTagDetector`, `FalconController`, Tkinter GUI `Monitor` class, and Telegram bot, all having their own properties and methods. Instances were then called when said data and functions are needed. These classes encapsulate all the logic such that we only interact with the exposed clean interface for other parts of the system. For example, the `AprilTagDetector` class contains the logic and result parameters of the entire AprilTag detection, and an instance is simply called, such as in `Control.py` whenever we want to perform an Apriltag detection



Implementation of class and use of the detect method in other parts of code

Abstraction was implemented by defining high-level operations in well-named methods (e.g.`compute_desired_velocity()`, `detect()`) while hiding lower-level complexities such as in private methods like `_process_detection` which handles the matrix math and logic behind apriltag detections. We can thus allow other parts of the codebase to use these functionalities without needing to understand how they work internally while also reducing the impacts to other parts

of the entire program (deal with spaghetti code problems) when amending its properties or methods.



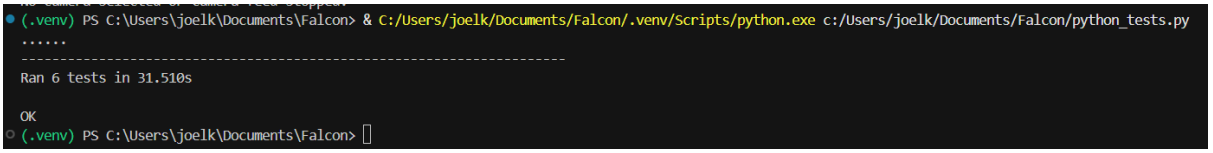
Use of private method _process_detection

7. Software Testing

1. Automated Script Testing (Unit Test)

FALCON uses Python's built-in unittest unit testing framework to test the various classes and associated functions that make up FALCON's backend code. For some of the test cases, we made customized images to ensure that detection and velocity estimations are correct. For example, we warped the AprilTag using image editing software to simulate the drone approaching from an extreme acute angle, which gives us the confidence that the detector class can function even with extreme image warping.

To run the automated unit testing, we can just run `python_tests.py` and check that all the tests pass. While there are only 6 tests, each test has multiple conditions that are checked before the test passes.



Below shows the table of tests that are performed automatically:

Test	Description of input	Conditions to check
#1 3 recursive AprilTags detection, straight and upright	Basic, unedited 3 recursive AprilTags (ID 0, 1, 2) test_3_recursive.png	- 3 tags are detected in the image - First detected tag ID must be 0 - Second detected tag ID must be 1 - Third detected tag ID must be 2
#2 No AprilTag detection	A blank white image with nothing inside test_0_blank.png	- No AprilTags are detected in the image
#3 3 recursive	3 recursive AprilTags (ID 4, 5, 6), rotated 45 degrees to the left	- 2 tags are detected in the image

AprilTags detection, rotated	test_3_rotated.png (due to the rotation, the innermost tag (ID 6) cannot be detected by the detector	- First detected tag ID must be 4 - Second detected tag ID must be 5
#4 3 recursive AprilTags detection, warped	3 recursive AprilTags (ID 0, 1, 2), warped 45 degrees on the X and Y axes test_3_warped.png (due to the warping, the innermost tag (ID 2) cannot be detected by the detector)	- 2 tags are detected in the image - First detected tag ID must be 0 - Second detected tag ID must be 1
#5 3 recursive AprilTags velocity estimation	Base image contains 3 recursive AprilTags (ID 0, 1, 2) test_3_original.png Shifted image shifts the original AprilTags slightly to the left of the original test_3_shifted.png The velocity estimator is tested twice, from original to shifted and then back to original (the speed is the same, with different direction)	- First x velocity should be -0.02 - First y velocity should be 0 - First z velocity should be 0 - Second x velocity should be 0.02 - Second y velocity should be 0 - Second z velocity should be 0
#6 3 recursive AprilTags velocity estimation	Similar to test #5, but using the same base image twice containing 3 recursive AprilTags (ID 0, 1, 2) test_3_original.png	- X velocity should be 0 - Y velocity should be 0 - Z velocity should be 0
#7 PID calculation	Test if the PID controller works with sample values	- First PID output should be 2.6 - Second PID output should be 0.2

2. User Testing

Since FALCON is a software meant for users to test their hardware integration, calibration and personalised parameters, it is crucial for users to perform manual testing, which we have described below. These users were given a working prototype of the software, along with all the necessary hardware required for FALCON to run (external camera, custom sensor board).

General (ease of use and clarity of instructions):

Question: How easy is it to set up FALCON, given the instructions within the GitHub README?

- Most users were able to follow the steps, and some commented that the steps were clear and concise

Question: Does the GUI make sense without further instructions?

- Most users were able to use the FALCON GUI without explicit instructions
- Some commented that the font size could be increased, as there is still space in the GUI

Reliability (detection, estimation):

Question: How do you find the AprilTag detection, and how reliable is it?

- Most users rated the detection at 4/5, while a minority rated it at 3/5
- Users who gave a lower score remarked that the camera FPS was too low, and that excessive shaking, which could occur during landing due to factors like wind and increased motor thrust, caused the camera to lose the AprilTag detection - but conceded that this issue could be fixed with better hardware (camera)

Question: Based on your experience with FALCON, how would you rate the estimation reliability?

- Users did not have the tools to properly measure the estimation to check the accuracy, but placed the estimation reliability at 4/5 for a lightweight estimation algorithm

Telegram bot functionality:

Question: How is the functionality of the bot now, and what other features would you add to the bot?

- Users felt that the addition of a Telegram bot was extremely useful for off-site notifications, and due to its open-source nature, is readily available for future expansion and addition of other features
- One suggestion that was given was to reduce the words in the messages, further simplifying the messages would make it even easier to see what went wrong at a glance

3. System/E2E Testing

App launch and startup of backend processes			
Goal	Approach	Expected outcomes	Result
User is able to launch the GUI	Run `main.py`	FALCON's GUI is correctly displayed and the user can interact with it	Pass
Telegram bot backend started successfully	Run `main.py`	Telegram bot responds to user's `/start` command and displays the user's Telegram name Terminal that runs `main.py` shows "Bot started"	Pass
AprilTag detector started successfully	Run `main.py`	Terminal that runs `main.py` shows "AprilTag detector set"	Pass
Velocity estimator started successfully	Run `main.py`	Terminal that runs `main.py` shows "Velocity estimator set"	Pass

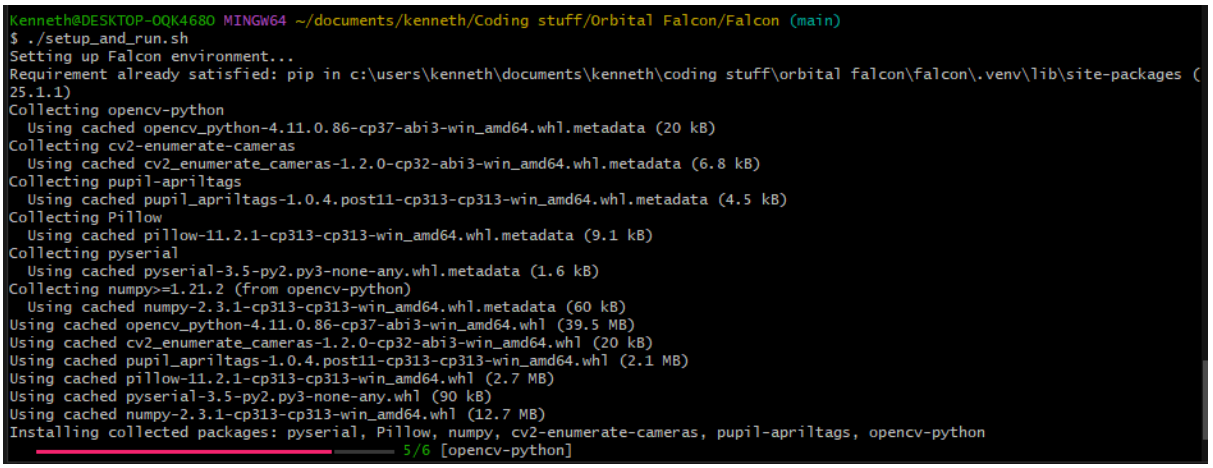
Camera startup and AprilTag detection/estimation			
List of cameras is correctly displayed on the dropdown	Click on the dropdown and check list of cameras on the system	All expected cameras (based on Device Manager -> Cameras) are detected and shown in the dropdown list	Pass
Selecting a camera opens the camera feed on the camera view	Select a camera from the dropdown list and pressing the confirm button	Live camera feed from the camera is shown on the “Camera View” subframe within the GUI	Pass
Refresh button works	Plug in a camera device after the GUI is launched, and use the camera refresh button	New camera device is shown in the dropdown list	Pass
Able to disable the camera feed when not in use	Selecting the “-1: Off” turns off the camera feed	Camera feed is removed, and any detection/velocity info (if any) stops updating	Pass
AprilTag detection works	Open the camera feed and point the camera towards the provided test AprilTag	AprilTag detection overlay is shown on the camera feed, “Detected AprilTag message” is updated with the tag ID(s).	Pass
Velocity estimation works	Move the camera or the AprilTag around slowly	Velocity data (text) should update, and graph should show the past results of all 3 velocity data (red, green and blue lines)	Pass
PID testing			
Invalid inputs to PID	Entering non-float values into PID boxes	Error message should show up beside the “Upload PID Params” button	Pass
PID works	Select a camera and point camera at the test AprilTag Press the start button	When camera is not moving, the predicted velocity should tend to the value set in the setpoint (default 0.5)	Pass

Failsafe works	Press the failsafe button after the above test	The predicted velocity should be reset to 0	Pass
Sensor communication			
List of available serial ports works	Press the dropdown for choosing serial port	List of all serial ports (if any) should be shown (based on Device Manager -> Ports (COM & LPT)	Pass
Refresh button works	Plug in the MCU after the FALCON GUI has been initialized, then press the refresh button	A new entry (the MCU) on the dropdown list should appear	Pass
Confirm button works	Select the correct MCU and press the confirm button	Sensor data should begin updating, following the motion of the sensor board	Pass
Telegram integration			
Telegram notifications are sent when values exceed a certain value	Select the camera, point camera towards an AprilTag, and send the `/start` command to the Telegram bot to initialize, then move the camera around semi-vigorously to exceed velocity setpoints	Telegram notification to the user should be sent when the speed exceeds the threshold	Pass

8. Roadmap

Before Milestone 1	Got the rough outline of the project, Tkinter GUI, finished sensor fusion
Before Milestone 2	Pipe data from sensor and camera into GUI, complete AprilTag detection, motion approximation, and main algorithm, docker

By Milestone 3	Complete all additional features and clean up.



9. Future Considerations

As we only had 1 summer to complete this entire project, we were not able to implement all of our features fully, and some implementations were simplified slightly.

1. Improvements to our velocity estimation algorithm

More development work can be done to further improve our velocity estimation. The detector class returns a multitude of information that is related to the detected AprilTag, all of which can be used to determine the position of the AprilTag with relation to the camera. More sophisticated algorithms can be implemented to calculate the relative position, skew and rotation of the AprilTag in relation to the camera, using all the available information from the detector class.

2. Additional features to the GUI

The GUI currently shows only the necessary information for the operator to use FALCON. However, more information and features could be added to the GUI. For example, buttons could be added to start/stop the Telegram bot feature, and the graph on the right can be modified to graph the various sensor data, like roll/pitch/yaw over time, in addition to just the velocity.

Additionally, from our experience with other robot control software, we have realized the usefulness of logging data. In our case, logging could come in the form of recording all the sensor and detection information, like the AprilTag detections, estimated velocity and sensor data. Logging would provide the operator an easy way to record all the data during operation, and would give FALCON users a way to check and refer back to all the information that was captured.

3. Performance improvements

FALCON occasionally experiences lag spikes when several tasks run simultaneously. More subtasks can be added to the Python multithread operation, which would significantly improve performance by allowing concurrent processing of the tasks.

4. Additional sensors

FALCON's GUI has added space for camera FPS and battery voltage, which can be monitored with higher end hardware, and modifications to the sensor board. This would provide more telemetry data for the drone operator.