Mechatronics II Report – Computer Vision Subsystem

Written by Alexander Evans Moncloa

Introduction

Objective and Background of the Project

This project consisted of a Martian rover, made for the IDESA, which had the intended purpose of autonomously reaching all red checkpoints on Martian terrain in an efficient manner. The rover had to avoid blue checkpoints, whilst avoiding collisions with moving green obstacles, to then finally arrive at a manually-entered destination.

System Architecture Overview

For full autonomous functionality, the rover needed functional hardware and software. The hardware compromised of a Raspberry Pi 3, a Pi hat with motor drivers and voltage dividers, an 8V battery, 3 motors with 3 encoders, 3D printed omnidirectional wheels/chassis and a clear plastic sphere. The software compromised of Simulink-compiled C for low-level control of the rover using MQTT signals and on-board PID control algorithms, along with Python with OpenCV for high-level control utilising a webcam, ArUco codes and coloured items for accurate positional feedback, object detection and path-planning.

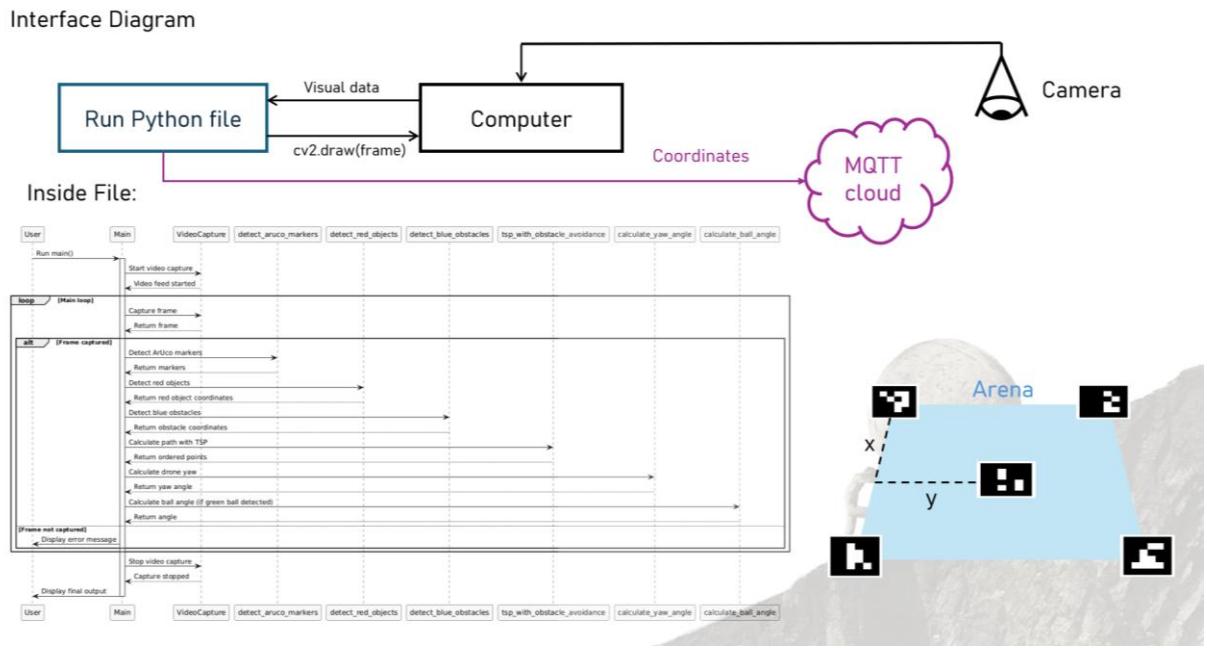

Figure 1: Block diagram and sequence diagram of setup, data transfer and code contents.

Focus of the Report

This report focuses on the high-level control, including image processing, path-planning and data transmission. The report consists of the subsystem design, testing, integration and the collaborative problem-solving approaches required to make the subsystem functional, reliable and repeatable.

Design Specifications

Index	Attribute	Verification Activity	Connection to the System
1	ArUco Marker Detection	Test detection accuracy under different lighting conditions.	Forms the basis for homography transformation, providing reliable position and orientation data to navigate the terrain accurately.
2	Homography Transformation	Validate real-world coordinate mapping by comparing known marker positions with transformed outputs.	Ensures all detected objects are mapped to real-world coordinates, allowing accurate path planning and obstacle avoidance.
3	Colour Detection (Red: Static Target)	Test detection of red markers in varied lighting and distances.	Identifies target points for the rover to travel towards, feeding into the path-planning algorithm for optimal target sequencing.
4	Colour Detection (Blue: Static Obstacle)	Verify blue marker detection accuracy, ensuring no false positives.	Allows the rover to avoid areas marked as obstacles, integrating with the path-planning algorithm to determine a safe route.
5	Colour Detection (Green: Moving Obstacle)	Test detection of moving green objects in real time and ensure detection accuracy in variable lighting conditions.	Used to dynamically avoid moving obstacles, modifying the rover's trajectory in response to real-time changes in the environment.
6	Travelling Salesman Algorithm (TSP) for Path Order	Verify that path ordering successfully avoids paths that intersect blue obstacles.	Reorders target points to avoid collisions, optimising the travel route and ensuring safe passage between red markers.
7	Angle Detection for Moving Obstacles	Compare previous frame data with current frame to validate accurate yaw and angular velocity calculations.	Enables real-time adjustment to avoid moving obstacles by modifying the path to maintain a safe distance.
8	MQTT Broadcasting	Ensure all relevant data (position, orientation, obstacle coordinates) is	Provides a communication link between high-level vision processing and low-level control,

		broadcast at the correct refresh rate.	ensuring synchronised navigation commands.
9	Low-Level Control Integration	Verify correct interpretation of vision data by the C-programmed low-level control module through test runs.	Integrates high-level image processing with motor control, allowing the rover to act on received path and obstacle data in real time.
10	Code Maintainability	Ensure code is modular, readable and has adequate back-ups.	Allows for rapid changes to be made during testing with low-level software, whilst allowing code to be restored to previous version if too many errors arise.

Subsystem Testing

[1-2] ArUco Detection and Homography Calculation

To test for these attributes accurately, the camera positioning was calibrated to optimise the field of view and minimise distortion. The testing involved placing corner ArUco markers at known, fixed positions with 1 metre of separation between each code.

Initial Detection Accuracy: Testing the detection under variable lighting conditions, adjusting lighting to emulate real-world scenarios that could impact detection.

Homography Validation: Once detected, the coordinates of each marker were transformed into real-world coordinates using homography calculations. The transformed outputs were compared to the known positions to assess accuracy. Any deviations were used to recalibrate the camera's positioning and adjust the algorithm.

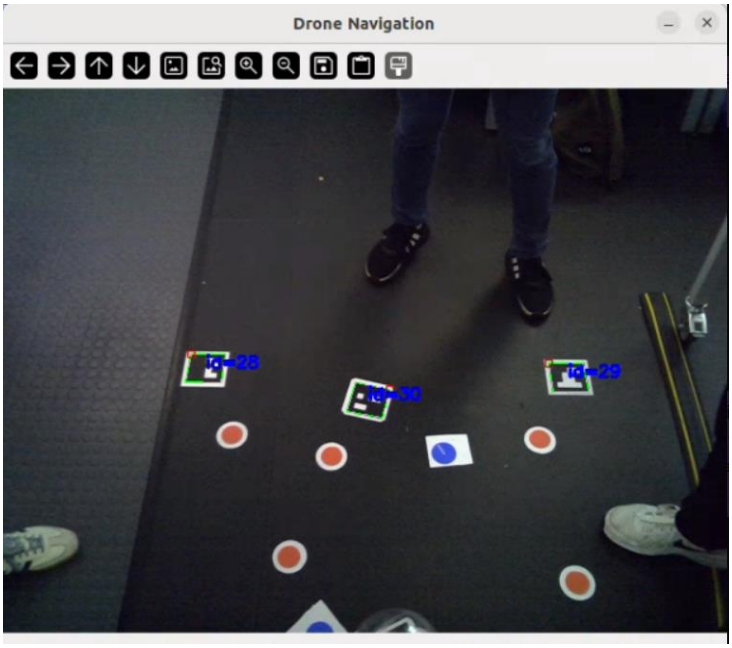

Figure 2: This is an example of the camera calibration procedure. The software written would run for 1 second then pause to give us an estimate of the camera positioning. In this example, half of the arena was not visible, giving us feedback to manually change the positioning.

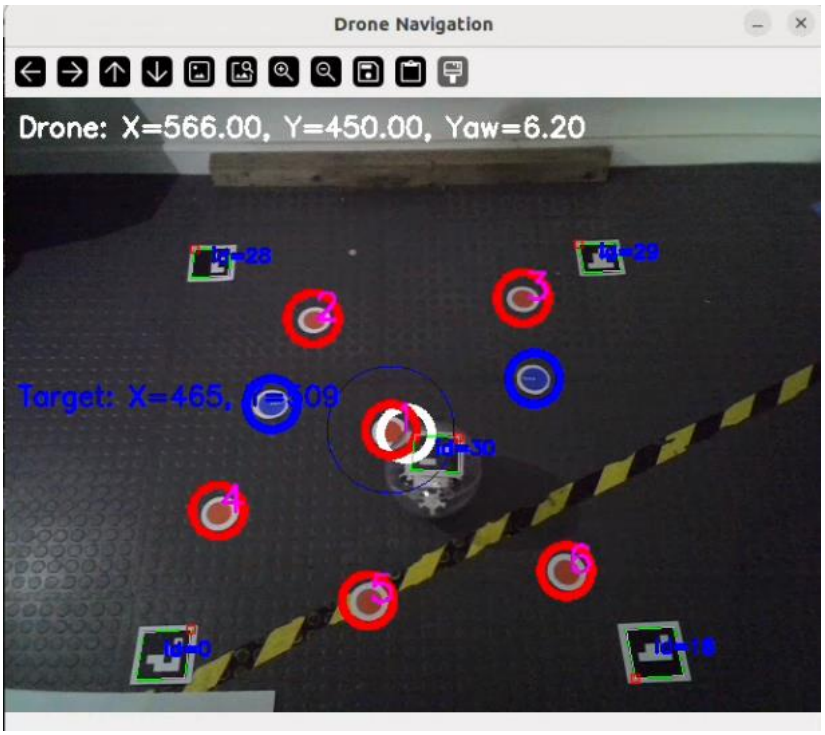

Figure 3: This is an example of us testing the limitations of the homography function. The top corner ArUco markers were placed in a position with high distortion relative to the bottom corners. Despite this, the returned coordinates were highly accurate (within 10mm of the actual location, as measured with a metre ruler).

[3-5] Obstacle Detection

Obstacle detection was split into static (red and blue) and moving (green) obstacle categories, each with unique detection criteria.

Red Static Target Detection: Red markers were tested under various lighting intensities and distances. The success rate was highest using minimal natural and artificial overhead lighting, while using a close-range a flashlight.

Blue Static Obstacle Detection: Similar tests were conducted with blue markers, focusing on minimising false positives. This was crucial since any misinterpretation would create navigational errors. Success rates for different setups were experimentally discovered, causing us to avoid using irregular 3D shapes as obstacles due to inconsistencies in detection.

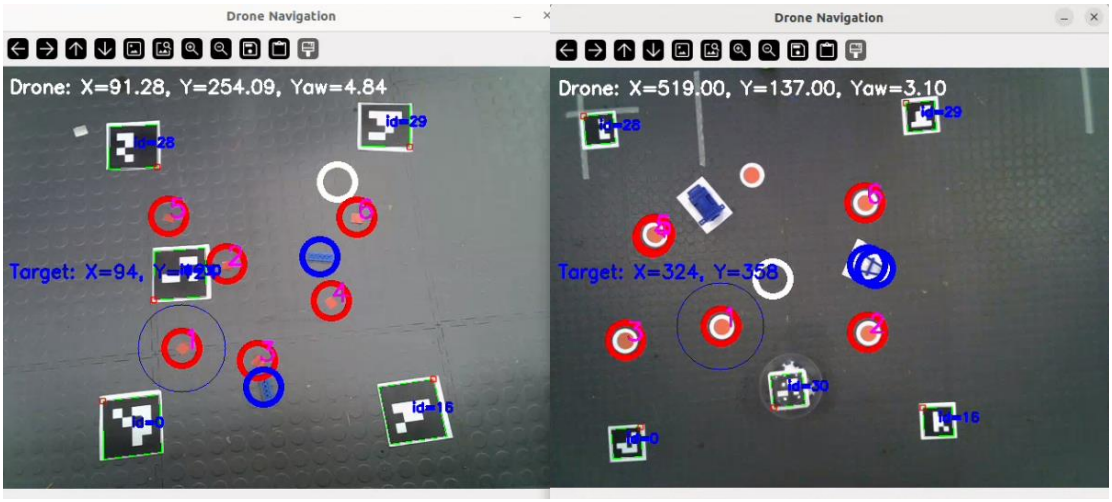

Figure 4: A demonstration of why 3D obstacles were removed from the final test. They brought in too much inaccuracy by sometimes detecting 1 obstacle as 2, ignoring the other obstacle. These images show the progression of detecting 1 continuous 2D obstacle and then experimenting with a 3D obstacle, with an increased error rate.

Green Moving Obstacle Detection: The green ball, representing the moving critical event, was tested in real-time to assess its detection and tracking accuracy under dynamic conditions. The optimal colour detection threshold was determined, leading to reliable detection with bright natural or artificial lighting. This proved to be a mistake, as during testing the green colour was not consistently detected in the very dark lighting conditions required for full functionality.

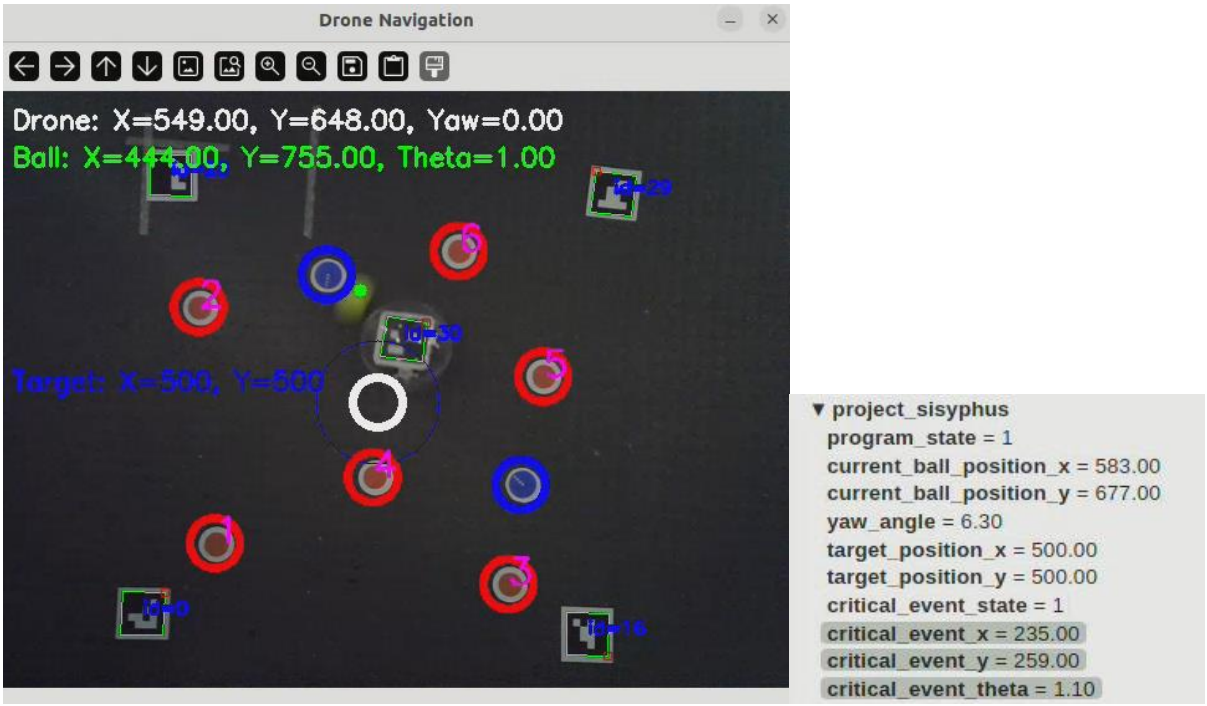

Figure 5: Green ball detecting accurately under dim lighting conditions with live positional data being sent to MQTT broker.

[6] Path Planning

Path planning was tested using a freeze mechanism to validate the obstacle detection and point order, ensuring the rover's path appropriately bypassed obstacles before broadcasting coordinates to MQTT. The Travelling Salesman Algorithm (TSP) was implemented to arrange target points efficiently.

Point Order Check: Each point sequence was verified to prevent any overlap with blue obstacle markers. The freeze mechanism held the order in place, allowing us to inspect and confirm the accuracy of the sequencing visually and programmatically.

Broadcast Validation: Before each test, different arrangements of red points and blue points would be set on the floor to rest the algorithm's accuracy at rerouting to avoid blue markers and start the path near the rover. This lead to an approximate rerouting success rate of 95%.

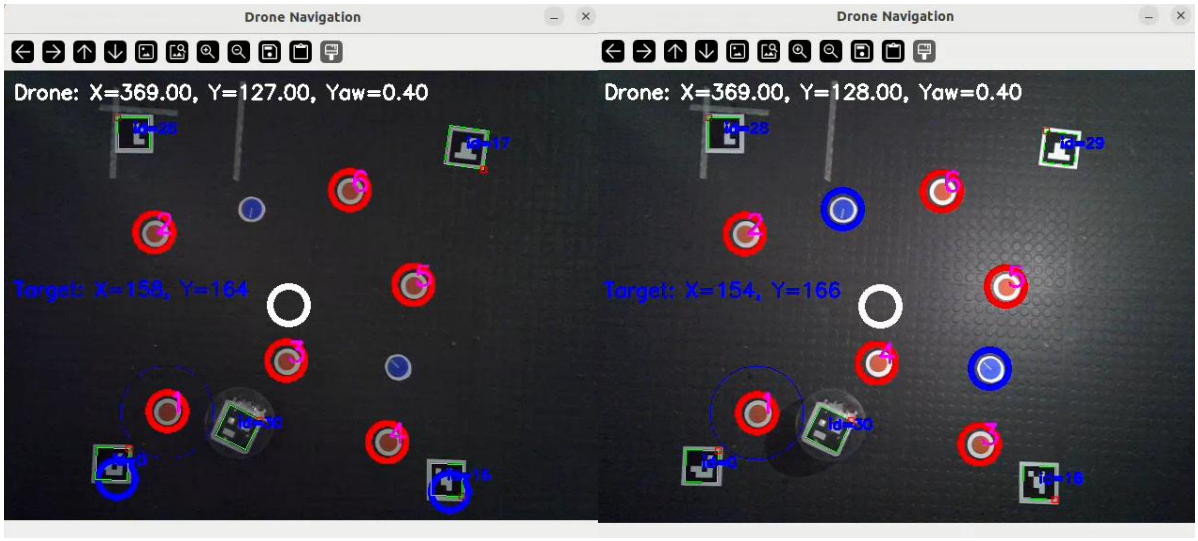

Figure 6: The left image shows the point order check with no handheld flashlight, causing the blue markers to be darker than the detection threshold. Their shade became indistinguishable compared to darker areas of the ground. The right image shows the point order check moments later with the handheld flashlight, clearly detecting the blue obstacles and rerouting the order of red obstacles. 4 to 5 path directly covers a blue obstacle, however when rerouted they don't.

[7] Moving Obstacle Angle Detection

To ensure accurate detection of a moving obstacle's angle, testing was conducted by comparing positional data between frames to verify angular displacement calculations.

Angle and Yaw Accuracy: The angle and yaw detection were cross-referenced with known movement patterns, confirming that the system could reliably predict obstacle trajectories and adjust the rover's path accordingly.

Real-Time Adjustments: The detection algorithm's responsiveness was tested in varying scenarios, confirming the rover's real-time response to moving obstacles by plotting angular changes over time.

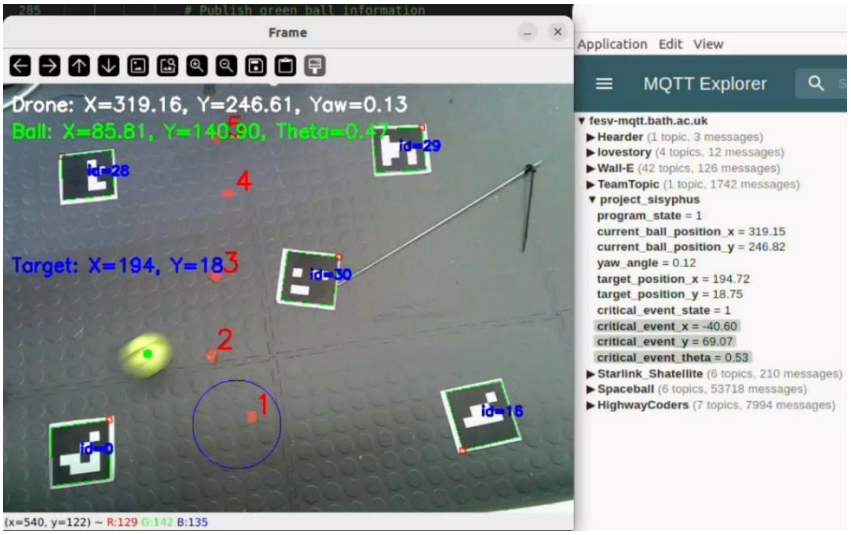

Figure 7: Frame taken during test of live ball coordinate estimation and live angle approximation. MQTT Explorer is opened to check how frequently the signal is being updated to the cloud.

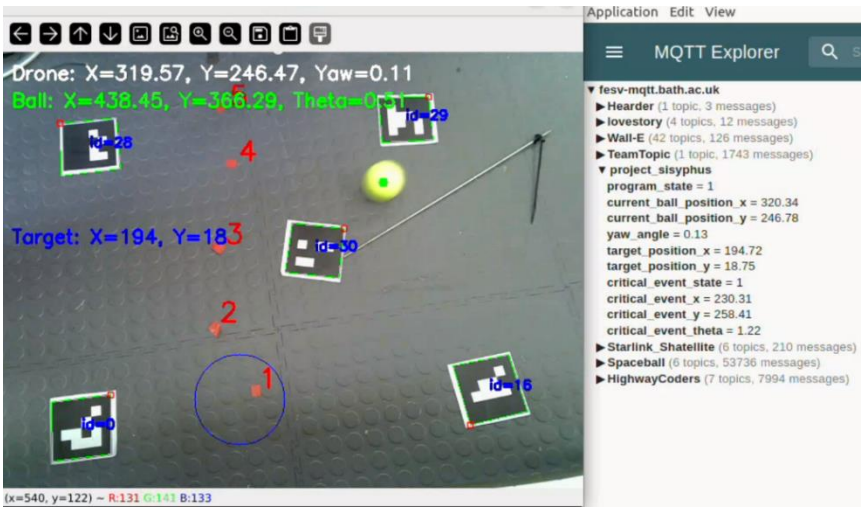

Figure 8: Frame taken 1 second later in order to compare theta values. As shown, the angle remained very consistent when following a straight path trajectory.

[8-9] Data Broadcasting and Integration Validation

Data broadcasting between the image processing module and the motor control was verified using MQTT Explorer to check for continuous, consistent data transmission.

MQTT Verification: Using MQTT Explorer, each broadcasted message, including positional and orientation data, was monitored to ensure no data loss and correct data refresh rates.

Live Feedback in MATLAB: Edwin's MATLAB plots displayed real-time changes from the broadcast data, showing path adjustments in response to detected obstacles. This allowed us to closely monitor abnormalities with the signal broadcasting which may have been too fast for us to catch using MQTT Explorer. An example was that the critical event state would occasionally turn off for a few milliseconds, only noticeable in MATLAB plots, which caused the robot movement to be erratic. To fix this, a loop was added in the Python code to keep critical_event_state to 1 for up to a second with no detection, keeping the signal consistent.

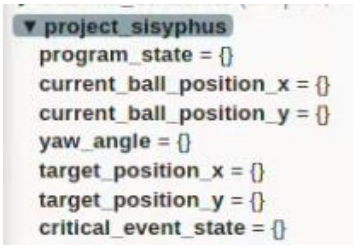

Figure 9: MQTT Explorer before code initialisation. All data entries are blank.

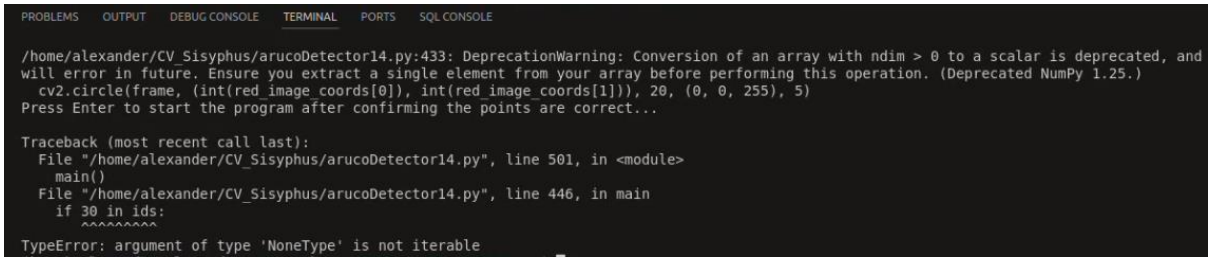

Figure 10: Terminal error message when ArUco code 30 (inside rover) was not detected.

```
'''
if 30 in ids:
    marker_detection_start_time = None  # Reset timer if all markers are detected
else:
    # Start the timer if it wasn't already started
    if marker_detection_start_time is None:
        marker_detection_start_time = time.time()

    # If markers have not been detected for 2 seconds, change to program_state = 2
    elif time.time() - marker_detection_start_time >= 20:
        program_state = 2
        client.publish(MQTT_TOPIC_STATE, str(program_state))
'''
```

Figure 11: Snippet of code causing errors. The functionality was to stop the drone if it went out of bounds for too long, however this was rarely used and caused more problems than it fixed.

[10] Code Development

Maintaining modular code was a focus to ensure efficient troubleshooting and future adaptations.

Modular Structure Maintenance: Functions were consistently separated by functionality (e.g. red object detection and blue object detection) to streamline debugging and improve readability.

GPT-Assisted Organisation: ChatGPT was used to restructure messy functions, providing clean and logical groupings for modular organisation.

Backup Versions: Regular backups were created, with each version saved incrementally to allow for quick restoration in case of errors. A total of 16 versions were created to ensure safe progress throughout testing and integration.

```python
# Helper functions (same as before)
def detect_aruco_markers(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    corners, ids, rejected = detector.detectMarkers(gray)
    return corners, ids

def estimate_homography(corner_coords):
    if len(corner_coords) < 4:
        return None
    real_points = []
    image_points = []
    for marker_id, (cx, cy) in corner_coords.items():
        real_points.append(real_world_coords[marker_id])
        image_points.append([cx, cy])
    real_points = np.array(real_points, dtype="float32")
    image_points = np.array(image_points, dtype="float32")
    H, _ = cv2.findHomography(image_points, real_points)
    return H

def apply_homography(H, image_point):
    """Convert image coordinates to real-world coordinates using homography."""
    image_point = np.array([image_point[0], image_point[1], 1]).reshape((3, 1))
    world_point = np.dot(H, image_point)
    world_point /= world_point[2]  # Normalize the point by dividing by the Z coordinate
    return world_point[0], world_point[1]
```

Figure 12: Example of 3 different organised functions, each doing separate tasks that may be called upon in the main() block whenever required.

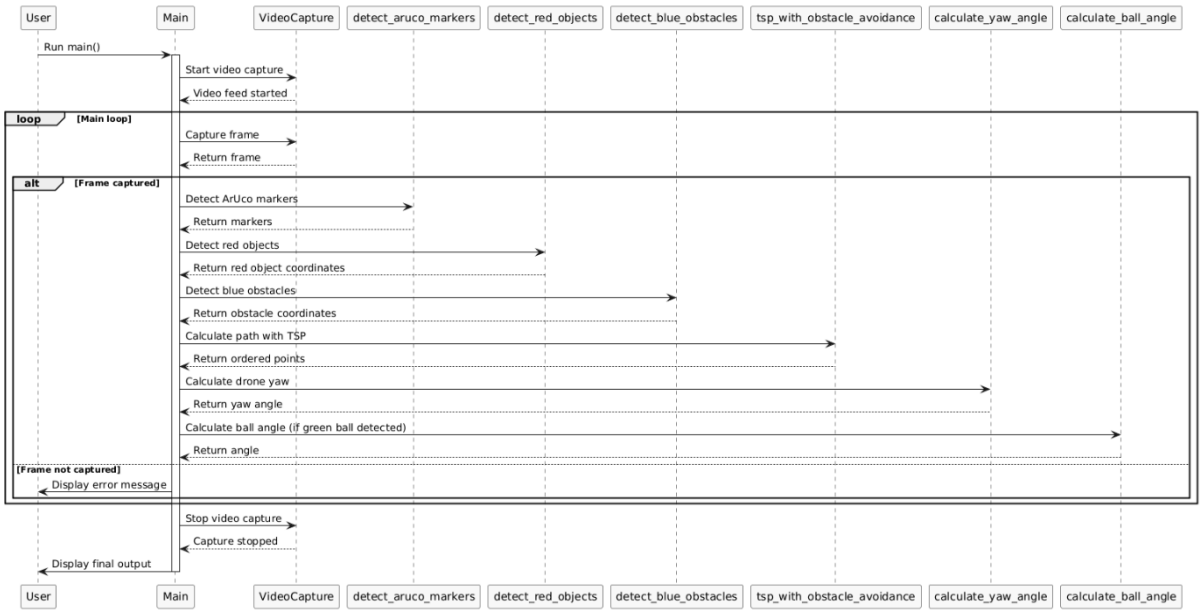

Figure 13: Simplified sequence diagram of main code functionality with primary functions.

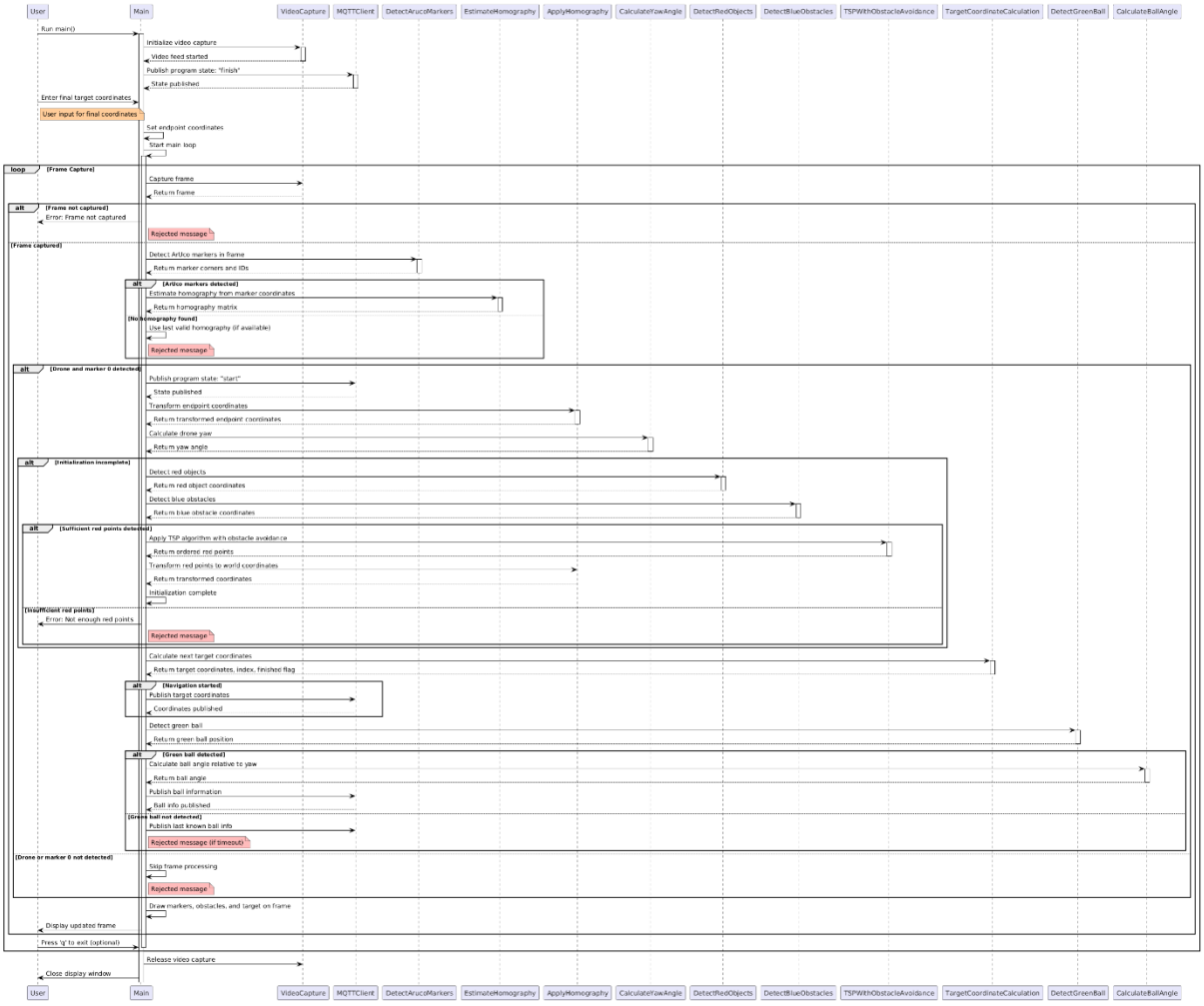

Figure 14: Full and detailed sequence diagram of exactly how the code recalls every function, including error messages.

System Integration

System Integration with Low-Level Controls

The integration of the high-level image processing module, written in Python with OpenCV, and the low-level control module in C relied heavily on the MQTT protocol to maintain real-time data synchronisation. The high-level module processed visual information, detecting ArUco markers and coloured obstacles, and then transmitted critical coordinates, angles, and obstacle details to the low-level module, which executed motor control based on these inputs. MQTT facilitated the asynchronous, low-latency communication necessary for this project's real-time demands (Hunkeler, Truong & Stanford-Clark, 2008). *See Figures 7 and 8.*

Initial Test of Blue Markers

The first test with blue markers illustrated how sensitive the system was to lighting changes. OpenCV sometimes misidentified dark regions, mistaking navy hues for black, resulting in unreliable detection of actual blue obstacles. This initial miscalculation influenced our choice to remove blue Lego obstacles later, as multiple detections on a single blue item confused the system's path-planning. Ensuring a consistent physical colour mirrors common practice for computer vision tasks with colour detection (Chen et al., 2017). *See Figure 6.*

First Critical Event Tests and Green Marker Detection

Our initial tests involved using a green marker (a green whiteboard pen cap) to emulate dynamic obstacles. We found that the system occasionally lagged in updating coordinates due to minor MQTT delays. To ensure responsive obstacle detection, frame-by-frame analysis was performed, verifying coordinate updates for accurate real-time detection. This approach is aligned with robotics industry standards for safety-critical systems, where data freshness is paramount for responsive behaviour (Åström & Hägglund, 2006).

```python
def detect_green_ball(frame):
    """Detect the green ball in the frame and return its coordinates."""
    hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower_green = np.array([35, 100, 100])
    upper_green = np.array([85, 255, 255])
    mask = cv2.inRange(hsv_frame, lower_green, upper_green)
    contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        ((x, y), radius) = cv2.minEnclosingCircle(largest_contour)
        if radius > 5:
            return int(x), int(y)
    return None
```

Figure 15: The arrays lower_green and upper_green determine RGB thresholds for detection.

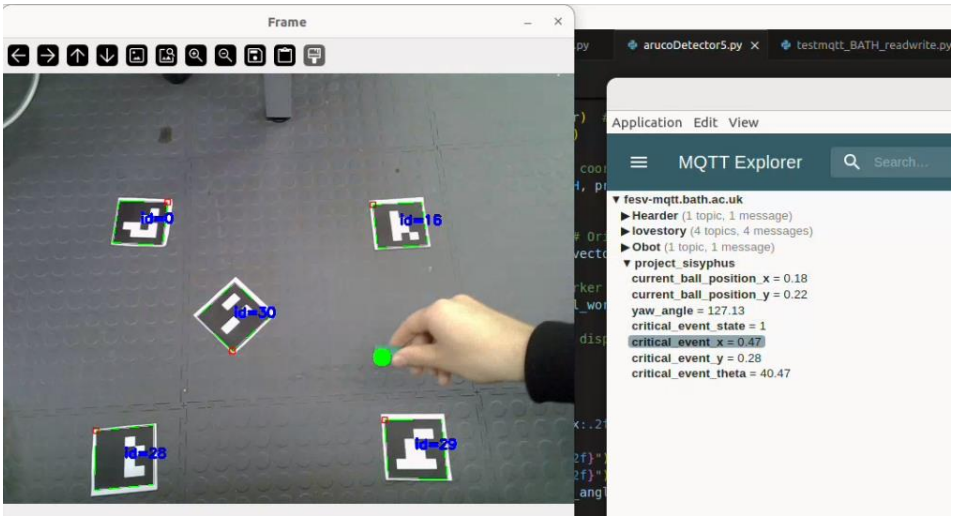

Figure 16: Early stage testing of green detection function using a green whiteboard pen lid. The centre point of the bounding box is displayed live onto the camera feed using the cv2.circle function from OpenCV.

Testing Initialisation Without ArUco Code 30

During setup, the program required ArUco code 30 to be visible for initialisation. This requirement became a bottleneck, as it occasionally failed to detect the code under inconsistent lighting. We devised a workaround: an “emergency ArUco code” was created to bypass the initialisation loop. While effective, this required additional setup, making initialisation more complex and dependent on multiple people, which is less ideal than standard practice in autonomous systems, where robust initialisation is crucial (Garrido-Jurado et al., 2016). Eventually, we adjusted the code to include a failsafe for initialisation without constant reliance on code 30, mitigating this dependency.

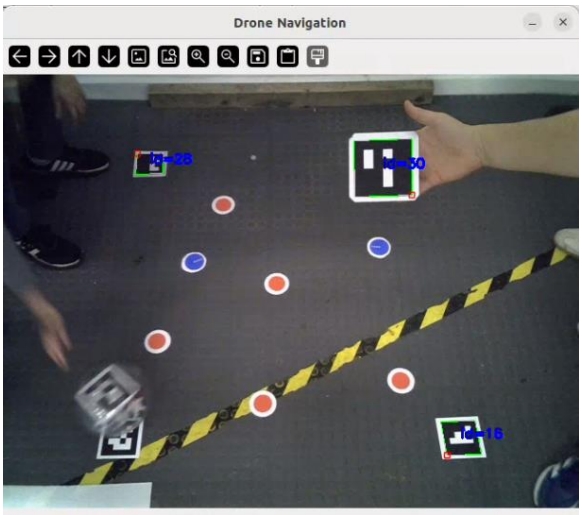

Figure 17: Example of rover’s ArUco code being undetectable, causing us to use the “emergency ArUco code” to allow initialisation. This caused hectic initialisation conditions.

Comparisons to Common Practice

In commercial and research robotics, the use of Travelling Salesman Problem (TSP) algorithms is a standard approach to ensure efficiency when managing multiple target points (Karapetyan, D. & Gutin, G., 2011). Our project aimed to line up the red markers, testing the TSP's efficacy in selecting the shortest path and proving successful in reducing travel time across points.

```python
def tsp_with_obstacle_avoidance(start_point, end_point, middle_points, blue_obstacles, obstacle_threshold=10):
    """Solve TSP while avoiding blue obstacles by expanding their size."""
    optimal_order = None
    min_total_distance = float('inf')

    # Test all permutations of middle points
    for perm in itertools.permutations(middle_points):
        p_list = [start_point] + list(perm) + [end_point]
        obstacle_in_path = False

        # Check for obstacles between consecutive points
        for i in range(len(p_list) - 1):
            for obstacle in blue_obstacles:
                if is_obstacle_near_line(p_list[i], p_list[i + 1], obstacle, obstacle_threshold):
                    obstacle_in_path = True
                    break
            if obstacle_in_path:
                break

        if obstacle_in_path:
            continue

        # Calculate total distance for this permutation
        total_distance = sum(calculate_distance(p_list[i], p_list[i + 1]) for i in range(len(p_list) - 1))

        if total_distance < min_total_distance:
            min_total_distance = total_distance
            optimal_order = perm

    # Return the optimal order
    return [start_point] + list(optimal_order) + [end_point] if optimal_order else [start_point] + middle_points + [end_point]
```

Figure 18: Code snippet of TSP algorithm. This function required lots of data before being recalled in main(), therefore making it one of the last things to be sequentially computed.

Challenges with Lighting and the Use of Flash

Lighting variations presented a significant challenge, as insufficient illumination led to inconsistent detection of blue and red markers (Lichtsteiner, Posch & Delbruck, 2008). During low-light tests, OpenCV misinterpreted shadows as blue markers, skewing obstacle detection results. Toggling the flash during initialisation ensured more consistent results by reducing reliance on ambient lighting. While toggling light sources is often employed to standardise conditions in robotics, solutions like dynamic thresholding or adaptive algorithms could further improve resilience to lighting changes, as seen in other computer vision systems for autonomous drones (Garrido-Jurado, S., et al., 2016).

```python
# Check if 1 second has passed
elapsed_time = time.time() - start_time

if elapsed_time >= 1 and waiting_for_input:
    print("Press Enter to start the program after confirming the points are correct...")
    user_input = input()  # Wait for user input

    if user_input.lower() == 'q':
        break
    else:
        program_state = 1  # Change to start state after input
        client.publish(MQTT_TOPIC_STATE, str(program_state))  # Publish the updated state
        waiting_for_input = False  # Stop waiting for input
```

Figure 19: Code snippet of function within main() that allows user to see a snapshot of the detection before committing to changing the program_state to 1 (which initialises the rover). The code required a 1 second timer before pausing. This was to prevent the first frame from being completely black, due to the webcam needing a few milliseconds to turn on.

Data Broadcasting and MQTT Broker Observations

Using MQTT Explorer, we examined the real-time broadcasting of data from the image processing module to low-level controls. The broker's messages confirmed correct data transmission under various operational states. Any error, such as failure to detect the ArUco code during initialisation, prompted an error message, indicating that MQTT's internal checks were functioning as expected. This real-time troubleshooting mirrored professional practice, where telemetry analysis tools like MQTT Explorer are invaluable for data monitoring and debugging (Hunkeler, Truong & Stanford-Clark, 2008).

```python
client = mqtt.Client()
client.username_pw_set("student", password="[redacted]")
client.connect("fesv-mqtt.bath.ac.uk", 31415, 60)
client.loop_start()

# The next position for rover to go towards
MQTT_TOPIC_TARGET_X = 'project_sisyphus/target_position_x'
MQTT_TOPIC_TARGET_Y = 'project_sisyphus/target_position_y'

# Other MQTT Topics (same as before)
MQTT_TOPIC_X = 'project_sisyphus/current_ball_position_x'           # ArUco code x coord
MQTT_TOPIC_Y = 'project_sisyphus/current_ball_position_y'           # ArUco code y coord
MQTT_TOPIC_YAW = 'project_sisyphus/yaw_angle'                       # ArUco code 30 angle
MQTT_TOPIC_EVENT_STATE = 'project_sisyphus/critical_event_state'    # State is 0 when no green ball detected, 1 when detected
MQTT_TOPIC_EVENT_X = 'project_sisyphus/critical_event_x'            # Coordinates of green ball
MQTT_TOPIC_EVENT_Y = 'project_sisyphus/critical_event_y'            # Coordinates of green ball
MQTT_TOPIC_EVENT_THETA = 'project_sisyphus/critical_event_theta'    # Green ball angle

# MQTT Topics to add to main
MQTT_TOPIC_STATE = 'project_sisyphus/program_state'  # 1 for start, 2 for stop
MQTT_TOPIC_X_ARENA = 'project_sisyphus/arena_x'      # to be 500 automatically during setup phase
MQTT_TOPIC_Y_ARENA = 'project_sisyphus/arena_y'      # to be 500 automatically during setup phase
```

Figure 20: Code snippet of Python connecting with MQTT broker and initialising MQTT topics for data transfer.

Moving Obstacle Angle Detection and Yaw Adjustments

Incorporating yaw detection to track moving obstacles required optimising the program's response to rapid shifts in angular data. This involved calculating angular displacement based on past frame data and adjusting the rover's trajectory in real time. Comparative analysis of the angle detection's accuracy highlighted the minor lag introduced by MQTT, and adjustments were made to the control loop to mitigate this effect. This approach reflects best practices in advanced robotics, where robust angle detection is essential for reliable path modification when encountering moving obstacles (Kortenkamp, D., Bonasso, P.R. & Murphy, R., 1998).

```python
def calculate_yaw_angle(corners0, corners_drone):
    """Calculate the yaw angle of ArUco marker 30 relative to ArUco marker 0 in radians."""
    vector_0 = corners0[1] - corners0[0]  # Orientation vector of marker 0
    vector_drone = corners_drone[1] - corners_drone[0]  # Orientation vector of drone marker

    # Calculate the orientation angles in radians
    angle_0 = np.arctan2(vector_0[1], vector_0[0]) % (2 * np.pi)
    angle_drone = np.arctan2(vector_drone[1], vector_drone[0]) % (2 * np.pi)

    # Calculate the yaw angle difference in radians
    yaw_angle = (angle_drone - angle_0) % (2 * np.pi)

    return yaw_angle
```

Figure 21: Code snippet of function that determines the angle of ArUco code 30 relative to the angle of ArUco code 0 by determining both codes orientation vectors, estimating an angle using arctan2 from numpy then comparing values for a relative estimation.

```python
def calculate_ball_angle(previous_position, current_position, marker0_angle):
    """Calculate ball movement angle relative to ArUco marker 0's orientation in radians."""
    dx = current_position[0] - previous_position[0]
    dy = current_position[1] - previous_position[1]

    # Calculate the angle of the ball's movement in radians
    ball_angle = np.arctan2(dy, dx) % (2 * np.pi)

    # Adjust ball angle relative to ArUco marker 0's orientation, assuming marker0_angle is in radians
    relative_ball_angle = (ball_angle - marker0_angle) % (2 * np.pi)

    return relative_ball_angle
```

Figure 22: Code snippet of function that estimates the angle of the moving green ball based on the change in x and change in y coordinates from 2 frames, then using the arctan2 function in numpy to estimate the angle.

Collaboration and Communication Challenges

Our team faced numerous challenges related to the integration of Python-based high-level controls and the C-based low-level motor code. Ensuring seamless communication required close collaboration and frequent debugging sessions (Brooks, 1986). During early tests, issues arose from inconsistent coordinate updates and MQTT lag, which were discussed in biweekly meetings and refined through code revisions. One notable example involved modifications in the path-planning function to improve point ordering and smooth transitions between detection and control.

We tackled these multi-language integration challenges with a mix of version control for code backups (totalling 16 iterations) and peer reviews to cross-check modifications. Our approach ensured that Python-C integration was reliable, a necessity in robotics where subsystems often span multiple programming languages, making synchronisation key to system stability.

Improving Detection Consistency with Freeze Frames

The freeze-frame mechanism became invaluable during testing, allowing us to check camera alignment and marker visibility before starting a session. This mechanism offered a controlled way to confirm camera accuracy, particularly under varied lighting conditions, before executing the high-level commands. This freeze-frame approach is commonly employed in robotics for preliminary calibration, particularly in visual feedback systems, to prevent error accumulation from minor initial misalignments (He, Barkana & Han, 2010).

Preliminary Critical Event Testing with Frame-by-Frame Verification

To test critical events, such as when the object left a designated area for more than two seconds, we implemented frame-by-frame verification, capturing the ball's movement in real time. This detailed analysis ensured that the ball's position and angle were promptly updated and broadcast via MQTT to prevent unplanned navigation changes. Frame-by-frame comparisons provided granular insights into the critical event function's responsiveness, aligning with common testing methodologies in robotics for validating time-sensitive functions (Kortenkamp, Bonasso & Murphy, 1998).

Hardware Constraints and Code Adjustments for Smaller Testing Space

Due to limited gantry space, our test environment required modifying the code to account for markers set 50 cm apart rather than the ideal 1-meter spacing. This limitation introduced challenges for PID control, as the smaller space increased sensitivity to positional errors

(Åström & Hägglund, 2006). Testing under these conditions emphasised the need for PID adjustments to maintain stability. Such testing constraints mirror real-world robotics scenarios, where environmental limitations often necessitate adaptive solutions in software to maintain reliable performance within constrained parameters.

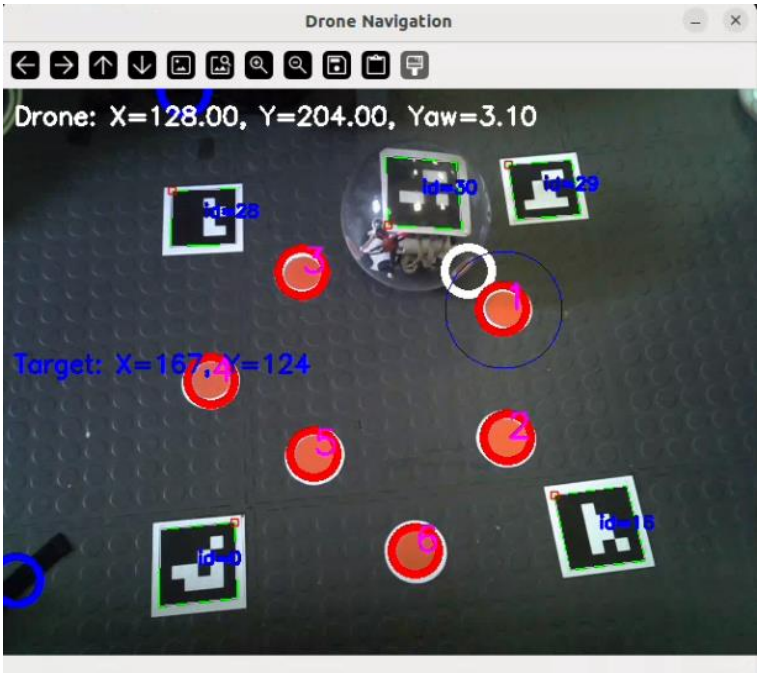

Figure 23: Image of sub-optimal testing environment with evidence of required adaptations.

Final Adjustments and Homography from Different Angles

To test homography transformations from different perspectives, final trials were conducted with varied camera angles. Adjusting for different viewpoints ensured that the real-world mapping remained accurate across orientations, a critical factor for navigating dynamic environments. Successful homography at various angles demonstrated the robustness of our setup, echoing industry practices where multi-angle calibration is used to enhance positional accuracy in visual localisation systems (LaValle, 1998).

In summary, our integration and testing procedures followed industry-standard practices in robotics, from robust error handling and data broadcasting to fine-tuning detection algorithms for dynamic conditions. Through continuous adjustments and team collaboration, we refined the system to handle real-time updates and react to critical events, balancing complex high-level processing with low-level control reliability. The freeze-frame mechanism, iterative TSP optimisation, and adaptive lighting solutions all contributed to a robust, synchronised system capable of navigating and responding to an unpredictable environment effectively.

Reflection on Aims

Our primary goal was to develop a Mars rover system with robust adaptive navigation, utilising high-level image processing and low-level motor control for real-time obstacle detection and avoidance. Achieving this balance required intensive testing to ensure that each subsystem communicated reliably under varied conditions. Our solution included techniques such as homography for spatial accuracy, innovative use of the freeze-frame mechanism, the Travelling Salesman Problem, and real-time data updates over MQTT, which collectively contributed to a stable navigation framework.

A notable achievement was the adaptive lighting toggling method, which significantly improved marker detection consistency in low-light environments. This innovation highlighted the importance of environmental controls in enhancing detection accuracy, yet revealed an area for improvement in dynamically managing lighting changes. Additionally, implementing the Travelling Salesman Problem (TSP) algorithm ensured optimal path planning, though future iterations could benefit from path planning methods that add more nodes to allow the rover to avoid more complex obstacle layouts, such as the A* or RRT* pathfinding algorithm (LaValle, 1998).

For future development, potential improvements include upgrading the webcam's lighting system to automatically adjust for varying brightness levels and include flash for the initial detection, thus increasing resilience in various lighting conditions. Refining the green moving obstacle detection colour threshold for dim lighting conditions would further increase reliability. Overall, this project provided a strong foundation in combining image processing with real-time robotic control, setting a benchmark for future advancements.

References

Åström, K.J. & Hägglund, T., 2006. *Advanced PID control*. ISA.

Brooks, R.A., 1986. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1), pp.14-23.

Chen, J., et al., 2017. *Robust Colour Detection in Computer Vision Systems*. IEEE Transactions on Image Processing, 26(5), pp.2614-2627.

Garrido-Jurado, S., et al., 2016. *Automatic generation and detection of highly reliable fiducial markers under occlusion*. Pattern Recognition, 47(6), pp.2280-2292.

He, H., Barkana, B.D. & Han, J., 2010. *Detection of critical event functions in real-time systems*. IEEE Transactions on Industrial Electronics, 57(5), pp.1762-1771.

Hunkeler, U., Truong, H.L. & Stanford-Clark, A., 2008. MQTT-S–A publish/subscribe protocol for Wireless Sensor Networks. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pp.791-796.

Karapetyan, D. & Gutin, G., 2011. *Algorithms for the Traveling Salesman Problem*. Lecture Notes in Computer Science, Springer, pp.121-128.

Kortenkamp, D., Bonasso, P.R. & Murphy, R., 1998. *Artificial Intelligence and Mobile Robots*. MIT Press.

LaValle, S.M., 1998. Rapidly-exploring random trees: A new tool for path planning. *Technical Report TR 98-11*, Computer Science Department, Iowa State University.

Lichtsteiner, P., Posch, C. & Delbruck, T., 2008. A 128x128 120 dB 15 μs latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits*, 43(2), pp.566-576.

Van De Sande, K.E.A., et al., 2010. Evaluating Color Descriptors for Object and Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9), pp.1582-1596.

Garrido-Jurado, S., et al., 2014. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6), pp.2280-2292.

Appendix

Code

```
import cv2
import numpy as np
import socket
import paho.mqtt.client as mqtt
import heapq
import itertools
import time

# ArUco and MQTT setup (same as before)
aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
parameters = cv2.aruco.DetectorParameters()
detector = cv2.aruco.ArucoDetector(aruco_dict, parameters)
corner_ids = [0, 16, 28, 29]
drone_id = 30
real_world_coords = {0: (0.0, 0.0), 16: (1000, 0.0), 28: (0.0, 1000), 29: (1000, 1000)}
ARUCO_MARKER_SIZE_M = 100
client = mqtt.Client()
client.username_pw_set("student", password="HousekeepingGlintsStreetwise")
client.connect("fesv-mqtt.bath.ac.uk", 31415, 60)
client.loop_start()

# The next position for rover to go towards
```

```
MQTT_TOPIC_TARGET_X = 'project_sisyphus/target_position_x'
MQTT_TOPIC_TARGET_Y = 'project_sisyphus/target_position_y'

# Other MQTT Topics (same as before)
MQTT_TOPIC_X = 'project_sisyphus/current_ball_position_x'           # ArUco code x coord
MQTT_TOPIC_Y = 'project_sisyphus/current_ball_position_y'           # ArUco code y coord
MQTT_TOPIC_YAW = 'project_sisyphus/yaw_angle'                       # ArUco code 30 angle
MQTT_TOPIC_EVENT_STATE = 'project_sisyphus/critical_event_state'    # State is 0 when no green ball
detected, 1 when detected
MQTT_TOPIC_EVENT_X = 'project_sisyphus/critical_event_x'            # Coordinates of green ball
MQTT_TOPIC_EVENT_Y = 'project_sisyphus/critical_event_y'            # Coordinates of green ball
MQTT_TOPIC_EVENT_THETA = 'project_sisyphus/critical_event_theta'    # Green ball angle

# MQTT Topics to add to main
MQTT_TOPIC_STATE = 'project_sisyphus/program_state'  # 1 for start, 2 for stop
MQTT_TOPIC_X_ARENA = 'project_sisyphus/arena_x'      # to be 500 automatically during setup phase
MQTT_TOPIC_Y_ARENA = 'project_sisyphus/arena_y'      # to be 500 automatically during setup phase

# Helper functions (same as before)
def detect_aruco_markers(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    corners, ids, rejected = detector.detectMarkers(gray)
    return corners, ids

def estimate_homography(corner_coords):
    if len(corner_coords) < 4:
        return None
    real_points = []
    image_points = []
    for marker_id, (cx, cy) in corner_coords.items():
        real_points.append(real_world_coords[marker_id])
        image_points.append([cx, cy])
    real_points = np.array(real_points, dtype="float32")
    image_points = np.array(image_points, dtype="float32")
    H, _ = cv2.findHomography(image_points, real_points)
    return H

def apply_homography(H, image_point):
    """Convert image coordinates to real-world coordinates using homography."""
    image_point = np.array([image_point[0], image_point[1], 1]).reshape((3, 1))
    world_point = np.dot(H, image_point)
    world_point /= world_point[2]  # Normalize the point by dividing by the Z coordinate
    return world_point[0], world_point[1]

def calculate_yaw_angle(corners0, corners_drone):
    """Calculate the yaw angle of ArUco marker 30 relative to ArUco marker 0 in radians."""
    vector_0 = corners0[1] - corners0[0]  # Orientation vector of marker 0
```

```
    vector_drone = corners_drone[1] - corners_drone[0]  # Orientation vector of drone marker

    # Calculate the orientation angles in radians
    angle_0 = np.arctan2(vector_0[1], vector_0[0]) % (2 * np.pi)
    angle_drone = np.arctan2(vector_drone[1], vector_drone[0]) % (2 * np.pi)

    # Calculate the yaw angle difference in radians
    yaw_angle = (angle_drone - angle_0) % (2 * np.pi)

    return yaw_angle

def calculate_distance(p1, p2):
    """Calculate Euclidean distance between two points."""
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def is_obstacle_near_line(p1, p2, obstacle, threshold):
    """Check if an obstacle is within a certain distance from the line segment between p1 and p2."""
    p1, p2 = np.array(p1), np.array(p2)
    obstacle = np.array(obstacle)

    # Calculate the distance from the obstacle to the line segment
    line_vec = p2 - p1
    p1_to_obstacle = obstacle - p1
    line_len = np.linalg.norm(line_vec)

    # Check if line is a point
    if line_len == 0:
        return np.linalg.norm(p1_to_obstacle) < threshold

    line_unit_vec = line_vec / line_len
    projection_length = np.dot(p1_to_obstacle, line_unit_vec)

    # Find the closest point on the line segment to the obstacle
    if projection_length < 0:
        closest_point = p1
    elif projection_length > line_len:
        closest_point = p2
    else:
        closest_point = p1 + projection_length * line_unit_vec

    # Distance from the obstacle to the closest point on the line
    distance_to_line = np.linalg.norm(obstacle - closest_point)

    return distance_to_line < threshold

def detect_red_objects(H, frame, x_coord, y_coord, num_points=6):
    """Detect red objects and return points sorted by distance to (x_coord, y_coord)."""
```

```
    hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower_red = np.array([0, 100, 100])
    upper_red = np.array([10, 255, 255])
    mask = cv2.inRange(hsv_frame, lower_red, upper_red)
    contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    red_points = []
    for contour in contours:
        ((x, y), radius) = cv2.minEnclosingCircle(contour)
        if radius > 5:
            red_points.append((int(x), int(y)))
            if len(red_points) == num_points:  # Limit to num_points
                break
    if len(red_points) < 2:
        return None, None, []  # Not enough points detected

    # Sort points by vector distance to the given x_coord and y_coord
    red_points.sort(key=lambda point: np.linalg.norm(np.array(apply_homography(H, [point[0],
point[1]])) - np.array([x_coord, y_coord])))

    point_1 = red_points[0]
    point_last = red_points[-1]
    middle_points = red_points[1:-1]
    num_points = num_points

    return point_1, point_last, middle_points, num_points

def detect_blue_obstacles(frame, max_obstacles=2):
    """Detect blue obstacles in the frame."""
    hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower_blue = np.array([100, 150, 0])
    upper_blue = np.array([140, 255, 255])
    mask = cv2.inRange(hsv_frame, lower_blue, upper_blue)
    contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    blue_obstacles = []
    for contour in contours:
        ((x, y), radius) = cv2.minEnclosingCircle(contour)
        if radius > 5:
            blue_obstacles.append((int(x), int(y)))
            if len(blue_obstacles) == max_obstacles:
                break
    return blue_obstacles

def tsp_with_obstacle_avoidance(start_point, end_point, middle_points, blue_obstacles,
obstacle_threshold=10):
    """Solve TSP while avoiding blue obstacles by expanding their size."""
```

```
    optimal_order = None
    min_total_distance = float('inf')

    # Test all permutations of middle points
    for perm in itertools.permutations(middle_points):
        p_list = [start_point] + list(perm) + [end_point]
        obstacle_in_path = False

        # Check for obstacles between consecutive points
        for i in range(len(p_list) - 1):
            for obstacle in blue_obstacles:
                if is_obstacle_near_line(p_list[i], p_list[i + 1], obstacle, obstacle_threshold):
                    obstacle_in_path = True
                    break
            if obstacle_in_path:
                break

        if obstacle_in_path:
            continue

        # Calculate total distance for this permutation
        total_distance = sum(calculate_distance(p_list[i], p_list[i + 1]) for i in range(len(p_list) -
1))

        if total_distance < min_total_distance:
            min_total_distance = total_distance
            optimal_order = perm

    # Return the optimal order
    return [start_point] + list(optimal_order) + [end_point] if optimal_order else [start_point] +
middle_points + [end_point]

def detect_green_ball(frame):
    """Detect the green ball in the frame and return its coordinates."""
    hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower_green = np.array([35, 100, 100])
    upper_green = np.array([85, 255, 255])
    mask = cv2.inRange(hsv_frame, lower_green, upper_green)
    contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        ((x, y), radius) = cv2.minEnclosingCircle(largest_contour)
        if radius > 5:
            return int(x), int(y)
    return None
```

```python
def calculate_ball_angle(previous_position, current_position, marker0_angle):
    """Calculate ball movement angle relative to ArUco marker 0's orientation in radians."""
    dx = current_position[0] - previous_position[0]
    dy = current_position[1] - previous_position[1]

    # Calculate the angle of the ball's movement in radians
    ball_angle = np.arctan2(dy, dx) % (2 * np.pi)

    # Adjust ball angle relative to ArUco marker 0's orientation, assuming marker0_angle is in radians
    relative_ball_angle = (ball_angle - marker0_angle) % (2 * np.pi)

    return relative_ball_angle

def target_coordinate_calculation(x_coord, y_coord, current_index, red_world_coords,
endpoint_coordinate_x, endpoint_coordinate_y):
    """
    Determine the next target coordinates based on the red object positions.

    Args:
    - x_coord (float): The x-coordinate of the drone.
    - y_coord (float): The y-coordinate of the drone.
    - current_index (int): The current index for the target coordinate arrays.
    - red_world_coords (list): List of red object world coordinates.
    - endpoint_coordinate_x (float): The x-coordinate of the final endpoint.
    - endpoint_coordinate_y (float): The y-coordinate of the final endpoint.

    Returns:
    - target_coordinate_x (float): The next target x-coordinate.
    - target_coordinate_y (float): The next target y-coordinate.
    - current_index (int): Updated index for the target coordinate arrays.
    - finished (bool): Whether the final target has been reached.
    """
    # Extract the x and y coordinates of the red objects
    target_coordinate_x_array = [point[0].item() for point in red_world_coords]
    target_coordinate_y_array = [point[1].item() for point in red_world_coords]

    # If no red objects are detected, return the current coordinates
    if not target_coordinate_x_array or not target_coordinate_y_array:
        return x_coord, y_coord, current_index, False

    # Initialize finished as False
    finished = False

    # Initialize target coordinates
    target_coordinate_x = endpoint_coordinate_x
    target_coordinate_y = endpoint_coordinate_y
```

```
    # If we haven't reached the last red marker yet
    if current_index < len(target_coordinate_x_array):
        target_coordinate_x = target_coordinate_x_array[current_index]
        target_coordinate_y = target_coordinate_y_array[current_index]

        # Check if the drone is within 50 units of the current red object
        if abs(x_coord - target_coordinate_x) <= 50 and abs(y_coord - target_coordinate_y) <= 50:
            # Move to the next red object or the endpoint if on the last red marker
            if current_index == len(target_coordinate_x_array) - 1:
                current_index = 7  # Lock index at 7 once at the endpoint
            else:
                current_index += 1

    # Check if the drone has reached the endpoint
    if current_index == 7 and abs(x_coord - endpoint_coordinate_x) <= 50 and abs(y_coord -
endpoint_coordinate_y) <= 50:
        finished = True
        print("JOURNEY FINISHED!")

    # Return the next target coordinates
    return target_coordinate_x, target_coordinate_y, current_index, finished

def main():
    # Set up video capture
    cap = cv2.VideoCapture(0)
    last_valid_homography = None
    program_state = 2  # 0 = idle, 1 = start, 2 = finish
    current_index = 0
    initialization_complete = False  # Flag to indicate initialization is complete

    client.publish(MQTT_TOPIC_STATE, str(program_state))

    previous_position = None  # Initialize previous position as None

    endpoint_coordinate_x = float(input("Enter the new final x-coordinate: "))
    endpoint_coordinate_y = float(input("Enter the new final y-coordinate: "))

    # Initialize a timer
    start_time = time.time()
    elapsed_time = 0
    waiting_for_input = True  # Flag to control when to wait for input


    # Initialize last detection time
    last_detection_time = 0

    while True:
        ret, frame = cap.read()
```

```
        if not ret:
            break

        # ArUco marker detection
        corners, ids = detect_aruco_markers(frame)

        corner_coords = {}
        if ids is not None:
            ids = ids.flatten()
            for i, marker_id in enumerate(ids):
                if marker_id in corner_ids:
                    c = corners[i][0]
                    marker_center = (np.mean(c[:, 0]), np.mean(c[:, 1]))
                    corner_coords[marker_id] = marker_center

        # Homography estimation
        H = estimate_homography(corner_coords)

        if H is not None:
            last_valid_homography = H
        elif last_valid_homography is not None:
            H = last_valid_homography

        # ArUco marker and drone detection, along with yaw calculation
        if H is not None and ids is not None and drone_id in ids and 0 in ids:

            client.publish(MQTT_TOPIC_STATE, str(program_state))

            drone_index = np.where(ids == drone_id)[0][0]
            marker0_index = np.where(ids == 0)[0][0]
            c_drone = corners[drone_index][0]
            c_marker0 = corners[marker0_index][0]

            # Gets endpoint coordinates relative to real coordinates then shows them with white circle
            raw_coordinates = (endpoint_coordinate_x, endpoint_coordinate_y)
            endpoint_coords = apply_homography(np.linalg.inv(H), raw_coordinates)
            endpoint_coordinate_x_real = endpoint_coords[0].item()
            endpoint_coordinate_y_real = endpoint_coords[1].item()
            cv2.circle(frame, (int(endpoint_coordinate_x_real), int(endpoint_coordinate_y_real)), 20,
(255, 255, 255), 5)

            # Drone world coordinates and yaw
            drone_center = (np.mean(c_drone[:, 0]), np.mean(c_drone[:, 1]))
            drone_world_coords = apply_homography(H, drone_center)
            x_coord = int(drone_world_coords[0].item())  # Fix deprecated conversion
            y_coord = int(drone_world_coords[1].item())  # Fix deprecated conversion
            yaw_angle = round(float(calculate_yaw_angle(c_marker0, c_drone)), 1)  # This is your
```

```
marker0_angle

            # Check if initialization is complete
            if not initialization_complete:
                # Detect red obstacles and blue obstacles
                point_1, point_last, middle_points, num_points = detect_red_objects(H, frame, x_coord,
y_coord)
                blue_obstacles = detect_blue_obstacles(frame)

                # Ensure we have enough red points
                if point_1 and point_last and len(middle_points) == (num_points - 2):
                    # Apply TSP with obstacle avoidance
                    ordered_points = tsp_with_obstacle_avoidance(point_1, point_last, middle_points,
blue_obstacles)
                    # Transform red points to world coordinates using homography
                    red_world_coords = [apply_homography(H, point) for point in ordered_points]

                    # Set the flag to indicate initialization is complete
                    initialization_complete = True
                else:
                    print("Not enough red points detected.")
                    continue  # Skip the rest of the loop if we don't have enough points

            # Get target coordinates from the function
            target_coord_x, target_coord_y, current_index, finished =
target_coordinate_calculation(x_coord, y_coord, current_index, red_world_coords,
endpoint_coordinate_x, endpoint_coordinate_y)

            if program_state == 1:  # Ensure the navigation starts after the user input
                # Continue with navigation logic...
                if finished:  # Only transition to state 2 after navigation is finished
                    program_state = 2
                    client.publish(MQTT_TOPIC_STATE, str(program_state))

            target_coord_x = int(target_coord_x)
            target_coord_y = int(target_coord_y)

            # Define the real-world coordinates
            real_world_coords = (target_coord_x, target_coord_y)

            # Apply homography to transform real-world coordinates to image coordinates
            image_coords = apply_homography(np.linalg.inv(H), real_world_coords)  # Invert H to go
from real-world to image

            # Convert image_coords to integers (OpenCV expects integers for the center point)
            image_coords_int = (int(image_coords[0]), int(image_coords[1]))
```

```
            # Draw a circle at the corresponding image coordinate
            cv2.circle(frame, image_coords_int, 50, (225, 0, 0), 1)

            # Publish drone coordinates and yaw to MQTT
            client.publish(MQTT_TOPIC_X, f"{x_coord:.2f}")
            client.publish(MQTT_TOPIC_Y, f"{y_coord:.2f}")
            client.publish(MQTT_TOPIC_YAW, f"{yaw_angle:.2f}")

            # Publish target coordinates to MQTT
            client.publish(MQTT_TOPIC_TARGET_X, f"{target_coord_x:.2f}")
            client.publish(MQTT_TOPIC_TARGET_Y, f"{target_coord_y:.2f}")

            # Detect green ball and calculate angle
            ball_center = detect_green_ball(frame)

            if ball_center:
                ball_world_coords = apply_homography(H, ball_center)
                ball_x = int(ball_world_coords[0].item())  # Fix deprecated conversion
                ball_y = int(ball_world_coords[1].item())  # Fix deprecated conversion

                # Current position of the ball (newly detected)
                current_position = (ball_x, ball_y)

                # If this is the first detection, set previous_position equal to current_position
                if previous_position is None:
                    previous_position = current_position

                # Calculate the ball angle relative to marker 0's orientation (yaw_angle)
                relative_ball_angle = calculate_ball_angle(previous_position, current_position,
yaw_angle)

                relative_ball_angle = round(relative_ball_angle, 1)

                # Update previous_position for the next frame
                previous_position = current_position

                # Draw a circle on the ball in the image
                cv2.circle(frame, ball_center, 5, (0, 255, 0), -1)

                last_detection_time = time.time()

                # Publish green ball information
                client.publish(MQTT_TOPIC_EVENT_STATE, "1")
                client.publish(MQTT_TOPIC_EVENT_X, f"{ball_x:.2f}")
                client.publish(MQTT_TOPIC_EVENT_Y, f"{ball_y:.2f}")
                client.publish(MQTT_TOPIC_EVENT_THETA, f"{relative_ball_angle:.2f}")
            elif time.time() - last_detection_time <= 1:
```

```
                # Continue broadcasting the last known values if within 1 second of last detection
                client.publish(MQTT_TOPIC_EVENT_STATE, "1")
            else:
                client.publish(MQTT_TOPIC_EVENT_STATE, "0")

            # Draw red object coordinates without blue circles
            for i, (x_world, y_world) in enumerate(red_world_coords):
                x_world_float = x_world[0]
                y_world_float = y_world[0]
                red_image_coords = apply_homography(np.linalg.inv(H), (x_world_float, y_world_float))
                cv2.circle(frame, (int(red_image_coords[0]), int(red_image_coords[1])), 20, (0, 0,
255), 5)
                cv2.putText(frame, str(i + 1), ordered_points[i], cv2.FONT_HERSHEY_SIMPLEX, 1.0, (255,
0, 255), 2)

            # Draw blue obstacles with blue circles
            for x, y in blue_obstacles:
                cv2.circle(frame, (x, y), 20, (255, 0, 0), 5)  # Blue circles around blue obstacles

            # Publish program state to MQTT
            client.publish(MQTT_TOPIC_STATE, str(program_state))

        # Draw detected markers and additional information on the frame
        frame = cv2.aruco.drawDetectedMarkers(frame, corners, ids)

        '''
        if 30 in ids:
            marker_detection_start_time = None  # Reset timer if all markers are detected
        else:
            # Start the timer if it wasn't already started
            if marker_detection_start_time is None:
                marker_detection_start_time = time.time()

            # If markers have not been detected for 2 seconds, change to program_state = 2
            elif time.time() - marker_detection_start_time >= 20:
                program_state = 2
                client.publish(MQTT_TOPIC_STATE, str(program_state))
        '''

        # Display basic MQTT information on frame
        if H is not None:
            cv2.putText(frame, f"Drone: X={x_coord:.2f}, Y={y_coord:.2f}, Yaw={yaw_angle:.2f}",
                        (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
            if ball_center:
                cv2.putText(frame, f"Ball: X={ball_x:.2f}, Y={ball_y:.2f},
Theta={relative_ball_angle:.2f}",
                            (10, 60), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
```

```
            x_world_float = int(real_world_coords[0])  # Fix deprecated conversion
            y_world_float = int(real_world_coords[1])  # Fix deprecated conversion

            if not finished:
                cv2.putText(frame, f"Target: X={x_world_float:.0f}, Y={y_world_float:.0f}",
                            (10, 90 + i * 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)
            else:
                cv2.putText(frame, f"TARGET REACHED",
                            (10, 90 + i * 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)

        # Display the frame
        cv2.imshow('Drone Navigation', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

        # Check if 1 second has passed
        elapsed_time = time.time() - start_time

        if elapsed_time >= 1 and waiting_for_input:
            print("Press Enter to start the program after confirming the points are correct...")
            user_input = input()  # Wait for user input

            if user_input.lower() == 'q':
                break
            else:
                program_state = 1  # Change to start state after input
                client.publish(MQTT_TOPIC_STATE, str(program_state))  # Publish the updated state
                waiting_for_input = False  # Stop waiting for input


    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

Logbook

September 2nd

Started building the main function for drone navigation with ArUco marker detection and a basic MQTT setup. Had a straightforward video capture set up with cv2.VideoCapture(0) and a few placeholders for different program states (Idle = 0, Start = 1, Finish = 2). Set up an MQTT topic to

publish the initial program state, but only on a single line and without any logic for adjusting the state. Detection worked roughly, but I didn't have yaw angle calculations or homography applied yet. Could detect a marker, but no logic to act on it or any proper image coordinates.

Added a few lines for detecting red objects, but wasn't sure how to handle blue obstacles. Code didn't have the function to properly transition between program states, and navigation was missing. I wanted to eventually add an A* or TSP pathfinding algo, but no idea yet how to structure that around detected obstacles or order red points.

September 15th

Added detect_aruco_markers() and estimate_homography() functions so that I could recognize and orient the drone in relation to ArUco markers. I then added homography matrices into the code and set up last_valid_homography to save the last good homography estimation, which would get reused in frames where the estimation failed. This made detection more consistent, but it still wasn't smooth, especially when the markers went out of frame.

Began integrating MQTT topics for drone coordinates and yaw angle, but realized I needed to isolate the red and green object detections from ArUco-based position tracking. Hadn't included green ball detection yet, but set a placeholder for it to add later. Also started target_coordinate_calculation() but didn't get it to output usable coordinates or switch the program state yet.

October 3rd

Integrated calculate_yaw_angle() into the main function to get the drone's orientation relative to marker 0. Needed it for smoother path alignment, but it's not perfectly accurate yet. Also started detect_green_ball() and added code to calculate the angle between the ball's position and the drone's yaw—this came out cleaner than I thought. Set last_detection_time to keep track of green ball detections, which helped prevent lag and kept MQTT publishing steady for the ball coordinates. Finally implemented client.publish(MQTT_TOPIC_STATE, str(program_state)) to update the state, so it could change to the "Finish" state after completing the navigation.

Noticed that the frame's imshow preview had slight lag—considering a fix for it but might need to revisit time.sleep() timing within the code. It's working, but I want to optimise it to see if it can handle more complex situations.

October 14th

Updated red object detection so that it arranges red points relative to ArUco marker 30 and the final endpoint, which now can be input manually through the terminal. Finally put together a version of target_coordinate_calculation() that switches program_state when all points are visited. Added a TSP solution to optimize ordering, considering blue obstacles which are now detected separately with detect_blue_obstacles().

Added apply_homography() for both endpoints and red objects so the coordinates got mapped to real-world locations. This allowed me to avoid re-detecting markers on each loop, which improved performance a bit. Finally, I removed the old logic for blue circle visuals around red objects and swapped it with blue circles for blue obstacles. Frame display is more intuitive now.

October 29th

Current code feels almost complete but still needs refinement. Added previous_position for the green ball so I can calculate movement angle between frames more accurately. Integrated a function for publishing drone coordinates, yaw, and target coordinates via MQTT, and added a failsafe so the system reverts to the last detected ball position if no new one is found within a second. Now, green ball angles relative to marker 0 yaw get published as well, which allows for better tracking.

Did a major restructure on how MQTT topics are updated; added separate functions for each topic with time.sleep(0.1) for refresh timing. Kept homography application for both real and image coordinates when drawing red object and obstacle points in imshow(). Main program now transitions directly to the finish state on reaching the endpoint.