EEL6606 Project 1 Report

Terrance Williams

September 28, 2023

Contents

1 Introduction

EEL6606 is a course that serves as an introduction to aerial robotics. In this course, we learn the basics of Unmanned Aerial Vehicle (UAV) flight in terms of kinematics and dynamics as well as the basics of autonomous flight such as perception, path-planning, and flight controller programming. A project-based course, students are expected to complete three projects that demonstrate principles covered in class.

For Project 1, Robolink's CoDroneEDU platform was used to learn the basics of interacting with a UAV system. The aim for this project was to work through the reference materials provided by the company as well as perform basic movement demonstrations to understand both how the drone performs and its capabilities for custom applications.

2 CoDroneEDU Platform

The CoDroneEDU (also referred to as 'codrone') is an educational drone produced by Robolink, an American robotics company. It's a small UAV that fits in one hand, but it has an array of sensors and features to assist aerial robotics education. As seen by Figure 1, the codrone is a quad-rotor UAV meaning it's designed to use four propellers to achieve liftoff and perform subsequent piloting. The propellers are arranged in a planar, square pattern; there are two colinear sets of propellers near the front of the drone (red) and two towards the back (black). In the center of this arrangement is a controllable LED unit that can be used to communicate status updates to the human monitors. The system comes with two lithium-ion batteries with each providing about seven to eight minutes of consecutive flight time.

The codrone is equipped with seven sensors that provide an variety of data to the user. These sensors include: a color sensor, proximity sensors (front-range and bottom range), optical flow sensors, an accelerometer, a gyroscope, and a barometer for measuring pressure. The accelerometer is used to sense translational acceleration. The color sensor is used to detect colors, specifically those included on the provided color landing pads. The two proximity sensors are used to help prevent collisions, with the front range sensor preventing head-on collisions and the bottom range sensor keeping the drone at a certain relative height. Optical sensors are used to sense relative position as the drone moves, and the gyroscope is used to sense rotational movement such as change in heading/attitude. Additionally, the drone can also provide information to the user such as internal temperature and battery level. This information can be used to create conditionals in program, to be discussed in a later section. The CoDroneEDU tutorials include more information and tips on interacting with these sensors to create interesting applications.

[1]Image Source: https://www.robolink.com/products/codrone-edu

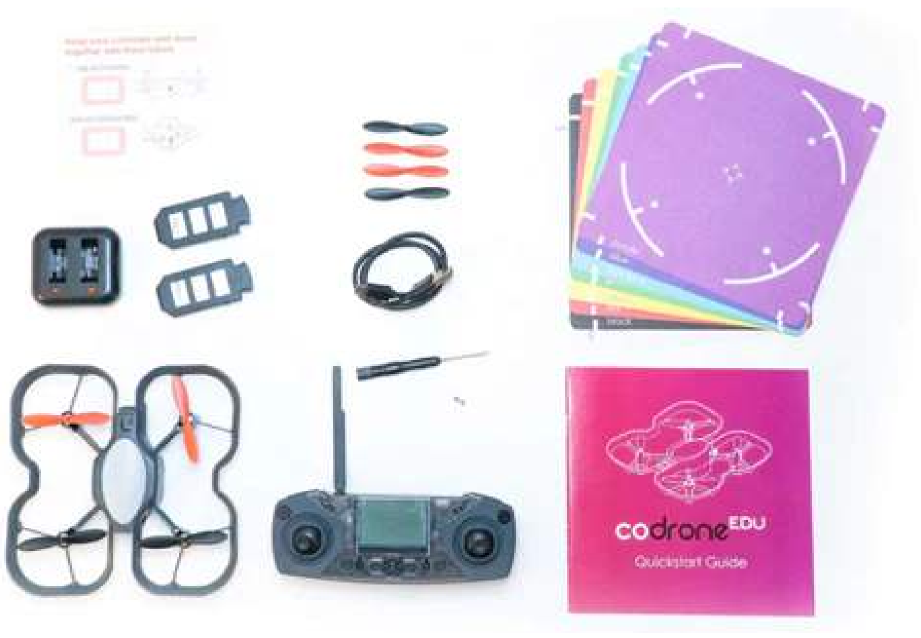


Figure 1: CoDroneEDU Kit[1]

Finally, the codrone comes with a remote controller that serves both as the controller for manual piloting and the connection point for programmatic control via the "LINK" mode. The vast majority of the project was spent in this mode.

3 Programmatic Interface

The CoDroneEDU has two forms of software-based control in terms of languages. The first, Blockly [1], is a visual programming language similar to MIT's Scratch. The user assembles programs using pre-defined blocks that fit together like puzzle pieces. Programming the drone this way is useful because the user is able to visually organize their program into functional blocks and connect sub-systems together to derive more complex behaviors. It's a great organization method for thinking and program design.

The other programming interface for the codrone is programming using Python [2]. While this method is not nearly as visual-based as Blockly, the programming library provided by Robolink allows the user to control almost every aspect of the drone including how it responds to button presses on the provided controller. So in lieu of visual organization, the user receives much more fine-grained control of the system, even allowing for system extension or modification if desired. Python

[2]Image Source: `https://learn.robolink.com/lesson/1-6-loops-junior-cde-blockly/`

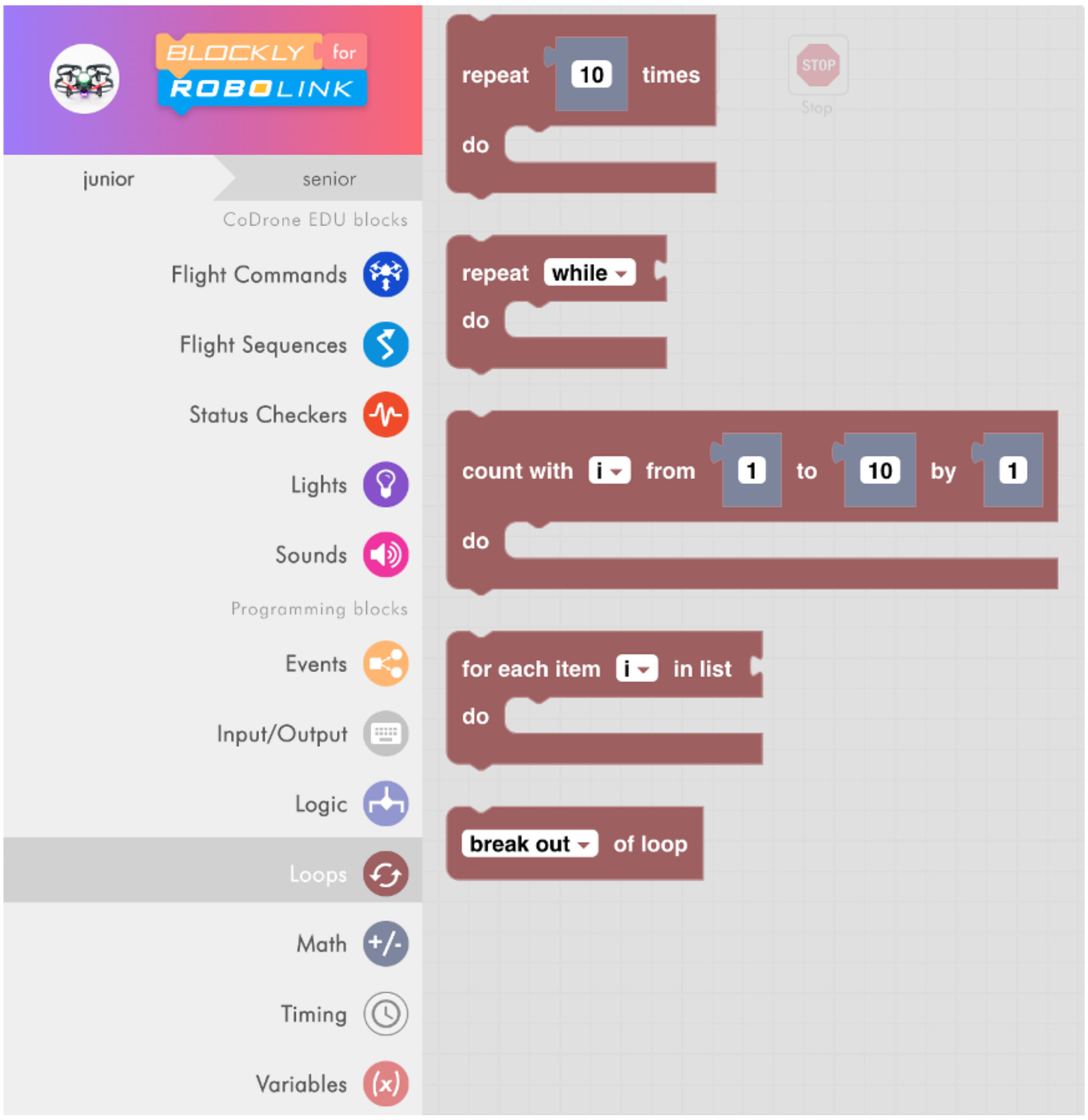


Figure 2: Example of the Blockly language[2]

was chosen as the language for this project, and it is the language that will be used for subsequent projects as well.

3.1 Basic Movement

In terms of movement, the codrone is omni-directional, a function of the robot's quad-rotor design. By modifying the rotation speed on any combination of the four rotors, the UAV can move forward, backward, left, right, or diagonally. It can also rotate left or right and adjust its hovering height. Movement commands have both direction and power components. For example, to move the drone forward, one sets the pitch parameter using the *set_pitch*() function. It accepts a value from 0 to 100 that represents the percentage of movement power. To change direction, the number can be passed as a negative value. Positive values move the robot forward, and negative values move it backward. A similar configuration is used for roll, yaw, and throttle movements. However, these functions simply set the parameters; the robot will not move until it's told to, so the user must provide a call to *move*() and pass in a movement duration value. Using these functions, the user can have to drone fly in numerous shapes, including a square, triangle, sinusoid, spiral, and many others.

The typical program flow suggested by Robolink is to do the following. First, the user creates a Drone object in Python that is used to connect to the drone and pass commands to it. Next, this Drone object connects the drone to the Python program through the controller (which is "LINK" mode while plugged into the computer via microUSB). Now, the drone will be able to execute commands sent from the computer. From here, the user has the drone takeoff and hover in mid-air; this step is necessary because other movement commands will not work if the drone is grounded. The user can then send the robot through their desired flight patterns and movements while the battery has enough power. Finally, the user lands the drone with the *land()* command, and, in the most important step, *disconnects the drone-computer connection* via the *close()* command. The last step is emphasized multiple times throughout the tutorials; if it is not performed, then the physical drone will still think it's connected to the computer once the program ends. On the next program run, the computer and the drone will not be able to form a connection. If this happens, the drone can be manually reset (unplug the battery and reconnect it). Of course, not having the problem in the first place is preferred, especially since there are some scenarios where the drone may be too high to reach.

3.2 Programming a Drone Context Manager

The pair/disconnect cycle is required for any Python program that controls the codrone. One can easily imagine a scenario in which the disconnect portion is neglected by mistake, leading to the problems discussed previously. Additionally, if the program fails at some point before the *close* command, the program will exit without performing the disconnect action, leading to the same problem. Part of the project was used to create a more robust system for interacting with the CoDroneEDU.

First, a `try-except-finally` model was used, whereby the entire program script is wrapped by Python's `try` block. If an error occurs (an exception is raised), the program 'catches' the exception and allows the user to handle it accordingly. The purpose of the `except` block in this context is to allow the user to document what went wrong. The most relevant part of the model, however, is the `finally` clause. This clause executes whether an exception occurs or not. It is used for clean-up actions in Python programs or to ensure some behavior occurs. Originally, this clause was used to perform the necessary disconnect action. The program structure looks like this:

```
try:
        <Code commands that control the drone>
except Exception:
        # Caught an error
        <perform actions>
finally:
        <perform cleanup actions>
        <disconnect drone>
```

While the above structure solved the drone connection problem when faced with an execution error, there was still the possibility that the user could forget to include the clause itself. Also, having complex program wrapped in such a block seemed inelegant with respect to readability, so another method was pursued.

Python has a programming pattern called a *context manager* [3][4]. Context managers are blocks that handle the setup and tear down aspects of programming actions for the user. Essentially, they are reusable "try-except-finally" blocks. Traditionally, they are used to handle file operations because the structure automatically handles closing the file for the user, an operation that, similar to the drone connection, has undesired and sometimes dire consequences if forgotten. A context manager is evoked using the *with* keyword:

```
with open(sample.txt) as f:
        f.operation1
        f.operation2
        ...
# <file automatically closes here>
```

Once the program context leaves the *with* block, the clean-up actions occur. What's useful about this structure is the cleanup occurs whether the program was successful or not; errors will still trigger the clean-up just ss it did with the *finally* clause. This seemed to be the more elegant solution, so the next step was to determine how to use a context manager with the codrone tooling.

3.2.1 Creating a Context Manager

Context managers have two parts that the programmer must include: `__enter__` and `__exit__` [5]. The former is responsible for setting up the desired context, including returning a reference to an object the user desires to interact with within the context (ex. the 'f' object in the earlier example). The latter defines the clean-up actions and also handles exceptions. As long as these two methods are included in the structure used in the *with* statement, a valid context manager is present.

To use the pattern with the CoDroneEDU Library, I extended the `Drone` class to a new class, `TDrone`. This class inherits from the `Drone` class, meaning it has access to all of the methods and attributes from `Drone` while also having additional capabilities that the user desires. The `TDrone` class is the `Drone` class with the addition of `__enter__` and `__exit__` as well as a modified `__del__` method, the method called upon garbage collection (when the object is fully deleted). In the original class, this method would call `close`, but since that is now done within `__exit__`, there is no need to do so a second time, so the new version:

```
def __del__(self):
        pass
```

simply passes in order for the garbage collection to occur. If desired, the user could place a `print` statement instead that signals the object is about to be collected.

For the other modifications, the `__enter__` method performs the pair action, and the `__exit__` method lands the drone and then closes the connection. It was discovered that landing the drone first is important because otherwise the drone remains suspended in the air if an error occurs. Landing the drone allows for a consistent, controlled exit action.

To use the context manager, one writes the following:

```
with TDrone() as drone:
        drone.takeoff()
        ... # Other actions here

# Exit the context
# drone.__exit__() is called, landing the drone
# and closing the connection.
```

Now, the entire program can be written within the `with` context, or a program can be written in an outside function and called within the context like so:

```
def main(drone):
        drone.action1
        drone.action2
        ...
        drone.final_action


with TDrone() as my_drone:
        main(my_drone)
```

This keeps the code readable and automatically handles drone connection and disconnecting actions, so the user will never forget to do so.

3.2.2 Testing

In order to test that the context manager works, I created a simulation of the pairing/disconnect actions. I overwrote the relevant pairing and disconnecting methods so that instead of performing the actions with the drone, the two methods print messages to the screen. This was done in order to get ensure that the context manager would work before adding complex Bluetooth connection to the system. Once I was certain that the context manager would work as desired, the custom `pair` and `close` methods were deleted, allowing the original implementations to take their place. This testing method was viable due to the fact that my custom drone class inherits from Robolink's class, allowing access to all of the original class' infrastructure.

The next test focused on ensuring the context manager would work even when the program has an error. To test, I purposefully introduced errors into the system. In one test, the general `Exception` was manually raised while the program was running, and in another test, a non-existing attribute was accessed (an `AttributeError`). Both scenarios resulted in the desired behavior; the drone made a controlled landing and disconnected from the Python program.

3.3 Using Sensor Data

With the addition of the context manager, the user can now focus more intensely on the desired flight actions. One important aspect of planning such actions are conditionals, programmed behaviors that occur only when some condition or set of conditions is met. The CoDroneEDU Library provides numerous methods that allow the user to access sensor data such as internal temperature, barometric pressure, gyroscope data, etc. Any or all of these data points can be used to determine the drone's next move, leading to more complex behaviors.

As an example, one can use the codrone's range sensors to ensure the drone remains some distance away from a wall or maintains a minimum relative altitude. Another example is battery-based LED color signal, where above, say, 75% the drone LED is set to green, between 50 and 75% the LED is yellow, and below 50% the LED is red. The user could write a program that periodically polls the battery level and sets the LED color accordingly to provide a visual, qualitative indicator of remaining battery life.

While the sensors included aren't those one may find in more complex, long-running systems, Robolink includes enough data variety that users can discover creative, non-conventional uses. The tutorials, for example, use sensor data to implement codrone-versions of common games such as "Red light, Green light" [6] or color-based codrone pianos [7].

4 Flowchart(s)

Figure 3 depicts the basic system diagram for the CoDroneEDU setup. The drone and the remote controller have a communication line through Bluetooth that allows the remote to send commands to the drone and allows the drone to send sensor data to the controller. The computer communicates with the controller via microUSB and includes the aforementioned context manager code for ease of operation.

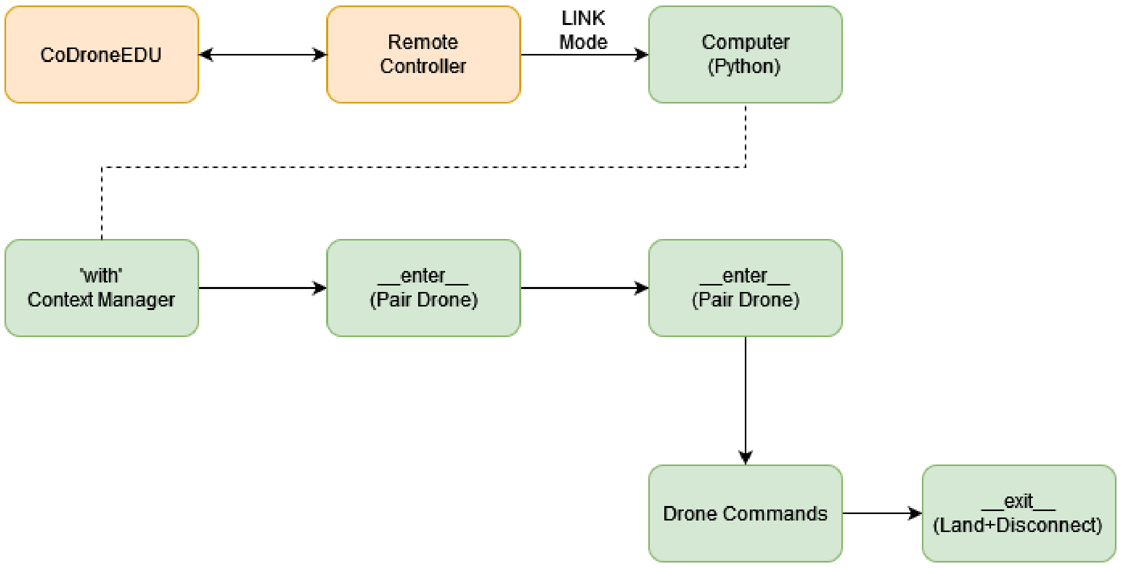


Figure 3: System Diagram of CoDrone Environment

5 Conclusion/Future Work

For this project, the CoDroneEDU UAV platform was used to learn the basics of practical aerial robotics. An alternate programmatic interface was developed and used to remove cognitive load from the CoDrone program development process. Additionally, basic movement parameters such as roll, pitch, yaw, and throttle were modified to create various flight trajectory shapes, and the drone's sensors were used to develop conditional behavior for more complex actions.

In future projects, I aim to use the drone's features in a creative, fun application such as emulating a baseball game where upon receiving a numeric input, the drone "runs" a set amount of bases. This would provide an interesting challenge because various flight parameters will need to be used, and one could introduce complexity by adding navigation rules based on "outs."

References

[1] Robolink, *Blockly with CoDrone EDU*, https://learn.robolink.com/course/blockly-with-codrone-edu/, Accessed: 18 September 2023, Robolink.

[2] Robolink, *Python with CoDrone EDU*, https://learn.robolink.com/course/python-with-codrone-edu/, Accessed: 18 September 2023, Robolink.

[3] Python Software Foundation, *3. Data model: With Statement Context Managers*, https://docs.python.org/3/reference/datamodel.html#context-managers, Accessed: 19 September 2023, Python Software Foundation.

[4] Python Software Foundation, *8. Compound statements: The with statement*, https://docs.python.org/3/reference/compound_stmts.html#with, Accessed: 19 September 2023, Python Software Foundation.

[5] L. P. Ramos, *Context Managers and Python's with Statement*, https://realpython.com/python-with-statement/#creating-custom-context-managers, Accessed: 19 September 2023, Real Python.

[6] Robolink, *2.1: LED*, https://learn.robolink.com/lesson/2-1-led-cde/, Accessed: 3 September 2023, Robolink.

[7] Robolink, *3.8: Color Classifier*, https://learn.robolink.com/lesson/3-8-color-classifier-cde/, Accessed: 6 September 2023, Robolink.

A Code

```python
from codrone_edu.drone import Drone
import time


class TDrone(Drone):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # self.connected = self.isOpen()

    def __enter__(self):
        # Pair the drone
        self.pair()
        time.sleep(0.2)
        return self

    def __exit__(self, exc_type, exc_value, exc_tb):
        self.land()
        self.set_drone_LED(255, 0, 0, 100)
        # Shutdown Connection
        self.close()
        # print(f"Connected?: {self.connected}")

        # Print Errors
        if exc_value is not None:
            print(exc_type, exc_value, exc_tb, sep='\n')

    """
        Testing Context Manager via Overloads

        def open(self, portname=None):
            print("OPEN: Opened!")
            self.connected = True

        def close(self):
            print("CLOSE: Closing CoDroneEDU Connection.")
            self.connected = False
    """

    def __del__(self):
        # Overload to prevent close from being called twice.
        print(f"TDrone: Deleting.")

    def fire_start(self):
        self.takeoff()
        self.hover()
        ready = False
        self.set_drone_LED(0, 0, 255, 100)
        while not ready:
            key = input("Press 's' to begin: ")
            if key.lower() == 's':
                print("Beginning Flight.")
                ready = True
                self.set_drone_LED(0, 255, 0, 100)
                time.sleep(0.1)

```

```python
from tjdrone import TDrone


def drone_movements(drone: TDrone):
    # Parameters
    pitch_power = 20
    roll_power = 20
    yaw_power = 50
    throttle_power = 30

    """Cycle through basic flight motions using the CoDroneEDU"""
    # Forward and Backward
    drone.set_pitch(pitch_power)
    drone.move(2)
    drone.hover(2)
    drone.set_pitch(-pitch_power)
    drone.move(1.5)
    drone.hover(2)

    # Left and Right
    drone.set_roll(roll_power)
    drone.move(1.5)
    drone.hover(2)
    drone.set_roll(-roll_power)
    drone.move(1.25)
    drone.hover(2)

    # Rotations
    drone.set_yaw(yaw_power)
    drone.move(2)
    drone.set_yaw(0)
    drone.hover(2)
    drone.set_yaw(-yaw_power)
    drone.move(2)
    drone.hover(2)

    # Vertical Movements
    drone.set_throttle(-throttle_power)
    drone.move(2)
    drone.hover(2)
    drone.set_throttle(throttle_power)
    drone.move(1)
    drone.hover(2)


if __name__ == "__main__":
    with TDrone() as my_drone:

        # Set parameter(s)
        my_drone.set_trim(-5, 0)
        my_drone.set_drone_LED(255, 0, 0, 100)

        # Begin flight on key-press
        my_drone.fire_start()

        # Run Basic Movements
        drone_movements(my_drone)

```

```python
from tjdrone import TDrone
import time


def exec_patterns(drone: TDrone):
    patterns = [drone.square, drone.triangle,
                drone.circle, drone.spiral,
                drone.sway]

    for func in patterns:
        drone.set_drone_LED(255, 0, 0, 100)
        ready = False
        while not ready:
            key = input("Press 's' to begin: ")
            if key.lower() == 's':
                print(f"Beginning Flight Pattern: {func}")
                drone.set_drone_LED(0, 255, 0, 100)
                drone.takeoff()
                drone.hover()
                ready = True
                time.sleep(0.1)
        if func == drone.spiral:
            func(speed=30, seconds=3)
        else:
            # Default Parameters
            func()
        drone.land()
        time.sleep(0.1)


if __name__ == "__main__":
    with TDrone() as my_drone:
        exec_patterns(my_drone)

```

EEL6606 Project 2 Report

Terrance Williams

October 31, 2023

Contents

1 Introduction

In the first project for EEL6606, Robolink's CoDrone EDU was examined, testing its flight capability and sensor features. The project also resulted in the creation of a new class to interface with the drone using a Python Context Manager for improved runtime safety.

Projects 2 and 3 seek to use the tools and knowledge gained in Project 1 to create an applicative use for the drone. Specifically, the aim is to create a means of emulating base running in the game of baseball. The goal for this project is to program the drone to detect a given base and navigate to the next one.

2 Drone Baseball

2.1 Definition

As previously mentioned, the purpose of the remaining two projects is to begin to use the drone's movement abilities and sensors to complete an objective, specifically navigating four way-points or "bases" similarly to that in baseball. However, the actual game of baseball has many complex rules and motions, so the objectives of the project will be adjusted to a simplified version.

By the end of both projects, the drone should be able to identify and begin from the Home base, receive an input for the number of bases to run based on 'hit', and round the bases accordingly, stopping whenever it reaches the Home base. There is no assumption of stealing bases or traveling in the clockwise direction; the drone must only navigate in the counter-clockwise base order, keeping track of the base it currently occupies or moved from.

For Project 2 specifically, the drone traversing between two base structures is considered a satisfactory result. The full four-base environment will be constructed in Project 3.

2.2 Sensors Used

The detection of bases will ultimately use two of the CoDrone's sensors. For detecting the presence of a base, the drone's bottom range sensor is used. To detect the specific base the drone has landed on, color detection via the drone's color sensors will be used.

3 Bases

3.1 What Constitutes a Base?

For the purposes of this project, a base is a flat, wide region that has significant elevation in comparison to the "floor" on which the bases sit. For example, if we consider the floor of a room, elevated cardboard boxes or two square-shaped chairs would provide enough elevation to serve as bases.

Because of the lack of robust localization on the CoDrone EDU, the drone experiences noticeable uncertainty in its motion. As a result, objects with large surface areas are necessary to use to provide the drone with a wide area to land. The trade-off, however, is that takeoff points will be inconsistent, introducing the possibility that the drone can miss another base when traversing the field.

3.2 Arrangement

Since the idea is to emulate a baseball game, the bases will ideally be in a diamond formation. However, during testing it was determined that the results of diagonal movement for the drone is not as predictable as square-like motions. Since this project is solely concerned with ensuring the drone can identify a base, the bases are assumed to be arranged in a rectangular formation, allowing for tests using pure pitch or pure roll movements.

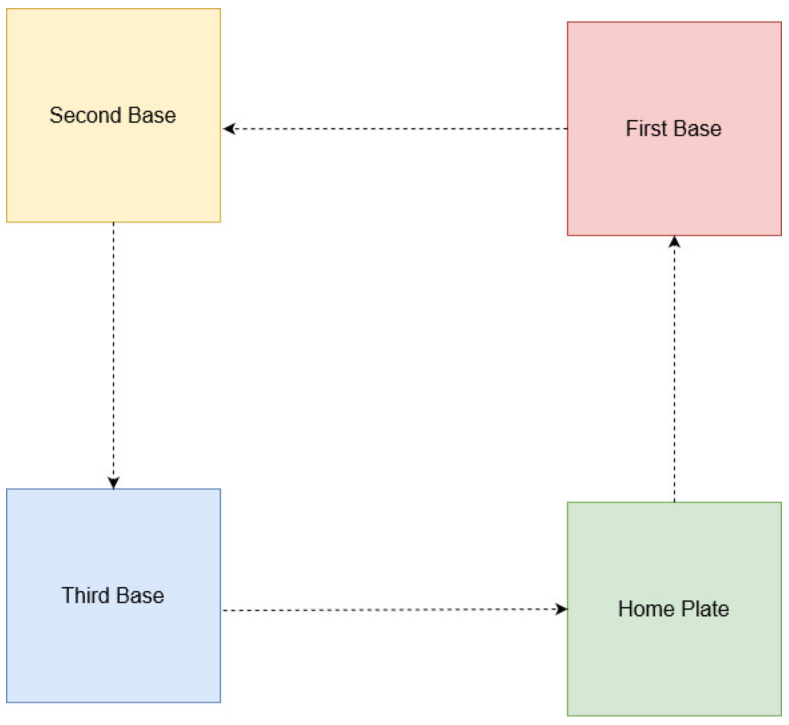


Figure 1: Square Base Arrangement

3.3 Identifying a Base

In baseball, there are four bases: Home, First, Second, and Third. In order for the drone to know what base to navigate to, it must first know what base it currently occupies. To achieve this, the baseball program must keep track of the drone's current base. In the program, each base is represented numerically, with the First, Second, and Third Bases all being assigned their respective numbers (1, 2, 3), and the Home Plate assigned to 0. Function references used for this project may be found in [1]

The program determines what move commands to send the drone based on its current base. The following table lists the movement directions based on the drone's current base:

Current Base	Movement Direction
Home	Forward
First	Left
Second	Down
Third	Right

Table 1: Base-Movement Associations

For testing purposes, the base movement program takes in user input to determine the drone's current base, allowing for the two test bases to be shuffled accordingly.

To actually detect a base, this project utilizes the drone's bottom range sensor to detect changes in relative height. The bases are elevated platforms such as cardboard boxes that have a known height (ex. 20 cm). When the drone performs its takeoff action to become airborne, it descends until it is within a relative height range compared to the top of the base. Currently, this range is from 15cm to 20cm. The program records this relative height and commands the drone to move in the specified direction (see Table 1).

The drone then moves in the specified direction[1] Once the drone leaves the base, its bottom range sensor then detects the change in relative height from the base to the floor. Because the difference in this change is so great, it is considered a relative height switch. Upon crossing over another base, this switch will occur again, allowing the program to determine that a base has been reached. Because this method of detection can be sensitive to sensor values, a threshold is used to filter for significant changes in relative height. This threshold is simply the height of a base (20cm).

[1]The program assumes there is indeed another base in the target direction.

In other words, to detect a base the program first stores the bottom range sensor reading when the drone is directly hovering over its current base. It continues to poll the bottom range sensor, calculated the difference in relative height between the stored reading and the most current. Once this difference exceeds the threshold (i.e. the drone leaves the base), the first switch is recorded and this relative height is stored as the value. The drone continues to move until it reaches another base, at which point the second switch occurs. After two switches, the drone is then told to land.

Ideally, the drone would then run color detection to ensure it has landed on the proper base; this feature will be implemented in the next project.

4 Performance Analysis

In terms of performance, the program's logic is sound. Upon prompt, the drone will perform a takeoff from the base and slowly descend such that its relative height is within the specified range. Upon reaching a new base, it lands as expected. It also flies in the base-specific directions, taking stock of its current base and move in the direction toward the next one.

The problem, however, comes as a result of the relatively unintelligent navigation algorithm. Because the drone flies in a specified direction until it reaches the expected height change, if it happens to miss the base, it will continue to fly in said direction, potentially landing on another object of similar elevation or even colliding with an obstacle. Currently, there is no way for the drone to actively seek the characteristics specific to a base compared to other elevated objects, nor is it able to course-correct upon deviation because that would require foreknown knowledge of the physical locations of the bases.

One approach attempted in order to correct this problem was the use of the drone's way-point-setting ability, storing the physical locations of the bases via a calibration process. The drone, in theory, could then travel to the stored coordinate value if no base is detected within a specified time. However, the results were unpredictable; the drone would fly in unexpected directions during trials. More research will have to be done in order to determine if this method could be viable.

Direction-wise, the drone performed perfectly, moving in the correct direction toward the next base in all trials. For the next project, work will be done to first confirm the performance still holds in the full-game setting and then subsequently test performance for diagonal movements.

5 Flowchart

Included below in Figure 2 is the flow of the program for testing base navigation.

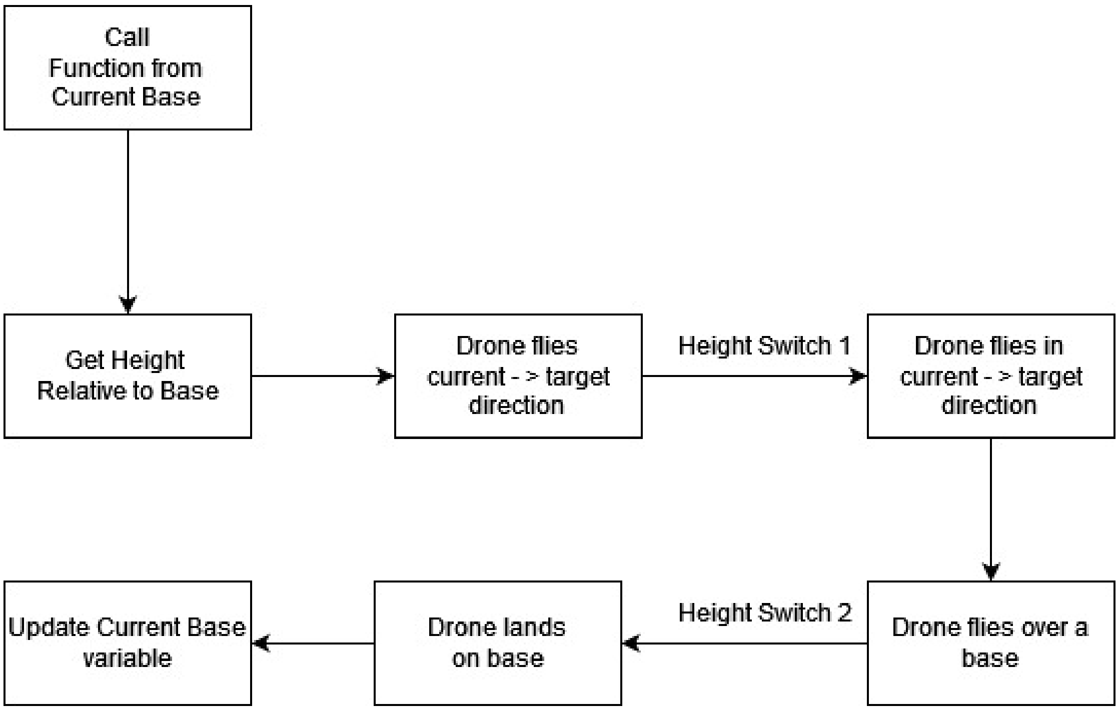

Call Function from Current Base
Get Height Relative to Base
Drone flies current - > target direction
Height Switch 1
Drone flies in current - > target direction
Drone flies over a base
Height Switch 2
Drone lands on base
Update Current Base variable

Figure 2: Base Navigation Program Flow

6 Conclusion

In this project, the performance objectives and environment for a "baseball" emulation were defined. The CoDrone EDU was used to test base identification, navigation, and overall program progression. For the next project, color detection will be integrated with the system to perform targeted base identification, and the motion patterns of the drone are to be adjusted to diagonal motions. Finally, the full four-base system will be assembled for demonstration.

References

[1] *CoDrone EDU Library Reference*, Robolink, Aug. 2023. [Online]. Available: https://docs.robolink.com/docs/codrone-edu/python/reference/library/.

A Code

```python
from tjdrone import TDrone
import time


# %% Constants
PITCH_POWER, ROLL_POWER, THROTTLE_POWER = 15, 15, -25  # power and directions: forward, right, and down
MOVE_TIME = 0.1
SLEEP_TIME = 1.5
SWITCH_DIST_THRESH: float = 20.  # (cm); Relative height difference that denotes a change from or to a base.
BTMRANGE_SENSOR_UNIT = 'cm'
SWITCH_DIST_THRESH: float = 20.  # (cm); Relative height difference that denotes a change from or to a base.
MIN_RELATIVE_HEIGHT, MAX_RELATIVE_HEIGHT = 15, 20  # (cm)
HEIGHT_SWITCHES = 2  # Number of times relative height must switch (i.e. exceed difference threshold)


# %% Function Definitions
def move(current_base: int, drone: TDrone) -> int:
    drone.set_drone_LED(255, 0, 255, 100)
    """Moves from current base to next base"""
    dist_switch = 0

    # Ensure that the drone is on the proper current base.

    '''
        Movement logic. ASSUMES ONLY TRANSLATIONAL MOVEMENT (for now).
        HOME: Move forward to Base 1
        Base 1: Move to the left to Base 2
        Base 2: Move backward to Base 3
        Base 3: Move to the right to HOME
    '''

    drone.takeoff()
    drone.hover(SLEEP_TIME)
    time.sleep(SLEEP_TIME)
    drone.set_throttle(THROTTLE_POWER)
    # Get initial bottom range value
    curr_dist = 0
    while not MIN_RELATIVE_HEIGHT < curr_dist < MAX_RELATIVE_HEIGHT:
        curr_dist = drone.get_bottom_range(unit=BTMRANGE_SENSOR_UNIT)
        drone.move(MOVE_TIME)
    else:
        # reset parameters
        drone.hover(SLEEP_TIME)
        # time.sleep(SLEEP_TIME)
        # Set new movement params
        if current_base == 0:
            drone.set_pitch(PITCH_POWER)
            drone.set_roll(ROLL_POWER)
        elif current_base == 1:
            drone.set_pitch(PITCH_POWER)
            drone.set_roll(-ROLL_POWER)
        elif current_base == 2:
            drone.set_pitch(-PITCH_POWER)
            drone.set_roll(-ROLL_POWER)
        elif current_base == 3:
            drone.set_pitch(-PITCH_POWER)
            drone.set_roll(ROLL_POWER)
        else:
            # If somehow the current_base is invalid.
            raise ValueError(f"Invalid current base value: {current_base}")

    # Move until the drone reaches another base (two huge rel. height changes)
    while dist_switch < HEIGHT_SWITCHES:
        drone.move(MOVE_TIME)
        # time.sleep(MOVE_TIME)
        next_dist = drone.get_bottom_range(unit=BTMRANGE_SENSOR_UNIT)
        print(f'[move] Bottom-Range Reading: {next_dist}')
        if abs(next_dist - curr_dist) >= SWITCH_DIST_THRESH and curr_dist > 0:
            dist_switch += 1
            print(f'[move] Relative Height switch no. {dist_switch} from {curr_dist} to {next_dist}')
            curr_dist = next_dist
    else:
        print('Distance-switching trips met.')

    # Land the drone
    drone.hover(MOVE_TIME)
    drone.land()
    print("[INFO] Landing.")
    while drone.get_bottom_range(unit=BTMRANGE_SENSOR_UNIT) > 0:
        time.sleep(SLEEP_TIME/4)
    else:
        print("[move] Landed.")
    time.sleep(SLEEP_TIME)
    # Change LED Color
    drone.set_drone_LED(0, 255, 0, 100)
    return True


with TDrone() as my_drone:
    done = False
    while not done:
        base = input('Insert a number: ').lower()
        if base == 'q':
            done = True
        else:
            # noinspection PyBroadException
            try:
                base = int(base)
            except Exception:
                print("Invalid entry.")
            else:
                assert move(base, my_drone)

```

EEL6606 Project 3 Report

Terrance Williams

November 20, 2023

Contents

1 Introduction

This report details the work done for Project 3 of EEL6606 Aerial Robotics. Specifically, it discusses the work continued from the previous projects. In Project 1, the educational drone, CoDrone EDU[1] was demoed, leading to familiarity and comfort with using and programmatically interfacing with the platform. Project 2 saw the beginning of the Drone Baseball project in which the goal is to have the drone round four bases in a baseball-reminiscent way. At the end of Project 2, the drone was able to identify when it has flown over a base and subsequently land on said base.

There were problems with the drone behavior, however. The Codrone experiences drift during takeoff, affecting its ability to sense the bases' relative heights. It also has inconsistent landings. In multiple runs, the drone will land on a variety of areas on the base's surface, thereby disturbing the trajectory for the next base traversal. This project aims to remedy this behavior. The goals for Project 3 are to:

- Develop color-dependent base identification
- Implement multi-base traversal
- Correct the takeoff drift
- Correct inconsistent landings

2 Color-Based Identification

The first goal for the project was to develop a color-oriented base identification method for the drone. To do so, the drone's color sensing system was tested. Initially, the aim was to have the drone hover over a given base and determine if said base is the correct target. However, it was found through initial testing that the Codrone's color sensors do not work while airborne; the drone must be grounded and stable for approximately 2 seconds for them to activate. This behavior means each base must be landed on to verify its identity.

Once this was determined, the color detection training data was created by following the company-provided tutorial [1]. Regarding physical implementation, while the Codrone kit comes with colored pads to use, they were not large enough to be used to cover the top surface of the base. Therefore, colored construction paper was used as the color model source. This paper is then used to cover the bases' top surfaces.

After training the color model, the base colors were chosen. The following mapping was chosen: Programmatically, this association is stored using a dictio-

[1]Also referred to as 'Codrone' in this report

Color	Base
Green	Home
Red	First
Yellow	Second
Blue	Third

Table 1: Color-to-Base Mapping

nary whose keys are the string names of the colors and whose values are both the numeric mappings discussed in the Project 2 Report[2] and the RGB values of the color (for visual LED changes). After landing on a base, the drone takes a color sample from its two color sensors. If both sensors return the same color and said color is a key in the dictionary, then the detected base number is returned and checked against the target base value. If the two are equivalent, the move was successful. Otherwise, an exception is raised. By incorporating color detection the program can now verify if it is on the correct starting base upon beginning a move and on the correct target base upon landing.

2.1 Base Construction

Each base is a 14in $\times$ 14in $\times$ 14in cardboard box whose top surface is covered with colored construction paper of the relevant base color. The entire base area is 55in $\times$ 55in, leaving about 40in[3] of space between each base center when arranged in a square pattern.

3 Multi-base Traversal

The next component of the project was the base traversal function. Rather than create a considerably long function that checks many cases (target base vs. current base), the movement function is split into two smaller functions. The first, *move* is responsible for only moving the drone from the current base to the next base in the arrangement. The second function, *move_bases* takes the drone's current base number and the amount of bases to traverse (based on user-provided hit type such as single, double, triple, or home run) and determines how many times *move* is called. This approach results in a more simplified method to achieve the desired complex movement. When testing, the drone attempts to perform the moves as expected, but the inconsistent takeoffs and landing result in movement error. As a result, the project shifted to the final two goals of solving these inconsistencies.

[2]{0: Home, 1: First, 2: Second, 3: Third}

[3]This value accounts for variance in box tolerance.

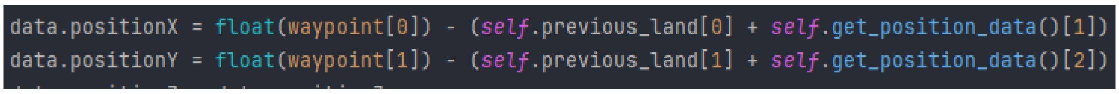
4 Waypoints

The solution to the motion drift problem utilizes the Codrone's ability to set and navigate to waypoints. The aim was to set target points for each base and move to said point once the drone realizes it has reached a base. Ideally, this method would result in the drone being "pulled" toward the target base's the center. Initially, the program required the that user perform a calibration at the start of every game, a procedure which proved tedious for testing. To remedy this, I instead only perform calibration on the first ever run of the program, saving the waypoint values in a JSON file and loading them upon subsequent runs. However, even that was optimized, electing to define the waypoint values explicitly based on the arrangement spacing and adjusting via trial and error.

4.1 How CoDrone EDU Waypoints Work

It is important at this point to discuss how the Codrone's waypoints are implemented and how it handles positioning in general. Initially, it was assumed that waypoint navigation is based solely on relative positioning, where each waypoint is simply the drone's current position from takeoff. Initial tests supported this, storing relative positions of each waypoint created from multiple takeoff points.

Later, it was discovered that while *setting* a waypoint is done solely from relative positioning, moving to a waypoint is not. To assist debugging, the source code to the *Drone.set_waypoint*, *Drone.goto_waypoint*, *Drone.land*, and *Drone.get_position_data* functions were analyzed. From *Drone.goto_waypoint*, I learned that the drone keeps track of its previous landing position. When this method is called for a given waypoint (created relatively), the displacement required to reach the drone is calculated terms of absolute position from the drone's first takeoff point. See the excerpt in Figure 1 to see how the waypoint displacement is calculated in the code.



```
data.positionX = float(waypoint[0]) - (self.previous_land[0] + self.get_position_data()[1])
data.positionY = float(waypoint[1]) - (self.previous_land[1] + self.get_position_data()[2])
```

Figure 1: Excerpt from the *goto_Waypoint* method [2].

The result of defining the waypoint navigation as such is that all desired waypoints are to be defined *before* the drone's first landing. This proved to be a problem in this project's use case because the drone has to land in on each base in order to perform manual calibration and because each waypoint is defined relative to the previous base.

In order to solve this problem, I added my own landing method to the *TDrone* class created in Project 1. The method calls the company-provided landing method and then resets the landing tracking variable to $[0, 0]$. As a result, each subsequent flight assumes its takeoff point to be the origin, such that the displacement formulas are effectively

```
data.positionX = float()waypoint[0]) - self.get_position_data()[1]
data.positionY = float()waypoint[1]) - self.get_position_data()[2]
```

This method results in truly relative waypoint navigation. Additionally, because the landing function is defined under a different name than the default method (*land_reset* vs *land*), the original behavior is still accessible if desired.

4.2 Correcting the Navigation

As stated previously, waypoints are set relative to the drone's takeoff position. This allows the base centers to be defined as waypoints where each point is relative to the base that came before it. So, the waypoint to First Base is defined relative to Home, Second Base is defined relative to First, Third to Second, and Home to Third. When the drone reaches the next base, the *Drone.goto_waypoint* function is called for it, adjusting the drone's position to the base's center. Ideally, because the drone lands in the same place upon every detection, the navigation from base to base should be both more accurate and precise.

Similar reasoning is used to correct the takeoff drift. Because the drone's landing is now completely relative via the custom method, each takeoff point is its own origin. Therefore, this origin can be passed as a waypoint upon takeoff. A new takeoff method, *TDrone.relative_takeoff* was defined, first calling the native *Drone.takeoff* function and then passes the origin $[0, 0, 0]$ as a waypoint. The drone then navigates to the original x, y values from its takeoff point[4]. The z value appears to be ignored.

The result of these two operations are much more consistent base runs. The drone now begins closer to the desired start point, and lands near the base center much more frequently as desired.

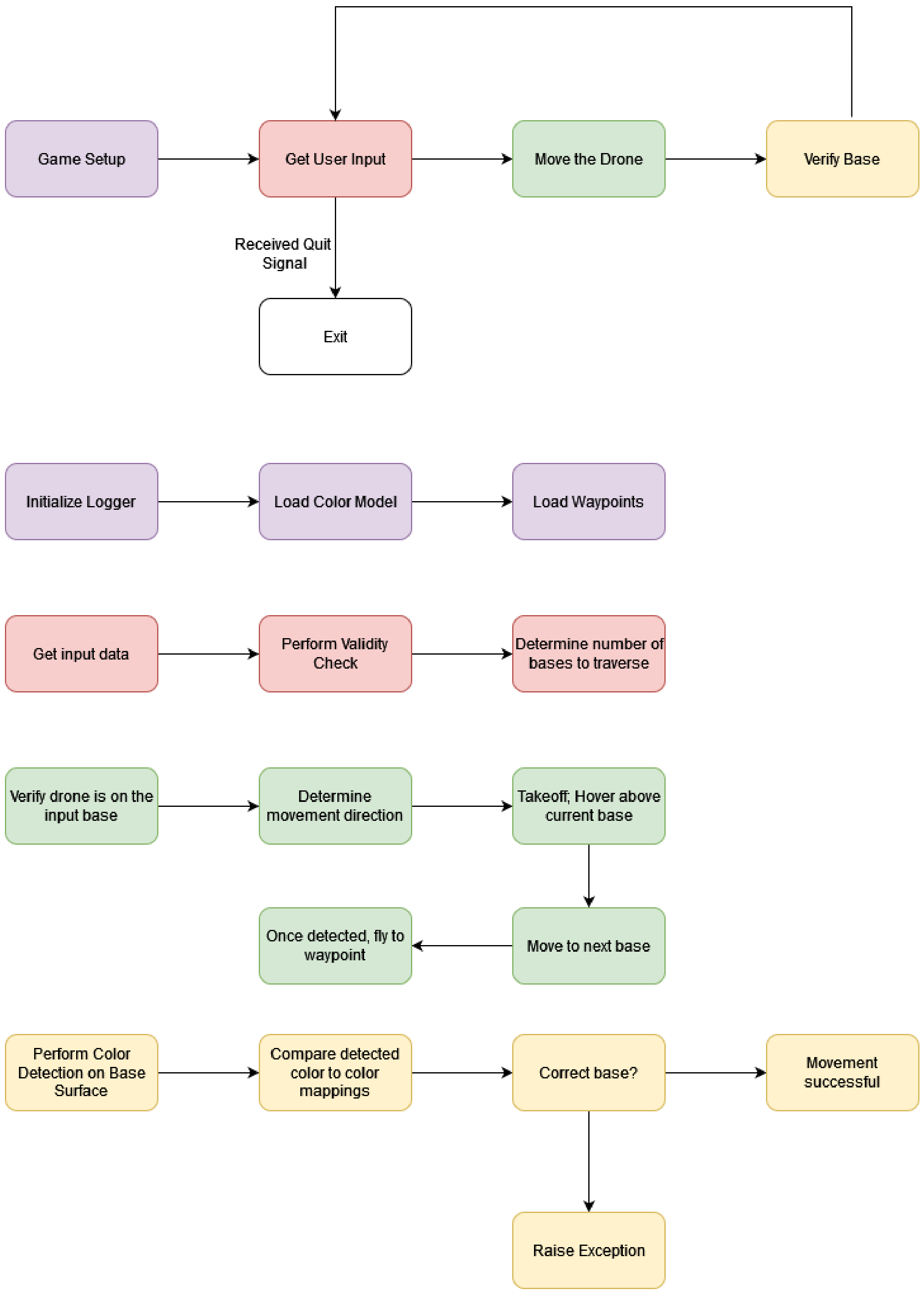
5 Program Flow and Results

In summary, the flow of the program is as follows. First, the setup is done, loading the color detection model, setting up a logger for output tracking, and loading the predefined waypoints. Upon completion, the drone then uses its buzzer to play the well-known baseball theme after which the program enters a *while* loop in which it continuously awaits user input. The user inputs a hit value, and the program

[4]Rather, the drone navigates to a value close to the original point. In practice, there was a slight drift to the right. At the time of writing, it is not known why this is the case.

calculates the number of bases to move. The drone is then commanded through the movements where the aforementioned custom methods ensure the drone doesn't move too far from its starting and target points. Finally, should the user decide to halt the game, the 'q' character is entered, triggering garbage collection and a successful exit. A visual representation of the program flow is depicted in Figure 2.

Regarding efficacy, though the program has shown considerable performance improvements compared to Project 2, the drone still has trouble with consistency. For example, the drone is able to perform perfectly when moving from one base to the next, regardless of its starting base. However, when called in succession, the drone performs unexpected movements such as unprompted descents or early landings. The program logic and flow has been reviewed multiple times to determine if this behavior is a result of a programming error. It is possible that there are unknown factors in the drone's operation that have not been taken into consideration such as communication error via the Bluetooth module; it is uncertain at this stage.



Game Setup
Get User Input
Move the Drone
Verify Base
Received Quit Signal
Exit
Initialize Logger
Load Color Model
Load Waypoints
Get input data
Perform Validity Check
Determine number of bases to traverse
Verify drone is on the input base
Determine movement direction
Takeoff; Hover above current base
Move to next base
Once detected, fly to waypoint
Perform Color Detection on Base Surface
Compare detected color to color mappings
Correct base?
Movement successful
Raise Exception

Figure 2: Flowchart for the Drone Baseball game

References

[1] *3.8: Color Classifier*, Robolink, Sep. 2023. [Online]. Available: https://learn.robolink.com/lesson/3-8-color-classifier-cde/.

[2] *codrone-edu 1.9*, Robolink, Aug. 2023. [Online]. Available: https://pypi.org/project/codrone-edu/.

Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""baseball_game:

A Series of functions to assist with the development of a drone baseball
emulation program.

@author: Terrance Williams
"""
# noinspection PyUnresolvedReferences
import time
import logging
import json
from pathlib import Path
from codrone_edu.drone import *
from tjdrone import TDrone


# %% CONSTANTS
PITCH_POWER, ROLL_POWER, THROTTLE_POWER = 20, 30, -25  # power and directions: forward, right, and down
MOVE_VELOCITY = 0.5  # (m/s) MAX: 2.0 m/s
MOVE_TIME = 0.1
SLEEP_TIME = 1.5
COLOR_DETECT_THRESH = 50
SWITCH_DIST_THRESH: float = 22.  # (cm); Relative height difference that denotes a change from or to a base.
MIN_RELATIVE_HEIGHT, MAX_RELATIVE_HEIGHT = 20, 35  # (cm)
HEIGHT_SWITCHES = 2  # Number of times relative height must switch (i.e. exceed difference threshold)
TOTAL_BASES: int = 4
BTMRANGE_SENSOR_UNIT = 'cm'
COLOR_DETECT_ATTEMPTS = 50

# Notes
C4 = Note.C4
E4 = Note.E4
G4 = Note.G4
C5 = Note.C5
REST = Note.Mute
NOTE_DURATION = 250  # (ms) 6/8 time at 120 BPM

# %% Mappings
HOME = 0
BASE_1 = 1
```

```python
BASE_2 = 2
BASE_3 = 3
# Base Color mapping [color: (base_num, color_rgb)]
base_color_mappings = {
    'green': (HOME, (0, 255, 0)),
    'red': (BASE_1, (255, 0, 0)),
    'yellow': (BASE_2, (255, 255, 0)),
    'blue': (BASE_3, (0, 0, 255))
}
base_number_mappings = {
    HOME: 'Home',
    BASE_1: 'First',
    BASE_2: 'Second',
    BASE_3: 'Third'
}
hit_mappings = {
    'miss': 0,
    'single': 1,
    'double': 2,
    'triple': 3,
    'home run': 4
}
base_waypoints = {}

# %% Logging
log_path = Path() / 'logs'
if not log_path.exists():
    log_path.mkdir()

log_index = len([x for x in log_path.iterdir()])
logfile = log_path / f'baseball_{log_index:02d}.log'
logging.basicConfig(filename=logfile, encoding='utf-8', level=
logging.DEBUG)


# %% Functions
def low_hover(
        drone: TDrone,
        min_height=MIN_RELATIVE_HEIGHT,
        max_height=MAX_RELATIVE_HEIGHT) -> float:
    """Hover the drone within some relative height range"""
    drone.relative_takeoff()
    drone.hover(SLEEP_TIME)
    drone.set_throttle(THROTTLE_POWER)

    curr_dist = 0
    while not min_height < curr_dist < max_height:
        drone.move(MOVE_TIME)
        curr_dist = drone.get_bottom_range(unit=BTMRANGE_SENSOR_UNIT)
        if curr_dist < min_height:
            drone.set_throttle(-THROTTLE_POWER)
        elif curr_dist > max_height:
            drone.set_throttle(THROTTLE_POWER)
        # time.sleep(0.01)
    else:
        drone.hover(SLEEP_TIME)
        return curr_dist


# noinspection PyPep8Naming
def calibrate_bases(drone: TDrone) -> dict:
    this_func = 'calibrate_bases'
    # noinspection PyUnusedLocal
    color_path = Path('../color_data')

    # Load color classifier
    if not color_path.is_dir():
        print("Could not find color_data directory. Using defaults")
        drone.load_classifier()
    else:
        drone.load_classifier(dataset=color_path)
    # Check if there are pre-configured waypoints
    way_path = Path('waypoints/saved_waypoints.json')
    if not way_path.parent.is_dir():
        way_path.parent.mkdir()
    if way_path.is_file():
        with open(way_path, 'r') as f:
            waypoints = json.load(f)
            print(waypoints)
    else:
        """Sets waypoints for each base if unable to load"""
        waypoints = {}

        # Set the waypoint to the next base beginning from HOME
        for i in range(TOTAL_BASES):
            _ = input('Press Enter to continue: ')
            base_to_calibrate = (i + 1) % TOTAL_BASES
            logging.info(f'<{this_func}> Calibrating Base {base_number_mappings[base_to_calibrate]} from Base {i}')
            print(f'<{this_func}> Place Drone on {base_number_mappings[i]} Base.')

            consec_detect = 0
            while consec_detect < COLOR_DETECT_THRESH:
                color = drone.predict_colors(drone.get_color_data())
                # print(color)
                frnt_clr, back_clr = color
                if frnt_clr == back_clr and frnt_clr in base_color_mappings:
                    current_base, LED_color = base_color_mappings[frnt_clr]
                    drone.set_drone_LED(*LED_color, 100)
                    if current_base == i:
                        consec_detect += 1
                        print(f'{COLOR_DETECT_THRESH - consec_detect}',
                              end=' ')
                    else:
                        print('\r', end="")
                        consec_detect = 0
                else:
                    print('\r', end="")
                    consec_detect = 0
            else:
                # Add waypoint to mapping dict
                print((
                    f"<this_func>: Pilot the drone to Base {base_to_calibrate}"
                    " and then input the requested key.")
                )
                drone.fire_start()
                drone.set_waypoint()
                drone.land_reset()
                waypoints[base_to_calibrate] = drone.waypoint_data[i]
                print(f'Waypoints:\n')
                for waypoint in waypoints.values():
                    print(waypoint)
                print('')
                # time.sleep(SLEEP_TIME)
        else:
            # Write waypoints to file
            with open(way_path, 'w') as f:
                json.dump(waypoints, f)
    return waypoints


```

```python
def await_input(drone: TDrone) -> None:
    quit_signal = 'q'
    done = False
    current_base = HOME

    while not done:
        input_val = input("Insert a Hit Value: ").lower()
        if input_val == quit_signal:
            print("[INFO] EXITING program.")
            done = True
        elif input_val in hit_mappings:
            num = hit_mappings[input_val]
            current_base = move_bases(current_base, num, drone)
        elif input_val in [str(i) for i in range(TOTAL_BASES + 1)]:
            num = int(input_val)
            current_base = move_bases(current_base, num, drone)
        else:
            print("Insert a hit value (miss, single, double, triple, or home run) or",
                  " the number of bases to run (0 to 4).\n", "Enter 'q' to quit.\n", sep='')


def move_bases(current_base: int, num_bases: int, drone: TDrone):
    """Performs a series of base movements"""
    this_func = 'move_bases'
    # Input checks
    try:
        if current_base < HOME or num_bases < 0:
            raise ValueError("Current Base and number of bases to run must be non-negative.")
        if current_base > BASE_3:
            logging.critical(f'Input base {current_base} is outside bounds.')
            raise ValueError("Current Base is limited from 0 to 3 (inclusive).")
    except TypeError:
        print("Current Base and number of bases to run must be integers.")
        raise
    # Ensure integer inputs (floors any non-int value)
    current_base, num_bases = int(current_base), int(num_bases)

    # Trivial condition
    if num_bases == 0:
        print(f"<{this_func}> Drone does not move.")
        logging.info(f"<{this_func}> Drone doesn't move.")
        return current_base

    # Check if provided number of bases to run result in a full run (back to HOME).
    if current_base + num_bases >= TOTAL_BASES:
        num_bases = TOTAL_BASES - current_base

    print(f'<{this_func}> (Current Base, Target Base): ({current_base}, '
          f'{(current_base + num_bases) % TOTAL_BASES})')
    logging.info(f'<{this_func}> (Current Base, Target Base): ({current_base}, '
                 f'{(current_base + num_bases) % TOTAL_BASES})')
    # Move to the bases
    for _ in range(num_bases, 0, -1):
        current_base = move(current_base, drone)
        time.sleep(SLEEP_TIME/6)
    else:
        return current_base


# noinspection PyPep8Naming
def move(current_base: int, drone: TDrone) -> int:
    this_func = 'move'
    """Moves from current base to next base"""

    # Ensure that the drone is on the proper current base.
    for _ in range(COLOR_DETECT_ATTEMPTS):
        test_color = drone.predict_colors(drone.get_color_data())
        if test_color[0] == test_color[1]:
            test_base, _ = base_color_mappings[test_color[0]]
            if test_base == current_base:
                break
    else:
        raise ValueError("Could not verify the drone's current base.")

    target_base = (current_base + 1) % TOTAL_BASES
    logging.info(f'<{this_func}> Moving from {current_base} to {target_base}')
    print(f'<{this_func}> Moving from {current_base} to {target_base}')

    curr_dist = low_hover(drone)
    logging.debug(f'<{this_func}> Initial Bottom Range Value {BTMRANGE_SENSOR_UNIT}: {curr_dist}')

    # Set new movement params
    """
    Movement logic. ASSUMES ONLY TRANSLATIONAL MOVEMENT (for now).
    HOME: move forward to Base 1
    Base 1: Move left to Base 2
    Base 2: Move backwards to Base 3
    Base 3: Move right to HOME
    """
    if current_base == HOME:
        drone.set_pitch(PITCH_POWER)
        # drone.set_roll(ROLL_POWER)
    elif current_base == BASE_1:
        # drone.set_pitch(PITCH_POWER)
        drone.set_roll(-ROLL_POWER)
    elif current_base == BASE_2:
        drone.set_pitch(-PITCH_POWER)
        # drone.set_roll(-ROLL_POWER)
    elif current_base == BASE_3:
        # drone.set_pitch(-PITCH_POWER)
        drone.set_roll(ROLL_POWER)
    else:
        # If somehow the current_base is invalid.
        raise ValueError(f"Invalid current base value: {current_base}")

    # Move until the drone reaches another base (two substantial
    # changes in relative height)
    dist_switch = 0
    while dist_switch < HEIGHT_SWITCHES:
        drone.move(MOVE_TIME)
        # time.sleep(MOVE_TIME)
        next_dist = drone.get_bottom_range(unit=BTMRANGE_SENSOR_UNIT)
        logging.debug(f'<{this_func}> Bottom-Range Reading: {next_dist}')
        # print(f'<{this_func}> Bottom-Range Reading: {next_dist}')
        if (abs(next_dist - curr_dist) >= SWITCH_DIST_THRESH
and
                (curr_dist > 0 and (0 < next_dist < 900))):
            dist_switch += 1
            logging.info(f'<{this_func}> Relative Height switch no. {dist_switch} from {curr_dist} to {next_dist}')
            print(f'<{this_func}> Relative Height switch no. {dist_switch} from {curr_dist} to {next_dist}')
            curr_dist = next_dist
    else:
        logging.info(f'<{this_func}> Distance-switching trips met.')
        print('Distance-switching trips met.')

    # Adjust Position; ensure landing
    drone.hover(MOVE_TIME)
    print("[INFO] Adjusting position...")
    logging.debug(f'{this_func}: Going to waypoint {base_waypoints[str(target_base)]}')
    drone.goto_waypoint(base_waypoints[str(target_base)], MOVE_VELOCITY)
    drone.hover(1)
    drone.land_reset()
    while drone.get_bottom_range(unit=BTMRANGE_SENSOR_UNIT) > 0:
        time.sleep(SLEEP_TIME/4)

    # Detect Base based on color
    for i in range(COLOR_DETECT_ATTEMPTS):
        colors_detected = drone.predict_colors(drone.get_color_data())
        if colors_detected[0] != colors_detected[1]:
            logging.debug(f"<{this_func}> Color-Detection {i}: Color values differ {colors_detected}.")
            continue
        if colors_detected[0] in base_color_mappings:
            current_base, LED_color = base_color_mappings[colors_detected[0]]
            logging.info(f'<{this_func}> Detected {colors_detected[0].upper()} associated with Base {current_base}')
            break
        time.sleep(SLEEP_TIME/10)
    else:
        # noinspection PyUnboundLocalVariable
        logging.critical(f"<{this_func}> Detected color {colors_detected} is not one of the colors assoc. with a base.")
        drone.set_drone_LED(0, 0, 0, 0)
        raise ValueError(f"<{this_func}> Detected color {colors_detected} is not one of the colors assoc. with a base.")

    # Change LED Color
    if current_base != target_base:
        drone.set_drone_LED(0, 0, 0, 0)
        logging.critical('Drone landed on incorrect base.')
        raise ValueError('[ERROR]: Drone landed on incorrect base.')
    else:
        drone.set_drone_LED(*LED_color, 100)
        logging.info(f"<{this_func}> Success.")
        print(f"<{this_func}> Success.")
        return current_base


def play_song(drone: TDrone):
    drone.drone_buzzer(C4, NOTE_DURATION)
    drone.drone_buzzer(E4, NOTE_DURATION)
    drone.drone_buzzer(G4, NOTE_DURATION)
    drone.drone_buzzer(C5, NOTE_DURATION // 2)
    drone.drone_buzzer(REST, NOTE_DURATION)
    drone.drone_buzzer(G4, NOTE_DURATION)
    drone.drone_buzzer(C5, 3 * NOTE_DURATION)


def play_ball(drone: TDrone):

    """
    Perform any required drone setup.
    Waits until drone is on the HOME plate, plays the start song, and then
    awaits user input
    """
    global base_waypoints

    # Print base-color associations
    message = ("Welcome to Drone Baseball!\nBefore we begin, let's calibrate"
               " the bases.\nHere are the current color associations:")
    print(message)
    logging.info("Current Base-Color Associations")
    for color in base_color_mappings:
        base_num = base_color_mappings[color][0]
```

```python
            assoc_base = base_number_mappings[base_num]
            print(f'{color.title()}:\t{assoc_base}')
            logging.info(f'{color.title()}:\t{assoc_base}')

    base_waypoints = calibrate_bases(drone)
    print(f'Check waypoints: {base_waypoints}')
    # Play Baseball Song
    play_song(drone)
    print("\nPlay Ball!\n")
    time.sleep(SLEEP_TIME)

    for key in base_waypoints:
        print(base_waypoints[key])
    await_input(drone)

    logging.info("User Exit.")


if __name__ == '__main__':
    # logging.getLogger().setLevel(logging.INFO)
    with TDrone() as t_drone:
        t_drone.set_drone_LED(255, 255, 255, 100)
        t_drone.reset_trim()
        play_ball(t_drone)

```