ME4232: Small Aircraft and Unmanned Aerial Vehicles

Group Report

The Control & Software Architecture of a Ball-catching Drone

Student Names:
Arijit Dasgupta
Jasper Tan
Arjun Agrawal

Student Number:
A0182766R
A0136232R
A0168924R

Student Emails:
arijit.dasgupta@u.nus.edu
jasper.tan@u.nus.edu
arjun.a@u.nus.edu

April 17, 2021

Contents

Acknowledgements

We would like to express our gratitude to Dr Sutthiphong Srigrarom (Spot) for teaching us the fundamentals of UAS and drone technology in this module, ME4232. We have thoroughly enjoyed learning from the lectures and have developed a great deal of appreciation for the world of unmanned aerial systems and drones. We also appreciate his effort to expose us to many aspects of unmanned aerial systems like drone racing, efforts by the industry and much more. His commitment to his students and our project's success is evident from the immense amount of time he has given to make this module a worthwhile experience.

We are lucky and privileged to be among the first few to take this module and we wish to express our extreme positive feedback as students taking this class. This is probably the most unique and best module we have taken in Mechanical Engineering and we hope that this module will continue to be taught. Finally, we would also like to thank the industry leaders who gave a lot of advice during the semester weeks during ME4232 lectures to inspire and improve our project implementation.

1 Introduction

1.1 Motivation & Objective

This paper details the conceptualisation, simulation and testing of a projectile catching drone with the help of several robotics, AI, Machine Vision and drone control concepts. The motivation of this project is to produce a versatile projectile detection and navigation stack to enable safer traversal of drones. In order to serve as a proof of concept, our conceptualisation makes use of a ping-pong ball projectile that the drone needs to catch. Catching a ball is visually engaging problem which can easily function as an indication of how successful the system is. The problem of catching the ball allows for the opportunity to explore strategies to identify and predict projectile trajectories.

The objective of this project is to conceptualise & build a control & architecture software that is integrated with MAVROS and PX4 to catch a ball projectile. We test our control & architecture software inside a Gazebo simulation environment as a form of proof-of-concept. To meet the standards of real-world and real-time systems, the architecture is designed such that there is no reliance on any external tools, like cameras fixed in the environment or motion capture tools. All tools used will be present on board the drone. This report makes the **assumption** that any drone using this software has a PX4 flight controller, a companion computer & a depth camera.

1.2 Background

Drone ball catching has generated many research works due to the exciting yet difficult task of catching quick moving objects out of the air autonomously [1–3]. The task of catching a ball requires us to push our creativity to create novel methods in colour detection and trajectory prediction. One excellent example of ball catching is the research done by the Flying Machine Arena research group in ETH Zurich [4]. In their research, they are able to perform ball catching tricks like juggling among multiple drone. To localise as well as identify the location of the ball, the Flying Machine Arena used a motion control system, using retro-reflective tape wrapped around a ball as a way to reflect infrared rays emitted from motion cameras. These infrared rays are able to localise the drones and the ball.

In this study, we are pushing the bounds of the research done by the Flying Machine Arena group. One clear disadvantage of using a motion control system is that localisation and ball detection must be done in the motion control lab, bounded by the motion capture cameras. Instead, we propose a different method - to use depth cameras to identify the location of the ball with respect to the drone, and to use odometry to localise the ball in relation to the room. This new method brings up a multitude of new problems. We need a robust method to identify the ball, and inform the drone's autopilot about the location of the ball. We also need a good trajectory prediction algorithm so that the drone can intercept the ball. In this report, we will explain how we attempted to solve each problem, as well as how far we were able to push our algorithms to catch the ball in different scenarios.

2 Proposed Control & Software Architecture

2.1 Software Development Framework

Robot Operating System (ROS) was chosen as our primary software framework, connecting all three main functions of our ball catching drone together: object detection, ball trajectory prediction and drone path planning. We chose ROS as it includes hardware abstraction, low-level device control and implementation of commonly used functionality. These traits allow us to repurpose open-source code for our own use as well as allow different sensors and actuators to communicate with one another. Lastly, ROS follows a modular philosophy of "complexity via composition", allowing each team member to work on different parts of the project and merge each separate part with little difficulty. This section will quickly introduce ROS and how we intend to utilise ROS on our ball catching drone.

Nodes: ROS nodes are a running instance of a ROS program. Nodes can subscribe to data, manipulate them, and finally publish them for other ROS nodes. In our project, we will use 4 nodes, each listed in the table at the end of the current section.

Topics: Nodes publish and subscribe on ROS topics. Topics connect nodes together and allow data to pass from one node to another in the form of messages.

Messages: ROS messages are data structures that hold data so that different nodes are able to communicate.

Functional Description of nodes in the Projectile Catching Drone	
Node Name	**Function**
MAVROS	The MAVROS navigation stack accepts Pose messages from our path planning node and ensures that the drone is able to follow the trajectory.
Point cloud Processing and Object Detection	Colour and shape identification is used to detect objects the drone needs to engage with. From the depth image data, the coordinates of the projectile along with the distance to the projectile is acquired in the drone frame.
Projectile Prediction	Once the drone has localised the projectile in its frame, a path prediction algorithm is used to track and predict the path of the projectile will take.
UAV Path Planning	Once the location of the projectile is acquired and predictions of its future states have been made, the path planning algorithm makes a global plan to catch the projectile

Table 2.1: Nodes in localisation pipeline

2.2 Object Detection

Object detection was fundamental to our ball catching task as we had to segment our image to identify an accurate position of the ball relative to the drone. By identifying the centroid of the ball's location and knowing a few other physical details about the ball and environment, we will be able to predict the trajectory of the ball. Using a depth camera, an RGBXYZ point cloud can be constructed around the drone, with cartesian coordinate (x, y, z) information and RGB colour model values encoded in each point of the point cloud. In this study, an Intel RealSense Depth Camera D435 was used as our depth camera. Using a ROS wrapper, we can set up a ROS node to start up and publish images from the colour and depth camera. By considering the distance between the depth and colour cameras, the ROS node is also configured to transform the coordinate frames between the two cameras so that their images align.

Using Point Cloud Library (PCL) [5], an open-source library for point cloud processing, we can easily convert colour and depth images to RGBXYZ point cloud data. The data is structured in the form of an unstructured array and is published from the Realsense camera at an average of 30Hz. As the flight time of the ball is expected to be very short ($< 1s$), we decided to write our object detection code from scratch in C++ language, using minimal libraries to reduce latency and maximise frames for ball trajectory prediction.

We begin our object detection algorithm by defining colour parameters – lower and upper red, blue and green colour values based on the RGB colour model. Using these colour parameters, we perform colour thresholding by searching through our entire array for points which fit between our colour parameters. These points are then isolated and based on their cartesian coordinates, points greater than one standard deviations away from the mean can be removed, as these points are likely background noise. Lastly, we will find the mean X and Y coordinate values from the remaining points, and the corresponding Z (depth) value. These coordinates are approximately where the centroid of the ball is located.

Finally, the centroid location is placed in a pose ROS message and published to the ball trajectory node.

One key limitation to our method of object detection is that the ball has to have a significantly different colour as the background. If the ball is orange (like a ping pong ball) and there are orange-coloured objects in the surrounding areas, then using standard deviations to remove stray orange points will not accurately produce the centroid's location. Instead, more advanced machine vision techniques like edge detection, where the circularity of the ball is also identified, or even realtime neural network algorithms (YOLO etc.).

Algorithm 1 Ball detection pseudocode

Initialise Colour range in RGB format
Subscribe and obtain the point cloud data in PointCloudXYZRGB format
for Each point in the PointCloudXYZRGB array **do**
    if RGB values fall in the colour range **then**
        Keep value
    else
        Remove value
    end if
end for
for Each point left in the PointCloudXYZRGB array **do**
    Calculate mean and standard deviation of points using X and Y Cartesian values. Remove points past threshold of 1 standard deviation.
    Get maximum and minimum X and Y Cartesian values left over to calculate Z Cartesian values.
end for
Publish mean X and Y values and the calculated Z value as the ball's centroid location relative to the drone.

2.3 Trajectory Prediction

This is the most challenging step of this framework. For the drone to plan its path to catch the ball, it can simply use this location of the ball and use that as the goal at every time interval the realsense camera detects the ball. This method, which is a sort of a "cat chasing mouse" path planning, will be discussed later as Method 1 in Section 2.4. Such a method requires no forward planning of where the ball is going. However, what if our architecture can predict where the ball is going? By subscribing to the sequences of real-time $X_{ball}, Y_{ball}, Z_{ball}$ information coming from the algorithm described in Section 2.2 (ball detection ROS node), we can also predict the future path of the ball in real-time. This provides for a new variety of path planning methods for the drone to catch the ball. We have written another ROS node (ball trajectory node) that uses a numerical method on basic kinematic and dynamic equations to predict and publish the predicted trajectory of the ball. Note that all steps of the algorithm occur sequentially after each callback to the subscription to the $X_{ball}, Y_{ball}, Z_{ball}$ data from the camera.

Step 1: Find the velocity

The velocity $(U_{ball}, V_{ball}, W_{ball})$ can be easily measured by observing how $X_{ball}, Y_{ball}, Z_{ball}$ changes with time, followed by a least squares linear regression (in each X, Y, Z direction) to take the gradient as velocity. Equation 2.1 shows the calculation of the speed in the X direction to get U_{ball}. This step makes the big assumption that between these successive measurements of $X_{ball}, Y_{ball}, Z_{ball}$, the velocity of the ball is unchanged. It is possible to make this assumption if the time interval of regression is small and hence a negligible true theoretical acceleration. As our system is real-time, the prediction algorithm uses a queue (linked-list) data structure to implement the regression. When the first two $X_{ball}, Y_{ball}, Z_{ball}$ points come in, they are enqueued into the queue and the algorithm immediately does the regression and determines the velocity. As more points come in, they are enqueued into the queue for regression and the velocity is constantly being determined. Having more points reduces the random error of the $X_{ball}, Y_{ball}, Z_{ball}$ measurements and hence a more accurate $U_{ball}, V_{ball}, W_{ball}$ determination. However, for our earlier made assumption to hold, we cannot have too many points as the time interval will become too big, hence there must be a balance of the maximum number of points in the queue. This maximum number of points then becomes a parameter to tune. When the queue is filled, and the next $X_{ball}, Y_{ball}, Z_{ball}$ point comes in, the earliest included point (front of the queue) will be removed and the new point will be enqueued at the back.

$$U_{ball} = \frac{n\sum tX_{ball} - \sum x \sum t}{n\sum t^2 - (\sum t)^2} \tag{2.1}$$

Another factor of consideration is the frame in which the velocity is measured. The predicted path of the ball should be determined in the world frame as the frame is not changing and it is easier to work with static frames. As mentioned before, the $X_{ball}, Y_{ball}, Z_{ball}$ coordinates are with respect to the drone frame. As the drone may be moving and accelerating during the ball detection with camera, we add the location of the drone (in world frame using the in-buit IMU of the flight controller, Section 2.5) to the location of the ball (in drone frame) in each time step to get the location of the ball in the world frame. Hence we replace $X_{ball}, Y_{ball}, Z_{ball}$ with $X_{ball} + X_{drone}, Y_{ball} + Y_{drone}, Z_{ball} + Z_{drone}$ before inserting into the queue to get the velocities in world frame.

Step 2: Determine the acceleration

Figure 2.1 illustrates the world coordinate frame of the ball. For reference to the real world, the positive y axis is opposite to the direction of gravity on earth. To get the acceleration of the ball, we must resolve its dynamics. The two forces acting on the ball are its weight and drag (from air resistance). We assume that the direction of drag is opposite to the ball's velocity. The mass of the ball (for ping pong ball) is 2.7 grams and hence has

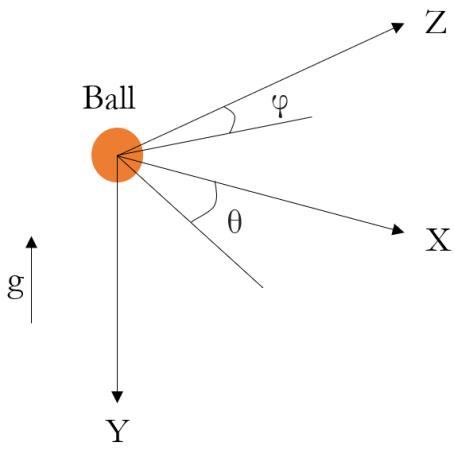

FIGURE 2.1: World Coordinate System of the ball

a weight of 0.026487N. To determine the drag of the ball we must first determine the drag coefficient (C_D) of a sphere, which is a function of the Reynolds number of the ball. Assuming that the ball moves at an average of $1m/s$, with a diameter (ping pong ball) of $0.04m$ and the air kinematic viscosity of $1.48 \times 10^{-5} m^2/s$, the Reynolds number is equal to $\frac{VD}{\nu} \cong 2700$. Based on the equation relating the Reynolds number (Re) of a sphere to C_D in Equation 2.2 by [6], we arrive at a C_D of $\cong 0.420$. We also need the magnitude of Velocity, $|V|$ to calculate drag and it can be determined via Equation 2.3.

$$C_D = \frac{24}{Re} + \frac{2.6(\frac{Re}{5.0})}{1+(\frac{Re}{5.0})^{1.52}} + \frac{0.411(\frac{Re}{2.63\times10^5})^{-7.94}}{1+(\frac{Re}{2.63\times10^5})^{-8.00}} + \frac{0.25(\frac{Re}{10^6})}{1+(\frac{Re}{10^6})} \tag{2.2}$$

$$|V| = \sqrt{U_{ball}^2 + V_{ball}^2 + W_{ball}^2} \tag{2.3}$$

Now we use Equation 2.4 to determine the value of drag, D_r, of the ball. ρ refers to the air density ($1.225kg/m^3$) and A is the projected surface area in contact with air(πr^2).

$$D_r = \frac{1}{2}\rho C_D |V|^2 A \tag{2.4}$$

Now we can resolve the free body diagram of the ball and calculate the accelerations in 3 directions (based on the world frame in Figure 2.1) as shown in Equation 2.5 where m and g are the mass and gravitational acceleration respectively. Note that ϕ & θ can be determined as $\tan^{-1}\frac{U_{ball}}{W_{ball}}$ & $\tan^{-1}\frac{V_{ball}}{U_{ball}}$ respectively.

$$a_x = -\frac{D_r}{m}\sin\phi\cos\theta \tag{2.5a}$$

$$a_y = g - \frac{D_r}{m}\sin\phi\sin\theta \tag{2.5b}$$

$$a_z = -\frac{D_r}{m}\cos\phi\cos\theta \tag{2.5c}$$

Step 3: Update of position & velocity

This final step involves updating the position and velocity of the ball for the next time step (in the future) using a pre-defined time step size parameter t_{step}. This update is done using the fundamental kinematic equations

for position and velocity as shown in Equation 2.6 for the X axis. It is important that the Position update is done before the velocity update as we should use the current measured velocity to predict the position of the next time step.

$$X_{ball} = X_{ball} + U_{ball}t_{step} + \frac{1}{2}a_x t_{step}^2 \tag{2.6a}$$

$$U_{ball} = U_{ball} + a_x t_{step} \tag{2.6b}$$

Completing the Algorithm

After obtaining the updated $X_{ball}, Y_{ball}, Z_{ball}$, also update the time at which the ball is expected to be in this position by incrementing the time by t_{step}. This position and time information is then published to a ROS topic called `pred_path` which the drone can use for its path planning. A pseudo-code of the algorithm is summarised in Algorithm 1.

Algorithm 2 Ball Trajectory Prediction Pseudo-Code

while Camera Detects the ball (in a specified frequency) **do**
  Initialise `path_list` to store position and time of predicted path
  Initialise the time step, `time_step` to specified value and the current time, `time`
  Subscribe and obtain $X_{ball}, Y_{ball}, Z_{ball}$ from Ball Detection node in drone frame
  Apply Step 1 (Equation 2.1)to determine $U_{ball}, V_{ball}, W_{ball}$ in world frame
  while Path is being determined **do**
    Apply Equations 2.2 to 2.5 in Step 2 to determine a_x, a_y, a_z
    Apply Equation 2.6 in Step 3 to update $X_{ball}, Y_{ball}, Z_{ball}$ and $U_{ball}, V_{ball}, W_{ball}$ in the order respectively
    `time` $\leftarrow$ `time` + `time_step`
    Append $X_{ball}, Y_{ball}, Z_{ball}$ and `time` to `path_list`
    Depending on the drone path planning algorithm, break out of path loop if necessary
  end while
  Publish `path_list` to a ROS topic for the drone to use in path planning
end while

2.4 Drone Path Planning

This section will discuss three major methods in drone path planning for catching the ball. for each method, we provide the benefits and downsides of implementing them from a theoretical standpoint.

2.4.1 Method 1 - Cat & Mouse

This first method that we call "Cat & Mouse" is the most straightforward implementation of path planning. This does not require the predicted path information from Section 2.3. Simply put, in every time interval that the camera detects the ball location, the path planning node in ROS tells the drone to go that exact location. Figure 2.2 illustrates the setpoint command that the path planning node gives to a stationary ball, which is straight into a ball. We define this specific scenario as scenario A.

In the case of a moving ball, the drone will still give commands to the location of the ball so long as it is in the view of the depth camera. Figure 2.9 shows how the setpoint command for the drone is different at different time intervals while a ball is moving away. Specific to there being no drone yaw rotation we label this as Scenario B.

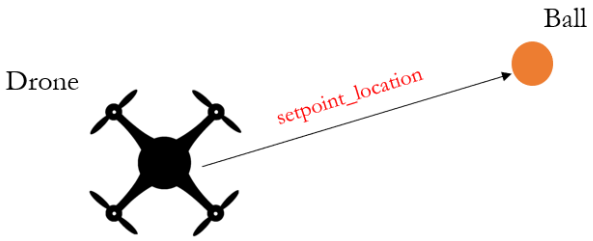

FIGURE 2.2: Scenario A: Drone path to stationary ball via the Cat & Mouse method

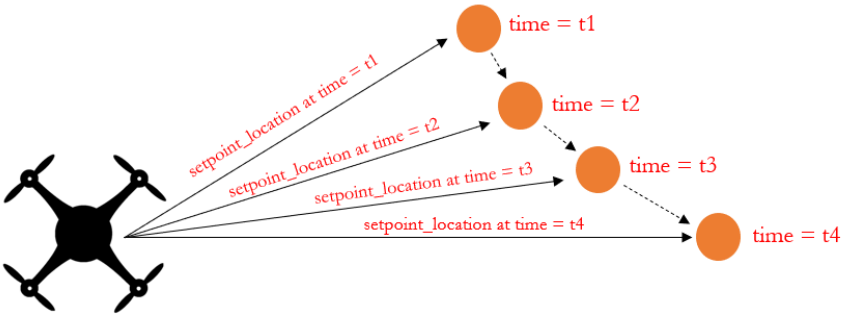

FIGURE 2.3: Scenario B: Drone path to moving ball via the Cat & Mouse method at different time instances

It is possible however for the ball to leave the sight of the depth camera, hence the drone is programmed to rotate if the detected ball appears to be too close to the edge of the camera view. This is done via another ROS node that calculates if the ball is in danger of escaping the 2D camera field of view, in which case it will command the drone to yaw clockwise or counter-clockwise. The drone could also roll and pitch to keep the ball in view but this could have an adverse effect if the drone is made to pitch or roll at a large bank angle, potentially leading to stall and fall. Hence we did not implement this. This can also be implemented in this Cat & Mouse method as illustrated in Figure 2.4. We call this Scenario C.

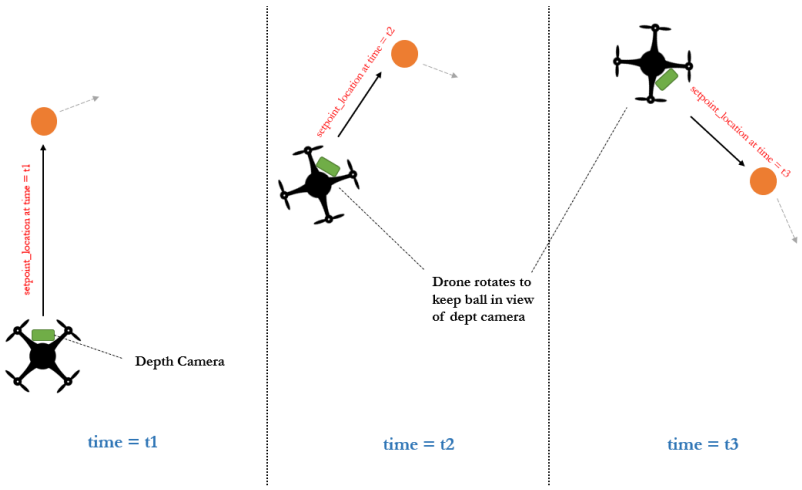

FIGURE 2.4: Scenario C: Drone path to moving ball via the Cat & Mouse method with yaw rotation at different time instances

The main and obvious benefit of this method is that it is very easy to implement and computationally inexpensive. Depending on the speed of the depth camera, the ROS nodes could be receiving the location of the ball multiple times per second. The algorithm for ball trajectory prediction as described in Section 2.3 can be computationally demanding and potentially show down the rate of information transfer to the path planning node. Not to mention, the path planning algorithm itself is literally just setting the waypoint to the ball coordinates (transposed to world frame), which is very fast. This reduces the response time for the drone to react to changes in the ball location.

However, the con of this algorithm is that the drone naturally takes an inefficient path. In the case of a real cat chasing a mouse, the mouse could move in unpredictable ways, so the cat naturally chases directly at the mouse to keep up. However, the trajectory for the ball is predictable with dynamics and kinematics. The drone would be more efficient in taking a shorter path to a location where the ball would end up instead of following wherever the ball goes. Figure 2.5 illustrates how a drone is able to capture the ball with a much shorter path if it knows where the ball will end up as opposed to the inefficient path using the Cat & Mouse method that requires the drone to be faster to catch the ball at the same point. This may mean that the drone may not be fast enough to catch the ball in certain scenarios. This could in fact lead to the drone crashing onto the ground (as the ball will eventually drop to the ground) if no failsafe is there in place.

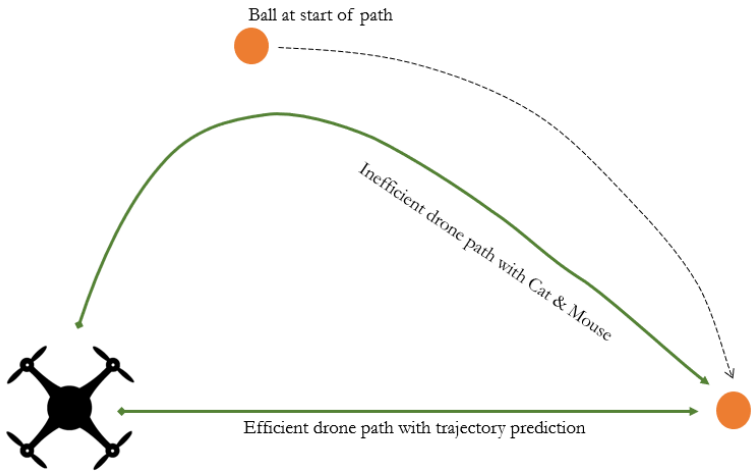

FIGURE 2.5: Comparison between a path with and without trajectory prediction

2.4.2 Method 2 - Prediction with Shortest Path

For methods 2 & 3, the drone yaws to keep the ball in the view of the depth camera at all times. We assume that the path planner node has knowledge on the maximum distance that a drone can travel based on given time intervals. The second method, which we call "Prediction with Shortest Path" first uses trajectory prediction (as described in Section 2.3) to determine the ball's path in the future. Note that the trajectory prediction is being continuously run to error correct in-flight as the predictions have errors in real-world conditions. This method then identifies which one of these points along the point the drone can reach before the ball reaches there. These points form regions as depicted by the green regions in Figure 2.6. This means that the time taken for the drone (based on its knowledge of how fast it can move) to move to a point must be less than the time taken for the ball to reach that point. Recall that the estimated time of when the ball will reach any point on the predicted path is also published by the ball trajectory node.

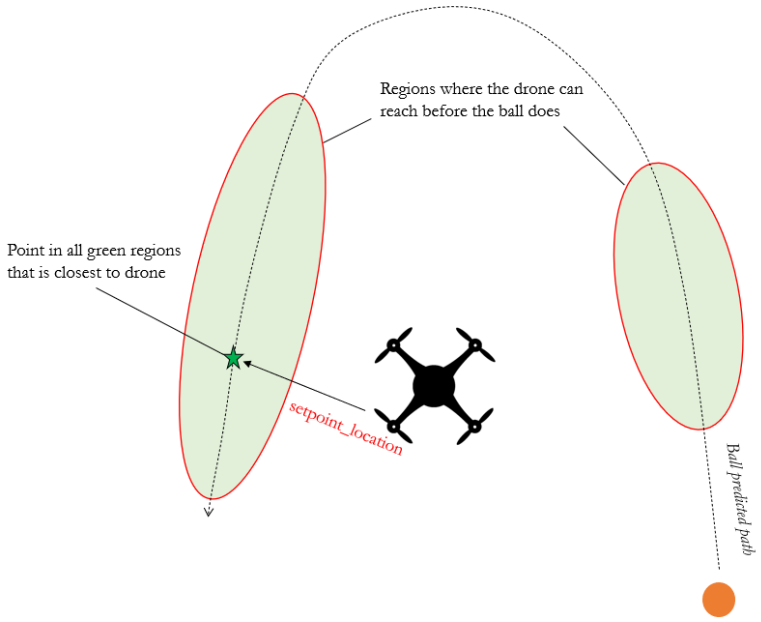

FIGURE 2.6: Scenario D: Drone path to a moving ball via the Prediction with Shortest Path method

After scanning through all points in the green region, the path planner node then selects the point nearest to the drone and then sets the waypoint to go to that path. We call this Scenario D. The clear advantage of this method is that it is efficient as it can predict the path of the ball. Second, it takes the shortest past route, which reduces flight time (in terms of movement) and thus makes the flight fast and efficient. Nonetheless, this method still has a serious downside, which is that the drone may end up moving to a position that is far down the predicted trajectory of the ball. Any error of the predicted path of the ball close to the ball will only get bigger and amplified with time, as illustrated in Figure 2.7. Such a scenario is even more likely when the ball is being thrown near to the drone. Certainly the fact that trajectory prediction is continuously done in-flight can slightly alleviate this problem, but even that comes with another issue. As the shortest path is continuously being calculated in flight, it is possible that the closest few points to the drone are in completely different parts of the predicted trajectory. This may possibly lead to the path planner node choosing completely different setpoints during flight, which may end up confusing the drone and never catching the ball.

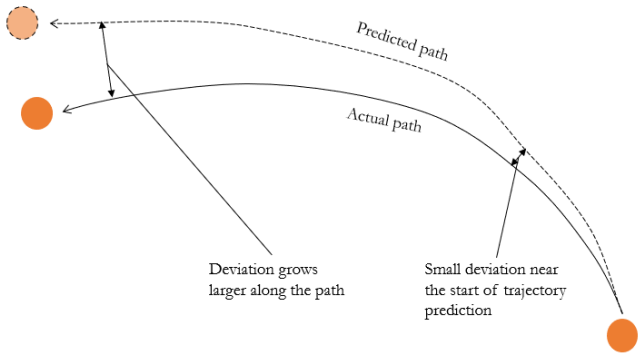

FIGURE 2.7: Blowup of small deviation at the start of trajectory prediction along the path

2.4.3 Method 3 - Prediction with Fastest Path

In light of the issues faced by the previous method, we have modified it to another method we call "Prediction with Fastest Path". In stead of choosing the closest point, the path planner node chooses the point that is the earliest along the path of the ball. This is illustrated in Figure 2.8, where the point in the green region that is earliest in the ball path becomes the setpoint for the drone. We label this Scenario E. This method has some additional benefits. Considering that it approaches the earliest point of the ball path, it is the fastest and hence most efficient method to catch the ball. Note that the fastest path may not be the shortest path as in the latter case, the drone may have to hover a while before the ball reaches it. Moreover, as it attempts to catch the ball in the earliest part of the predicted path, the errors and path deviations are less pronounced. Not to mention that as the drone approaches the ball, the continuous in-flight path planning node will only reinforce a setpoint early in the ball path as the drone will be closer to the early region of the ball's predicted path.

Like the two earlier methods, method does indeed come with setbacks. This method poses a higher risk to missing the ball as it relies on very accurate trajectory prediction and an accurate understanding of its own speed. This is very difficult to achieve, especially in a real world scenario. The ball detection and ball trajectory prediction stages of the control & software architecture must be extremely robust in all scenarios. This method is a somewhat ideal method that is difficult to achieve in a non-ideal world. Nonetheless, it is worth exploring this option for this project.

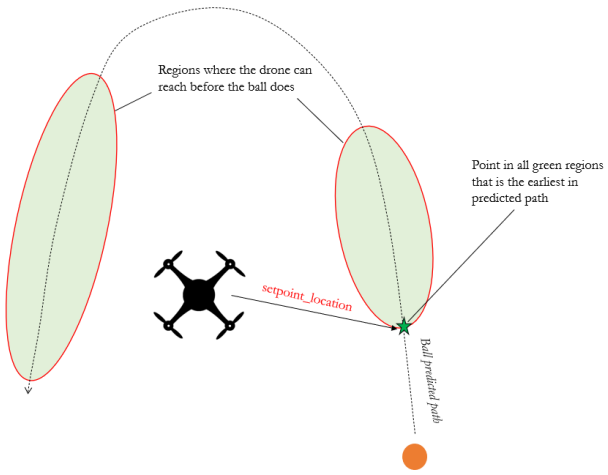

FIGURE 2.8: Scenario E: Drone path to a moving ball via the Prediction with Fastest Path method

2.5 Flight Control System

MAVROS is a ROS package which enables interaction between ROS and autopilot software using the MAVlink protocol. MAVLINK is an efficient protocol which consists of 17 byte messages including the message ID, target ID and data. MAVROS allows the companion computer, in particular the path planning node, to create position setpoints for the drone to follow.

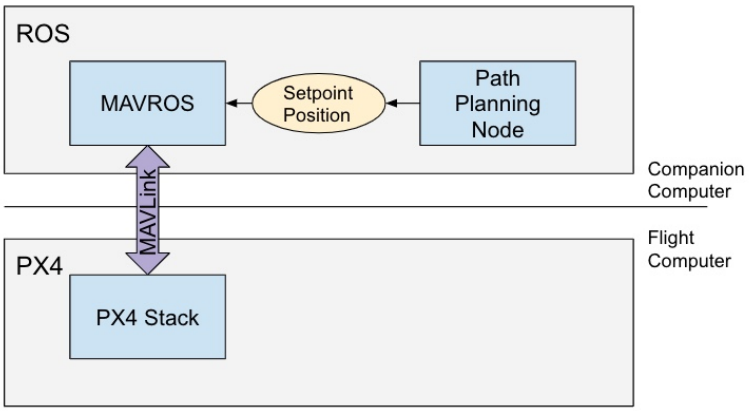

FIGURE 2.9: Comunication link between companion and flight computer

In this project, the companion computer is the Jetson Nano while the Flight Controller is the PX4. The PX4 uses sensors to determine vehicle state needed for stabilization and to enable autonomous control. The system minimally requires a gyroscope, accelerometer, magnetometer and barometer. This group of minimal sensors is incorporated in PX4 flight controller. Additional sensors can be configured to provide data to the PX4. A GPS is often a must to enable global frame referencing. In our implementation, we only use the sensor suite found onboard the PX4. Since our world frame is the original position of the drone, the drone and ball position can easily be tracked with the help of basic odometry. Figure 2.10 shows the system's TF tree and the nodes which create the world-drone transform and the drone-ball transform.

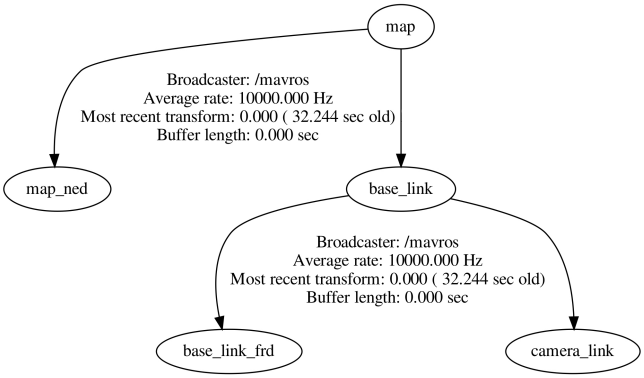

FIGURE 2.10: System TF tree.

The PX4 uses a cascading control architecture with a mix of P and PID controllers to ensure that the drone meets its setpoint values. The estimates of the values in the controllers come from an EKF which merges available sensor data for more accurate pose information. GNSS data and Range finder data be merged into the EKF to improve estimates. This is not done so in our system as we have a singular goal of catching the projectile with the help of the depth camera. If needed the data RGBD data from the camera can be used to carry out visual odometry hence improving both the control of the PID and the localisation system. Once the

entire setup (From ball detection to flight software integration) was built & integrated, Figure 2.11 shows the nodes and active topics present in the architecture. Note that drone_test_1 is the path planning node and this RQT graph is for scenario C, hence the trajectory prediction node is not being run.

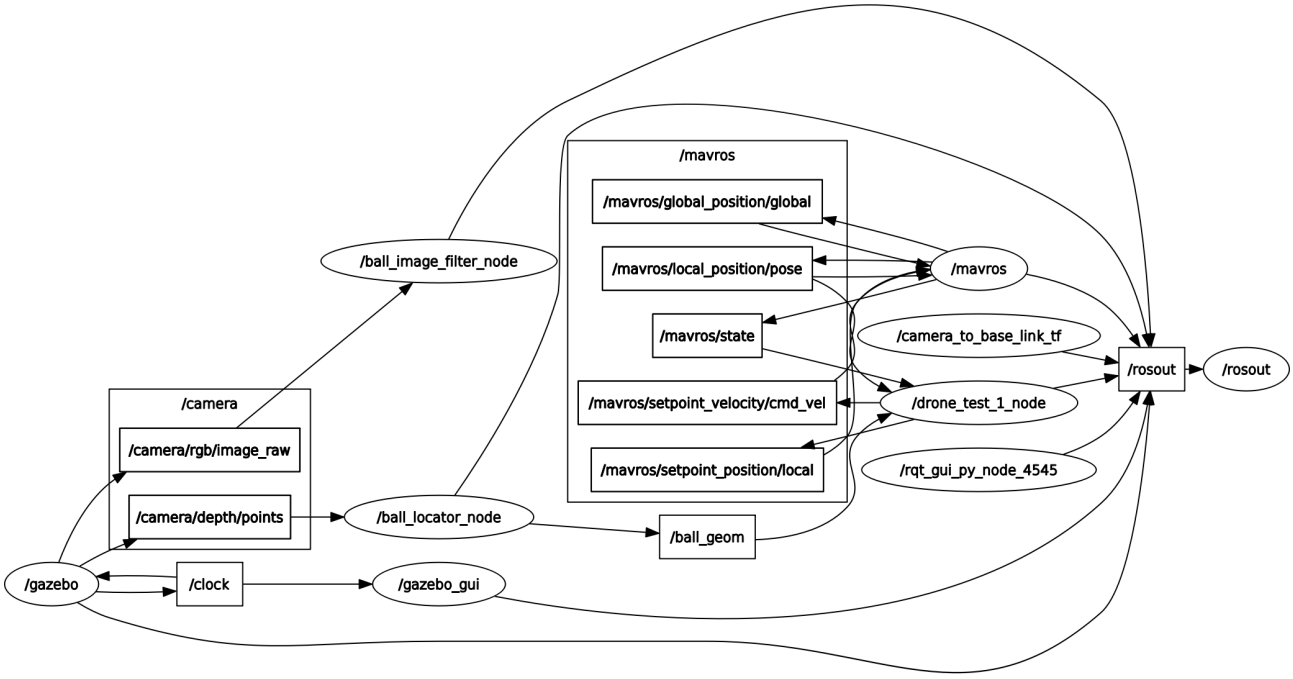

FIGURE 2.11: The RQT Graph showing all nodes & active topics in the control & architecture.

3 Methodology

3.1 Simulation vs Real-World

In this project, we originally aimed to create a physical demo of our control and software architecture using a physical drone with a PX4 Pixhawk flight controller borrowed from Temasek Laboratories, NUS. We also acquired a Jetson Nano for an onboard computer and a RealSense T435 depth camera. After testing out the camera and flight controller with MAVROS inside the Jetson Nano for a few weeks, we made the decision to switch to a purely simulation-only demo. This decision stemmed from three reasons.

The first reason was that we realised it was incredibly hard to come up with an accurate enough colour threshold for an orange ping pong ball. Due to the changes in lighting conditions in the real world, the ball would most often be detected, but often swarm out into a random bunch of pixels in a slightly different lighting condition. This made it incredibly difficult to reliably detect the ball and determine the centroid with real depth images from the RealSense depth camera, as shown in Figure 3.1. Naturally, a good solution is to use advanced machine vision techniques like edge detection, where the circularity of the ball is also identified, or even real-time neural network object detection algorithms (YOLO, ResNET etc.). Due to time constraints, we decided not to implement any advanced machine vision approaches.

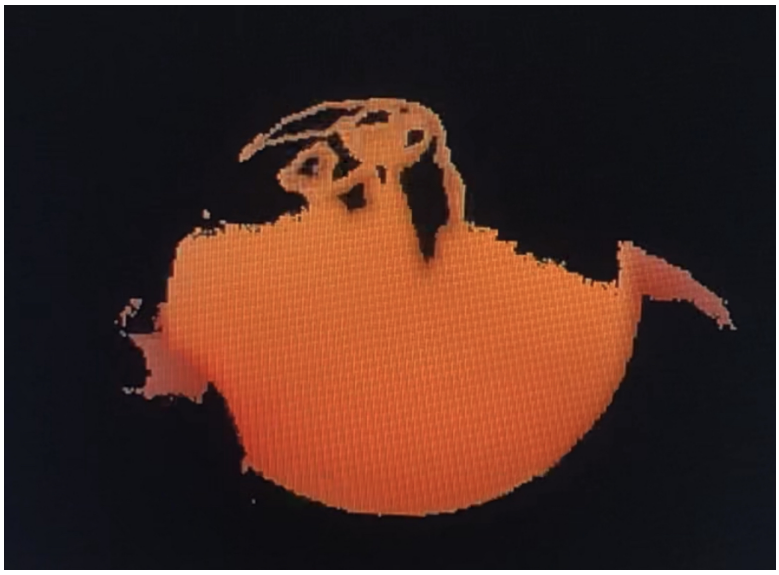

FIGURE 3.1: Example of a bad ball detection with a **real** depth camera

The second reason is that we realise that the ball moves too fast in the view of the camera. Often the ball would only stay in the view of the depth camera for approximately 0.2s (even after slowing the way we throw the ping pong ball). With a frame rate of 25Hz, that would mean that the ball only appears for 5 frames in view, which is way too little to conduct an accurate and meaningful trajectory prediction. Such non-ideal conditions pose extra challenges to the architecture we designed.

The third reason is the time constraints and impracticality of conducting a physical demo. This project was at most 2 months long, which is a short period to figure out everything about the hardware and software of the drone and making them with with the two aforementioned non-ideal conditions. Not to mention that physically testing the drone can only be done in the Vicon room of Temasek Laboratories, NUS which is not always freely available. Additionally, if the flight of the drone ever goes wrong and it mistakenly collides with the ground or walls, there will be a lot of damage that our team would have to account for.

The aim of this project is to show a proof-of-concept of our control & software architecture instead of tackling the aforementioned challenges head-on. Such a proof-of-concept can just as easily be shown in simulation. Hence, we decided on using a simulation for this project.

3.2 Simulation Environment

Gazebo was used to simulate the drone and the ball catching ability. Gazebo is an open-source program that offers the ability to simulate robots in any environment. Gazebo provides a physics engine that is highly programmable with a wide variety of open source models available for use.

Gazebo World Generation

To handle the different simulation requirements, we edited the settings on Gazebo's physics engine. Firstly, we had to decrease the physics update rate to 0.1x of real time speed. This is due to the high computation power required to simulate the drone, the depth camera and the ball physics. We realised that the computational power required was negatively impacting our simulations when the refresh rate of the depth camera was registering 3-4 frames per second. By slowing down the physics engine by 10 times, we increased the update rate to roughly 30 frames per second. This was crucial to the success of the simulations as prior to the slow down, the drone would frequently miss the ball if the ball was moving too fast for the camera to capture. Figure 3.2 shows an example of the gazebo environment with three random objects placed in front a drone.

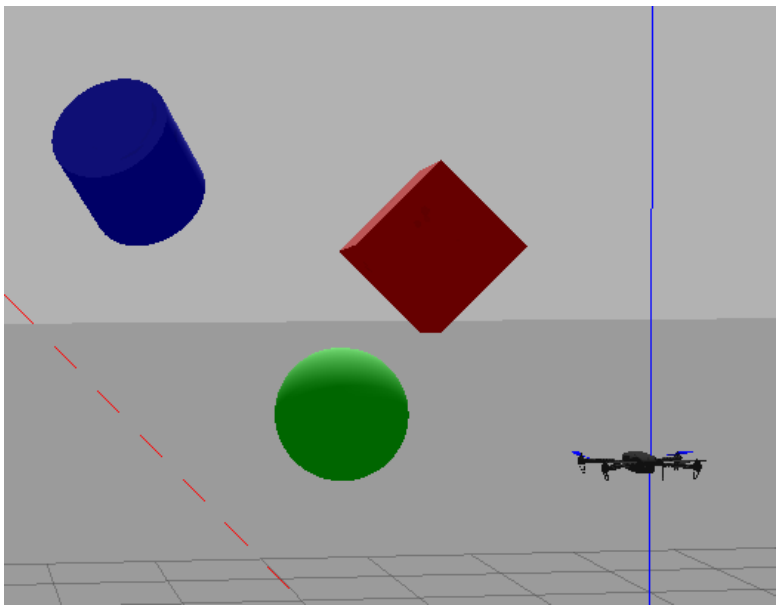

FIGURE 3.2: Gazebo Environment

Ball Simulation

The ping pong ball is simulated in Gazebo using a round ball model as seen in Figure 3.3. Editing the Simulated Description Format (SDF) file, we can apply an orange colour (255, 165, 0) in RGB format. Furthermore, we can set the mass of the ping pong as 2.7 grams, equivalent to the standard set by the International Table Tennis Federation. We chose an orange ping pong ball instead of using a ball with one of the three primary colours of light (red, blue, green) to make our task a little more challenging.

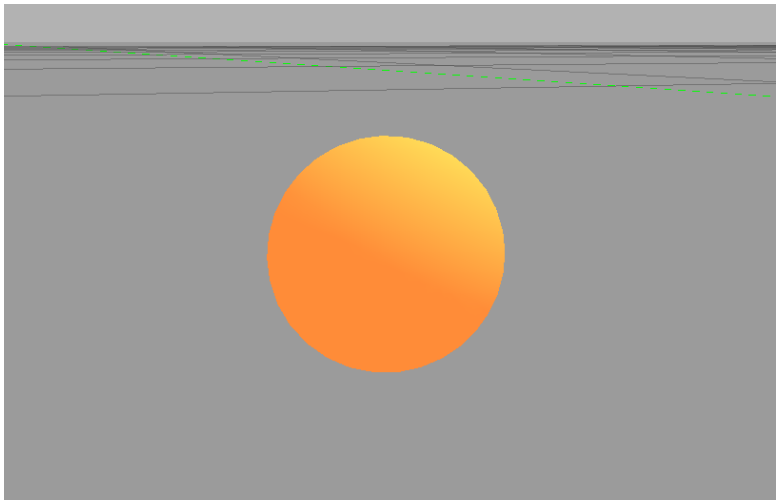

FIGURE 3.3: Ping Pong Ball in Gazebo

Depth Camera Simulation

In our implementation, hardware testing was carried out on the Realsense D435i and the Jetson Nano. The depth camera is able to provide a point cloud of objects surrounding the camera's field of view. The point cloud in Figure 3.4 was formed by aligning the depth image with the 2D RGB image. This alignment provides both colour and depth information that is needed for the ball detection.

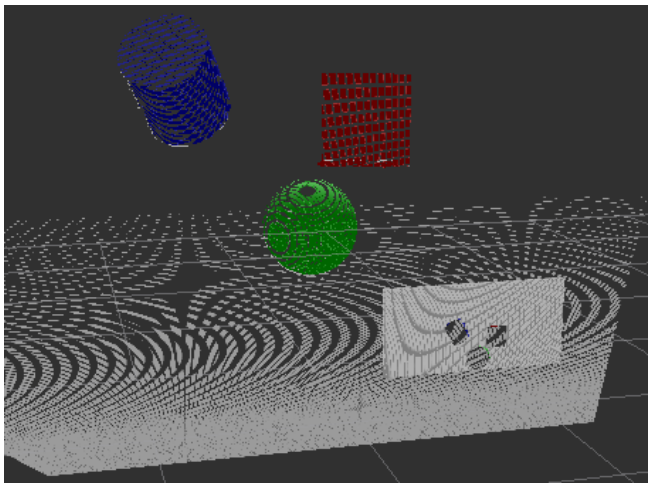

FIGURE 3.4: Point Cloud of the Gazebo environment in Figure 3.2

One of our concerns with the implementation was the frequency at which the data from the camera could be processed and published into the ROS framework. After optimisation through voxel filtering, rectification and image size reduction, the publication rate was a mere 5Hz. This low frequency was concerning because the

Jetson Nano was only able to capture a maximum of 3 frames with the projectile's trajectory. Hence making it improbable that the Jetson would be able to track and predict the ball's trajectory and command the movement of the drone. Once the project focus was switched to a simulated contextualisation, simulated realsense D435i from intel was tested.

Drone Simulation

For our simulations, we use an iris drone, a built-in model in Gazebo that has the RealSense depth camera attached to it. This drone is shown in the Gazebo environment in Figure 3.5. To control the drone, MAVROS topics are used. Simulation nodes are written to run the drone control in the scenarios simulations. These nodes take in data right from the path planning node. To control the drone, the `mavros/set_mode` topic is accessed in a to call a service to change the mode of the drone to "OFFBOARD", which means that it can now be controlled by a companion computer (which is our linux system running the simulation in this case). After wards, the `mavros/arm` topic is accessed to call a service to arm the drone. Once the drone is armed, it can now takeoff. The drone is then sent a setpoint location to fly and hover at an elevation of $2m$ above the ground. Now the drone is prepped and ready to commence the simulation of our scenarios. Note that all of the simulations are done in an Ubuntu 18 operating system and Figure 3.6 shows an illustration of the dashboard used for our control & software architecture implementation.

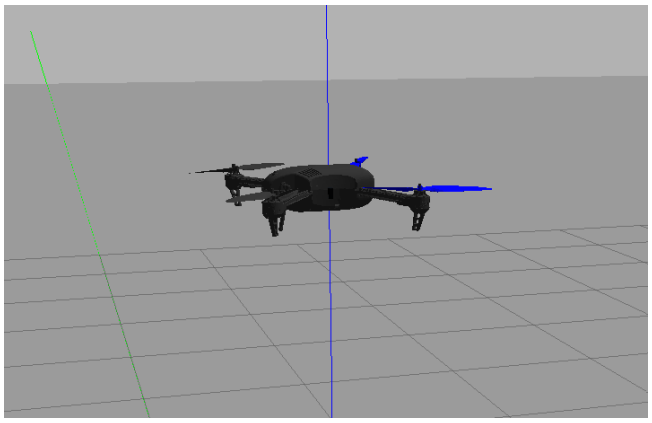

FIGURE 3.5: Iris drone in Gazebo

For this project, the control of the drone is done by simply publishing all the setpoint locations from the path planning node to the `mavros/setpoint_position/local`. Our simulation ROS nodes can publish the setpoint locations as fast as the Linux system can handle the computations of all the previous steps (ball detection, ball trajectory prediction). Note that to identify the location of the drone in the world frame, the trajectory prediction and path planning nodes can subscribe to the `mavros/local_position/pose`. With this, the drone can be simulated in Gazebo.

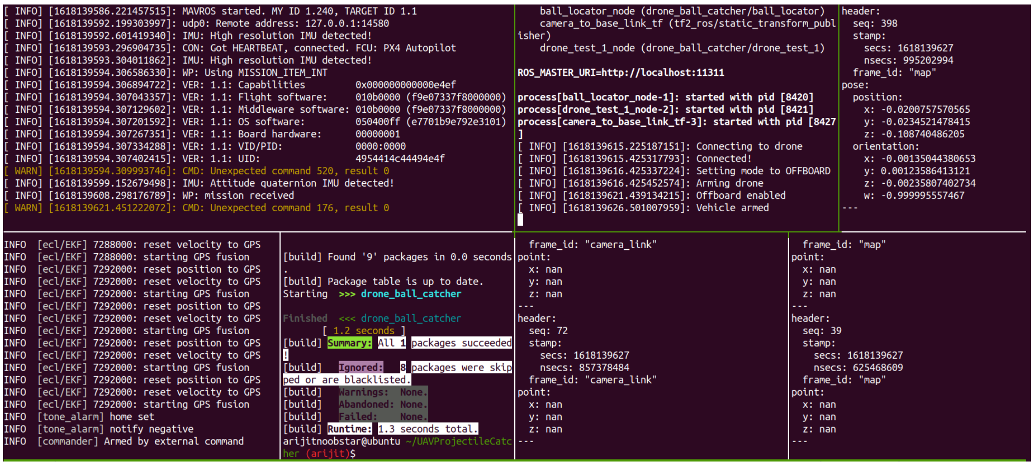

FIGURE 3.6: Dashboard of the control & software architecture

4 Results

4.1 Scenario A

To begin testing the simulations, we wanted to start at the simplest level of object detection and drone path planning. A stationary ball was placed in the simulation environment with the gravity on the ball turned off. We pointed the drone's depth camera directly at the ball and when the ball was detected by the object detection software, the drone was given the depth camera frame coordinates of the ball and instructed to move towards it. Because the ball was placed directly in the line of sight of the drone, there was very little complications for drone path planning. Figure 4.1 shows a comparison of the camera view of the drone before and after object detection of the ball.

FIGURE 4.1: Comparison of the view of the ball from the camera before (left) and after ball detection (right)

Drone path planning was done by adding the ball's location relative to the drone's location, which gives the ball's world frame location. This calculated position was sent to the PX4 autopilot via a MAVROS topic called `mavros/setpoint_position/local`, instructing the drone to move to that position. When the depth camera identifies that the ball's position relative to the drone was less than 20cm, we considered Scenario A a success and instructed the drone to hover in place.

The reason the ball's relative position can be added to the drone's location is due to the fact that both the world frame and the camera frame are oriented in the same direction as shown in Figure 4.2, simplifying the drone-ball system to a translational 2-dimensional system. Hence, we are allowed to translate between reference frames by adding the vectored difference in distance between the two frames.

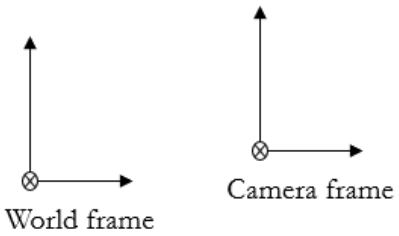

FIGURE 4.2: Translational 2 dimensional frame transformation

While Scenario A was straightforward, it provided valuable insights that would further more complex scenarios. Firstly, we realised that for the drone to accelerate, it had to pitch at an angle. By tilting, it can cause the camera to lose sight of the ball. This was easily rectified with two options. The angle of pitch is measured through the internal IMU sensor, and the drone's height would be increased proportionately using the previously recorded ball location such that the ball is in view again. The other option was to reduce the drone's acceleration, reducing the thrust vectoring and hence the pitch angle needed. For Scenario A, the latter option was chosen. Secondly, it was discovered that the ball detection algorithm in the simulation is powerful enough that it can detect a ball up to 20m away from the drone.

The successes of Scenario A are shown using RVIZ, a ROS visualisation software, where the red line represents the path of the drone:

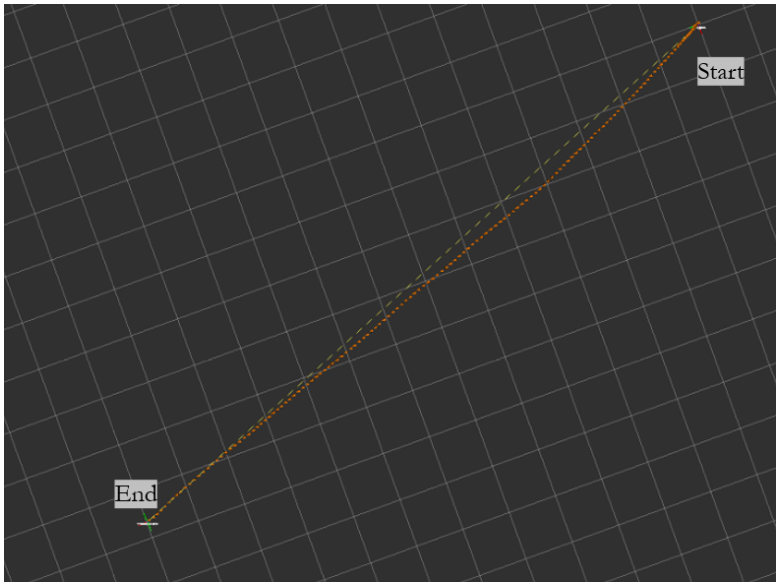

FIGURE 4.3: Drone moves directly towards the ball placed directly at the centre of the field of view

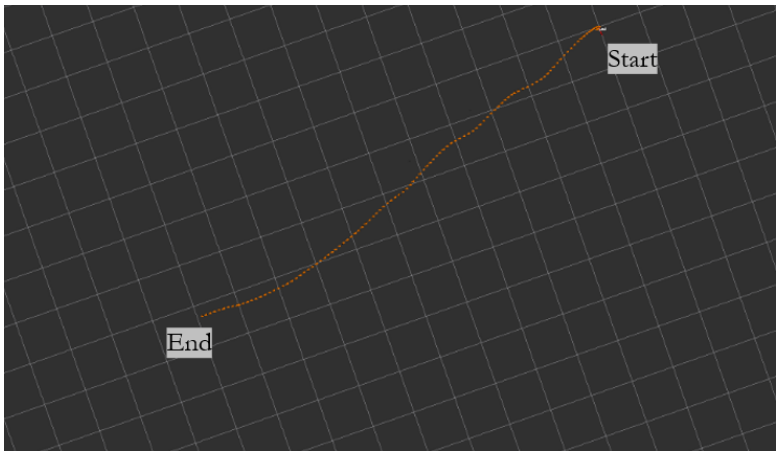

FIGURE 4.4: Drone takes a curved and shaky path towards a ball placed off the centre of the field of view

4.2 Scenario B

Scenario B is similar to Scenario A, we place a ball some distance away and in the line of sight of the drone. However, we also introduce a motion to the ball. The algorithm for Scenario B is similar to Scenario A, the ball's relative position to the drone is added vectorily to the drone's position, giving us the ball's absolute position. The ball's absolute position is then sent to the `mavros/setpoint_position/local` topic, instructing the drone to the position of the ball.

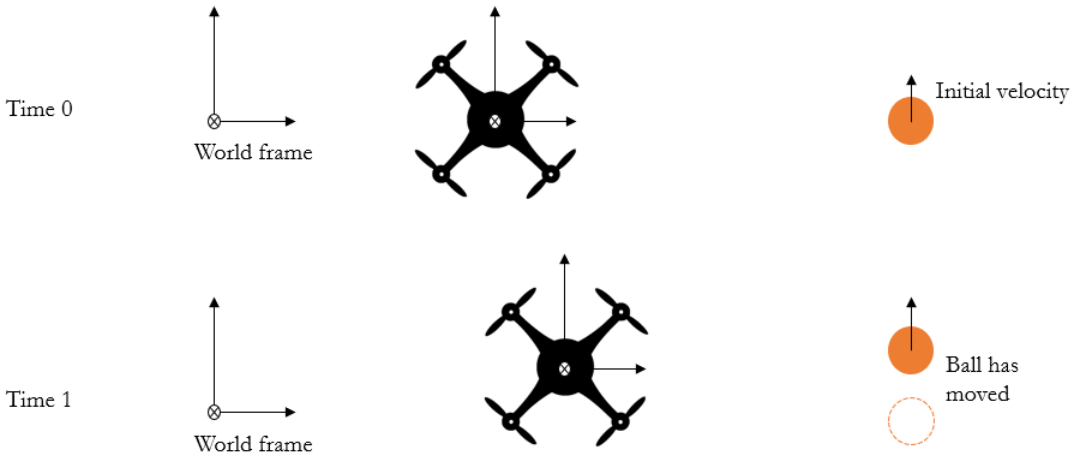

FIGURE 4.5: A moving ball forces the drone to translate sideways

Through scenario B, we were informed that we needed higher computational power than our laptop is able to provide as the frame rate of the camera was severely affected by computational power. This resulted in a decrease in simulation time to real time ratio. Consequently, we discovered that there is an upper limit to speed of the ball moving relative to the drone for object detection. This speed limit is tied to the frame rate of the depth camera, as the drone path planning can only update itself when the ball's new location is updated.

Secondly, a new problem surfaced. When the drone moves closer to the ball, the ball will travel past the camera's sensor at a faster rate. The solid angle of the depth camera measures the field of view the depth camera covers. For a fixed solid angle Ω, the amount of area A it covers on a sphere is given as:

$$\Omega = \frac{A}{r^2} \tag{4.1}$$

where r represents the distance from the depth camera's sensor. Hence, when the drone is close to the ball, the distance and area of field of view decreases. This means that the ball would very quickly move past the vision of the depth camera. At a far enough distance, the drone would translate to the side, trying to keep the ball in its field of view. However, when it gets close, the drone's speed is unable to match up to the ball, losing track of the ball as the ball crosses the camera's field of view.

The results of Scenario B are shown below.

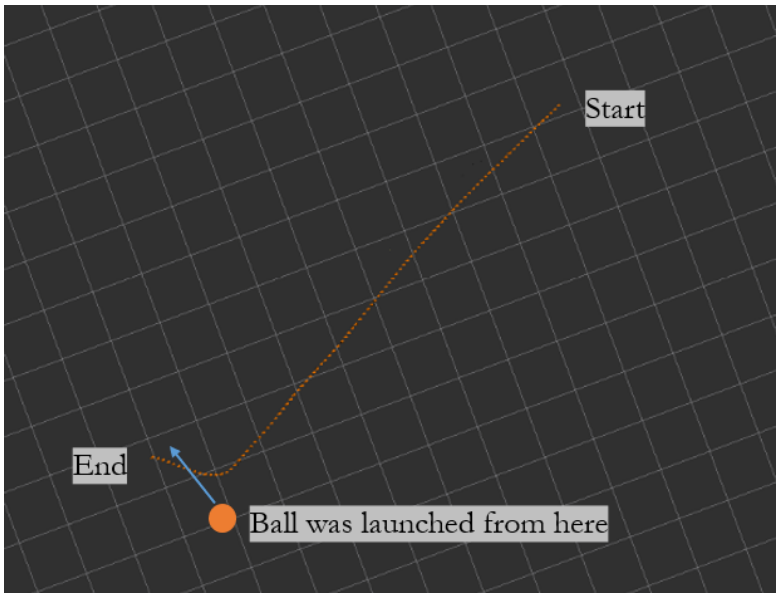

FIGURE 4.6: Drone moves towards the moving ball. It takes a sharp turn near the end to chase after the ball but is unable to intercept.

4.3 Scenario C

Building on from Scenario B, we introduce rotations and yawing. Just as the previous scenario, we place a ball some distance away and in the line of sight of the drone and introduce a motion to the ball. This time, we requiring the drone to rotate and catch the ball to circumvent the issues faced in Scenario B as shown in Figure 4.7.

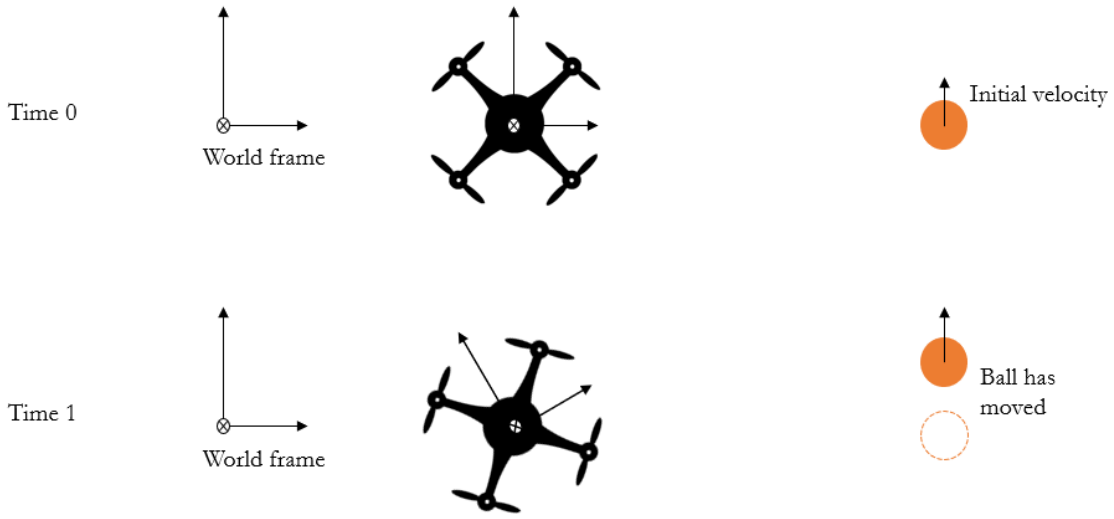

FIGURE 4.7: A moving ball forces the drone to yaw

When a drone yaws, the frame transformation between the camera and the map is no longer only translational, but also rotational, shown in Figure 4.8 below. To accommodate the rotations, we are unable to add the vectored distance between two frames, but have to consider the angle of rotation between the two frames. For this consideration, we use the *Transform Frame* library built for ROS. The *Transform Frame* library allows us to easily convert between the frames without worrying about the technicalities of rotation transformation. By using the camera's absolute quaternion values, the ball's absolute position can be known.

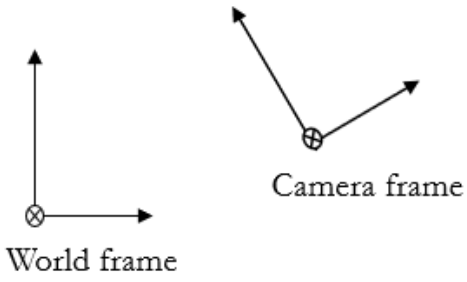

FIGURE 4.8: Rotational and translational frame transformation

To instruct the drone to yaw, a trigger is set such that if the relative ball angular position is too high, the drone will try to yaw such that the ball moves back to the centre of the camera's field of view. From Figure 4.9, when θ exceeds a certain maximum value, the drone will yaw back by $-\theta$ so that the ball will stay in the centre of the field of view of the camera.

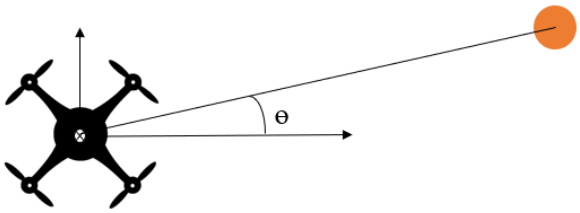

FIGURE 4.9: relative ball angular position

Scenario C was met with tremendous success as the drone reliably intercepts the ball. The results are shown below:

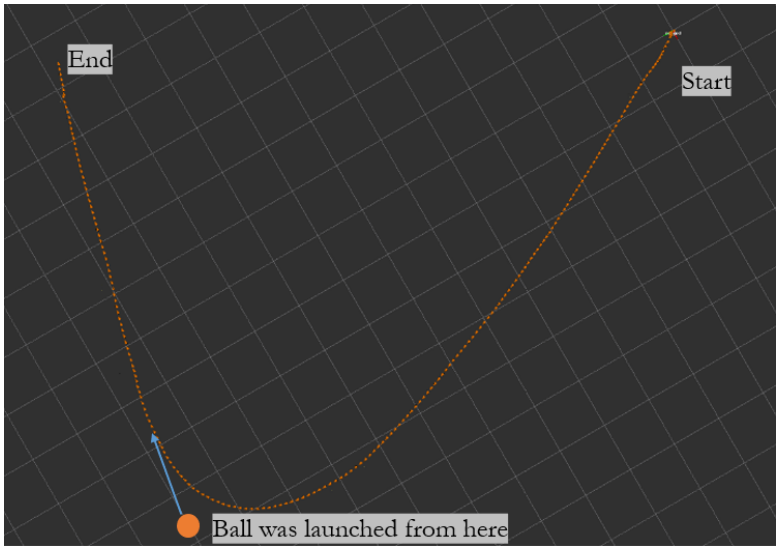

FIGURE 4.10: Drone shows a clear curved path chasing after the ball, showing excellent position and yaw control

4.4 Scenarios D & E

For scenario D E, we simulated a straight path of the ball to reduce complexity. Hence the expected drone path for scenarios D& E would be very similar even though the underlying logic is different. We managed to verify that our code for trajectory prediction was very accurate to about +- 0.1m for every 6.72m the ball was away from the drone. This was the case for an environment with gravity and artificially placed wind forces to act as drag.

Hence we were able to successfully simulate scenario D & E by making the ball move at an inclined angle to the right of the drone. The drone would accurately predict where the ball would intercept to it closest and it would move to that point and wait for the ball to hit it. As the results for both scenarios were the same, Figure 4.11 illustrates the path of the drone after predicting the path of the ball using the logic from scenario D. What was observed was that the drone was present in the predicted spot before the ball came and hit it accurately.

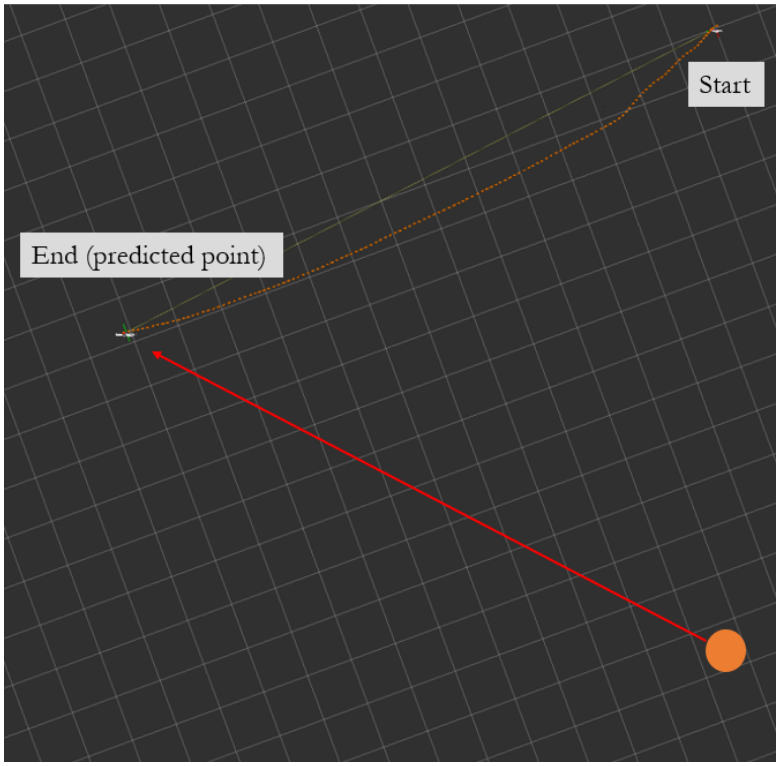

FIGURE 4.11: Drone shows a relatively straight path to a predicted point where the ball eventually ended up

What can be seen from Figure 4.11 is that the drone did make some auto-correction of the path during actual movement. Nonetheless, all the results surprised us due to the level of accuracy that we achieved.

5 Conclusion

5.1 Main Findings and limitations

Overall, we managed to simulate a drone that is able to identify and plan a path toward an object moving in a predictable manner. In this study, we were able to catch a orange table tennis ball. Using Gazebo, we simulated a drone that is able to move towards a ball using position and yaw control, as described in our different scenarios. At slower speeds, the "Cat & Mouse" method is a dependable way to catch the ball. In our simulations, we found that this method worked almost every time. However, the maximum speed of the object using this method is limited to the speed of the drone.

For an object travelling at a higher speed than the top speed of the drone, we must use methods 2 and 3, predicting the objects trajectory with either the shortest or fastest path. To predict the location of these objects, a high degree of precision of the objects current location must be known. We found that a huge limitation of the depth camera was that it was unable to provide the object location with enough precision. The recorded ball location had simply too much variance even when the ball was stationary. The variance affected the ability to perform numerical differentiation to calculate the velocity of the ball, which is required by the trajectory prediction and path planning code.

5.2 Future Works

The project has provided a good stepping stone for further research into object catching drones. In future iterations of this project, more experiments can be done to look at different depth cameras or even other types of sensors such as LiDAR. In fact, having multiple sensors on-board will allow the drone have a larger field of vision, and different filters (such as Kalman filter) can be employed to ensure that the sensor noise are reduced to a minimum. By improving the precision of our sensors, we can begin to explore different trajectory prediction and path planning codes.

Additionally, the holy grail of the project is to perform the ball (or object) catching on physical hardware. Hardware brings into another issue that was conveniently side stepped: odometry. Using the odometer of the simulated PX4 autopilot does not accurately represent a real life scenarios as it does not encounter odometry drift. The odometry drift means that the exact position of the drone is unknown. One interesting method to tackle this problem is to use LiDARs and odometry algorithms like LOAM (Lidar Odometry and Mapping).

Nonetheless, the results presented by this report exceeded our expectations. Even though this project was very challenging, we are happy to conclude that it was successful, especially in achieving our fundamental goals. We have successfully shown that ball catching need not be dependent on a motion capture room, but can rely solely on-board path planning to successful carry out the task. Appendix A contains the code for ball detection, trajectory prediction & drone path planning for scenario C. Alternatively, the code can be found on GitHub.

A Appendix

```
1
2  #include <ros/ros.h>
3  #include <pcl_ros/point_cloud.h>
4  #include <pcl/point_types.h>
5  #include <geometry_msgs/PointStamped.h>
6  #include <numeric>
7  #include <cmath>
8  #include <algorithm>
9
10 using namespace std;
11
12 typedef pcl::PointCloud<pcl::PointXYZRGB> PointCloud;
13 typedef pcl::PointXYZRGB PointData;
14 typedef Eigen::aligned_allocator<PointData> EigenPointData;
15
16 // Function Declarations
17 void callback(const PointCloud::ConstPtr&);
18 void findball(std::vector<PointData, EigenPointData>&);
19 void isolate_ball(std::vector<PointData, EigenPointData>&, float&, float&, float&);
20 float median(vector<float> &v);
21
22 // Global publisher pointers
23 ros::Publisher *ball_points_pubPtr;
24 ros::Publisher *ball_geom_pubPtr;
25
26 //Colour Range CONFIG
27 int red_lower = 150;
28 int red_upper = 255;
29 int blue_lower = 0;
30 int blue_upper = 80;
31 int green_lower = 135;
32 int green_upper = 190;
33
34
35 void isolate_ball(std::vector<PointData, EigenPointData> &Frame, float& ball_x, float& ball_y,
       float& ball_z){
36
37
38   // Assign x, y values to new vector
39   std::vector<float> xs;
40   std::vector<float> ys;
41   for (auto i = Frame.begin(); i != Frame.end(); i++){
42     xs.push_back(i->x);
43     ys.push_back(i->y);
44   }
45
46   // FINDING STDDEV
47   // Find sum
```

```
48   float sum_x = std::accumulate(xs.begin(), xs.end(), 0.0);
49   float sum_y = std::accumulate(ys.begin(), ys.end(), 0.0);
50   // Mean
51   float mean_x = sum_x / xs.size();
52   float mean_y = sum_y / ys.size();
53   // Sum of squares (std::inner_product(first1, last1, first2, init)
54   float sq_sum_x = std::inner_product(xs.begin(), xs.end(), xs.begin(), 0.0);
55   float sq_sum_y = std::inner_product(ys.begin(), ys.end(), ys.begin(), 0.0);
56   // 2 stddev using n-1 as population mean unknown (93% of points in here)
57   float stdev_x = std::sqrt(sq_sum_x / (xs.size()-1) - mean_x * mean_x);
58   float stdev_y = std::sqrt(sq_sum_y / (xs.size()-1) - mean_y * mean_y);
59
60   std::vector<PointData, EigenPointData> temp;
61   // Get rid of fringe points
62   for (auto i = Frame.begin(); i != Frame.end(); i++){
63     if (true || i->x >= mean_x - stdev_x && i->x <= mean_x + stdev_x &&
64       i->y >= mean_y - stdev_y && i->y <= mean_y + stdev_y)
65       temp.push_back(*i);
66   }
67
68   // Pass the x,y,z values back to the callback function
69   std::vector<float> newx;
70   std::vector<float> newy;
71   std::vector<float> newz;
72   //for (auto i = temp.begin(); i != temp.end(); i++){
73   for (auto i = Frame.begin(); i != Frame.end(); i++){
74     newx.push_back(i->x);
75     newy.push_back(i->y);
76     newz.push_back(i->z);
77   }
78
79   ball_x = median(newx);
80   ball_y = median(newy);
81   ball_z = median(newz);
82
83   // ball_x = std::accumulate(newx.begin(), newx.end(), 0.0) / newx.size();
84   // ball_y = std::accumulate(newy.begin(), newy.end(), 0.0) / newy.size();
85   // ball_z = std::accumulate(newz.begin(), newz.end(), 0.0) / newz.size();
86
87   // Swap the pointers
88   Frame.swap(temp);
89
90 }
91
92 float median(vector<float> &v){
93   int n = v.size()/2;
94   nth_element(v.begin(), v.begin() + n, v.end());
95   return v[n];
96 }
97
98 void findball(std::vector<PointData, EigenPointData> &Frame){
99   std::vector<PointData, EigenPointData> temp;
100  //identify ball based on colour selection
101  for (auto i = Frame.begin(); i != Frame.end(); i++){
102    if (i->r >= red_lower && i -> r <= red_upper &&
103      i->b >= blue_lower && i -> b <= blue_upper &&
104      i->g >= green_lower && i -> g <= green_upper)
105      temp.push_back(*i);
106  }
```

```
  Frame.swap(temp);
}

void callback(const PointCloud::ConstPtr& msg)
{
  // Create Frame to hold points from the RealSense camera
  std::vector<PointData, EigenPointData> Frame = msg->points;
  // Create x,y,z coordinates for ball
  float ball_x; float ball_y; float ball_z;

  //find the ball, almost, with background
  findball(Frame);
  if (Frame.size() == 0){
    ball_x = ball_y = ball_z = NAN;
  }
  else{
    isolate_ball(Frame, ball_x, ball_y, ball_z);

    // Ensure z value is not 1.0, it is a default pointcloud depth when there is an error
    if (ball_z == 1.0 || ball_z <= 0){
      return;
    }

    // Now we publish the filtered point cloud under the ball_points topic
    PointCloud::Ptr ball_points_msg (new PointCloud);
    ball_points_msg->header.frame_id = "camera_depth_optical_frame";
    ball_points_msg->height =1;
    ball_points_msg->width = Frame.size();
    ball_points_msg->points = Frame;
    ball_points_pubPtr->publish(ball_points_msg);
  }

  // Finally, publish the x,y,z coordinates of the ball centroid
  geometry_msgs::PointStamped::Ptr ball_geom_msg (new geometry_msgs::PointStamped);
  ball_geom_msg->header.frame_id = "camera_link";
  ball_geom_msg->header.stamp = ros::Time::now();
  ball_geom_msg->point.x = ball_x;
  ball_geom_msg->point.y = ball_y;
  ball_geom_msg->point.z = ball_z;
  ball_geom_pubPtr->publish(ball_geom_msg);
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "ball_locator_node");
  ros::NodeHandle nh;

  // extract colour parameters (stored in config/ball_colour.yaml)
  nh.getParam("red_lower", red_lower);
  nh.getParam("red_upper", red_upper);
  nh.getParam("green_lower", green_lower);
  nh.getParam("green_upper", green_upper);
  nh.getParam("blue_lower", blue_lower);
  nh.getParam("blue_upper", blue_upper);

  ros::Subscriber sub = nh.subscribe<PointCloud>("/camera/depth/points", 10000, callback);
  ball_points_pubPtr = new ros::Publisher(nh.advertise<PointCloud>("ball_points", 10000));
  ball_geom_pubPtr = new ros::Publisher(nh.advertise<geometry_msgs::PointStamped>("ball_geom",
     10000));
```

```

  ros::spin();

  delete ball_points_pubPtr;
  delete ball_geom_pubPtr;
}
```

Listing A.1: Ball Detection Code

```

#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <pcl/point_types.h>
#include <geometry_msgs/Point.h>
#include <geometry_msgs/PointStamped.h>
#include <geometry_msgs/PoseArray.h>
#include <cmath>
#include <string>
#include <deque>

using namespace std;

struct point_t{
  float x;
  float y;
  float z;
  double t;
};

// CONFIG
const int frame_cycle = 5; // max number of frames in queue for velocity estimation, a highher value reduces random error, but is less sensitive to acceleration changes
const double TIME_STEP = 0.1; // in seconds
const double MAX_TIME_PATH_PLANNING = 100; // in seconds
const double PI = 3.1415265;
const double MASS = 0.0027; // in KG
const double G = 9.781;// approx G in Singapore (in m/s^2)
const double DRAG_COEF = 0.5; // for spheres at approx Reynold Number 5000
const double BALL_DIAMETER = 0.04; // in metres
const double AIR_DENSITY = 1.225; // in kg/m^3
string goal_type = "plane_fixed";

// Global publisher pointer
ros::Publisher *setpoint_pubPtr;

// Create deque for the points and other global variables
deque<point_t> PoseData;
geometry_msgs::Point::Ptr goal (new geometry_msgs::Point); // setpoint location for drone
double t_0; // initial time
double mag_V, v_x, v_y, v_z; // velocities
double phi, theta; // angles (phi is Z to X axis and theta is X to Y axis)
double sum_t, sum_t_2, sum_x, sum_y, sum_z, sum_tx, sum_ty, sum_tz; // intermediate values for linear regression
double drag, s_x, s_y, s_z, a_x, a_y, a_z; // trajectory prediction intermediate variables
bool goal_uncheck = true; // bool variable check for iterative algorithm to stop

// Function Declarations
void callback(const geometry_msgs::PointStamped::ConstPtr& msg);
void add_point(const geometry_msgs::PointStamped&);
```

```
49 void compute_velocities();
50 void predict_trajectory(const geometry_msgs::PointStamped::ConstPtr& msg);
51 bool implement_plane_fixed_goal();
52
53 bool plane_fixed_path_planning(){
54   if (s_z <= 0){
55     // set the goal setpoint for the drone
56     goal->x = s_x;
57     goal->y = s_y;
58     goal->z = 0;
59     return false;
60   }
61   else{
62     return true;
63   }
64 }
65
66 void compute_velocities(){
67
68   // computes velocities using least squares linear regression
69   // set intermediate values as 0.0;
70   sum_t = sum_t_2 = sum_x = sum_y = sum_z = sum_tx = sum_ty = sum_tz = 0.0;
71   // loop through PoseData to calculate intermediate values
72   for (int i = 0; i < PoseData.size(); ++i){
73     sum_t += PoseData[i].t;
74     sum_t_2 += pow(PoseData[i].t, 2);
75     sum_x += PoseData[i].x;
76     sum_y += PoseData[i].y;
77     sum_z += PoseData[i].z;
78     sum_tx += (PoseData[i].t * PoseData[i].x);
79     sum_ty += (PoseData[i].t * PoseData[i].y);
80     sum_tz += (PoseData[i].t * PoseData[i].z);
81   }
82   // Apply least squares formula to calculate the slopes (velocity)
83   v_x = ((PoseData.size() * sum_tx) - (sum_x * sum_t)) / ((PoseData.size() * sum_t_2) - pow(
       sum_t, 2));
84   v_y = ((PoseData.size() * sum_ty) - (sum_y * sum_t)) / ((PoseData.size() * sum_t_2) - pow(
       sum_t, 2));
85   v_z = ((PoseData.size() * sum_tz) - (sum_z * sum_t)) / ((PoseData.size() * sum_t_2) - pow(
       sum_t, 2));
86
87 }
88
89 void predict_trajectory(const geometry_msgs::PointStamped::ConstPtr& msg){
90   // create vector for predicted trajectory
91   vector<point_t> PredictedPose;
92   // Calculate the maximum iteration count based on slowest speed approximation
93   double max_count = MAX_TIME_PATH_PLANNING / TIME_STEP;
94   int count = 0;
95
96   s_x = msg->point.x;
97   s_y = msg->point.y;
98   s_z = msg->point.z;
99
100  cout << "VEL" << v_x << ',' << v_y << ',' << v_z << endl;
101
102  while(goal_uncheck && count <= max_count){
103    // calculate the velocity magnitude, vector angles, acclerations wrt point cloud frame
104    mag_V = sqrt(pow(v_x, 2) + pow(v_y, 2) + pow(v_z, 2));
```

```
105     phi = atan2(v_x, v_z);
106     theta = atan2(v_y, v_x);
107     drag = 0.5 * AIR_DENSITY * DRAG_COEF * pow(mag_V, 2) * PI * pow(BALL_DIAMETER / 2, 2);
108     a_x = - (drag / MASS) * sin(phi) * cos(theta);
109     a_y = G - (drag / MASS) * sin(phi) * sin(theta);
110     a_z = - (drag / MASS) * cos(phi) * cos(theta);
111     // caclulate the displacements in x,y,z direction after one time step using the equation
        of motion (assuming constant velocity throughout the time step)
112     s_x = s_x + v_x * TIME_STEP + 0.5 * a_x * pow(TIME_STEP, 2);
113     s_y = s_y + v_y * TIME_STEP + 0.5 * a_y * pow(TIME_STEP, 2);
114     s_z = s_z + v_z * TIME_STEP + 0.5 * a_z * pow(TIME_STEP, 2);
115     // update velocities in 3 directions after one time step (assuming constant acceleration)
116     v_x = v_x + a_x * TIME_STEP;
117     v_y = v_y + a_y * TIME_STEP;
118     v_z = v_z + a_z * TIME_STEP;
119     // check to see if iterative steps should cease
120     if (goal_type == "plane_fixed"){
121       goal_uncheck = plane_fixed_path_planning();
122     }
123     count++;
124     if (count <= max_count){
125       cout << count << ',' << s_x << ',' << s_y << ',' << s_z << endl;
126     }
127 
128   }
129 }
130 
131 void add_point(const geometry_msgs::PointStamped& pt){
132   // Add point to PoseData deque
133   point_t temp_point_t;
134   temp_point_t.x = pt.point.x;
135   temp_point_t.y = pt.point.y;
136   temp_point_t.z = pt.point.z;
137   // use ROS time to input time
138   if (PoseData.size() == 0){
139     t_0 = ros::Time::now().toSec();
140     temp_point_t.t = 0.0;
141   }
142   else{
143     temp_point_t.t = ros::Time::now().toSec() - t_0;
144   }
145   // push to the front
146 
147   PoseData.push_front(temp_point_t);
148 }
149 
150 
151 void callback(const geometry_msgs::PointStamped::ConstPtr& msg)
152 {
153   if (PoseData.size() < frame_cycle){
154     add_point(*msg);
155     // ensure that at least two data points are present;
156     if (PoseData.size() < 2){
157       return;
158     }
159   }
160   else{
161     // remove the first point in the frame cycle and add the new point (like a queue)
162     PoseData.pop_back();
```

```
    add_point(*msg);
  }
  // set default goal->x setpoint to NAN first (math.h macro)
  goal->x = goal->y = goal->z = NAN;
  // calculate the velocities in 3 directions and get the Magnitude of V and its angles
  compute_velocities();
  // make sure ball is moving towards drone (0.1m/s clearance given)
  if (v_z > -0.1){
    return;
  }
  // predict the trajectory of ball and conduct drone path planning and extract goal setpoint
  predict_trajectory(msg);
  // publish goal setpoint
  if(true || !isnan(goal->x)){
    setpoint_pubPtr->publish(goal);
  }
  // reset goal_uncheck to true
  goal_uncheck = true;
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "ball_trajectory_node");
  ros::NodeHandle nh;

  ros::Subscriber sub = nh.subscribe<geometry_msgs::PointStamped>("ball_geom", 10000, callback
    );
  setpoint_pubPtr = new ros::Publisher(nh.advertise<geometry_msgs::Point>("predicted_setpoint"
    , 10000));

  ros::spin();

  delete setpoint_pubPtr;
}
```

Listing A.2: Trajectory Prediction Code

```

#include <ros/ros.h>
//#include <math.h>
#include <cmath>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/Point.h>
#include <geometry_msgs/Quaternion.h>
#include <geometry_msgs/PointStamped.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/TwistStamped.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
// Added tf2 headers to hold world frames
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2_ros/transform_listener.h>
#include <tf2_ros/transform_broadcaster.h>
#include <tf2/LinearMath/Quaternion.h>
#include <geometry_msgs/TransformStamped.h>

using namespace std;
```

```
23
24 //global
25 bool look_for_ball = false;
26 float collsion_threshold = 0.2;
27 tf2_ros::Buffer tfBuffer;
28 tf2_ros::TransformListener *tfListener;
29 ros::Publisher *pubber;
30
31 mavros_msgs::State current_state;
32 void state_cb(const mavros_msgs::State::ConstPtr& msg){
33     current_state = *msg;
34 }
35
36 geometry_msgs::PoseStamped current_location;
37 void pose_cb(const geometry_msgs::PoseStamped::ConstPtr& msg)
38 {
39     current_location = *msg;
40 }
41
42 geometry_msgs::Point ball_location;
43 geometry_msgs::Point ball_location_relative;
44 void ball_cb(const geometry_msgs::PointStamped::ConstPtr& msg)
45 {
46     // cout << "sub run" << endl;
47     // ball location is in camera_link frame now
48     // set this relative location to ball_location_relative
49     look_for_ball = true;
50     ball_location_relative.x = msg->point.x;
51     ball_location_relative.y = msg->point.y;
52     ball_location_relative.z = msg->point.z;
53     // converting ball_location to map frame
54
55     try{
56         //transformStamped = tfBuffer.lookupTransform("/map", "/camera_link", ros::Time(0));
57         //tf2_geometry_msgs::do_transform(*msg, ball_location_map, transformStamped);
58         geometry_msgs::PointStamped ball_location_map;
59
60         tfBuffer.transform(*msg, ball_location_map, "map");
61
62         // ensure drone is not told to go underground
63         if (ball_location_map.point.z <= 0){
64             return;
65         }
66
67         ball_location.x = ball_location_map.point.x;
68         ball_location.y = ball_location_map.point.y;
69         ball_location.z = ball_location_map.point.z;
70
71         geometry_msgs::PointStamped::Ptr pub (new geometry_msgs::PointStamped);
72         pub->header.frame_id = "map";
73         pub->header.stamp = ros::Time::now();
74         pub->point.x = ball_location.x;
75         pub->point.y = ball_location.y;
76         pub->point.z = ball_location.z;
77         pubber->publish(pub);
78     }
79     catch (tf2::TransformException &ex) {
80         //ROS_WARN("%s",ex.what());
81         //ros::Duration(1.0).sleep();
```

```
    }
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "drone_test_1_node");
    ros::NodeHandle nh;

    // TF listener
    tfListener = new tf2_ros::TransformListener(tfBuffer);

    // SUBSCRIBERS
    ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>
            ("mavros/state", 1000, state_cb);
    ros::Subscriber local_pos_sub = nh.subscribe<geometry_msgs::PoseStamped>
            ("mavros/local_position/pose", 1000, pose_cb);
    ros::Subscriber ball_locator_sub = nh.subscribe<geometry_msgs::PointStamped>
            ("ball_geom", 1000, ball_cb);

    // PUBLISHERS
    ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>
            ("mavros/setpoint_position/local", 10);
    ros::Publisher vel_pub = nh.advertise<geometry_msgs::TwistStamped>("/mavros/
    setpoint_velocity/cmd_vel", 1000);
    pubber = new ros::Publisher(nh.advertise<geometry_msgs::PointStamped>("check_coord", 1000)
    );

    // SERVICES
    ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>
            ("mavros/cmd/arming");
    ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>
            ("mavros/set_mode");

    // extract initial pose parameters (stored in config/initial_drone_pose.yaml)
    float init_x, init_y, init_z, init_qx, init_qy, init_qz, init_qw = 0.0;
    bool move_drone = false;
    int min_readings;
    int initial_min_readings;
    int reading_count = 0;
    float theta_max = 0.25; // radians of maximum allowed yaw deviation from camera center
    float theta;
    geometry_msgs::Point ball_location_relative_last;
    ball_location_relative_last.x = NAN;
    ball_location_relative_last.y = NAN;
    ball_location_relative_last.z = NAN;
    nh.getParam("init_x", init_x);
    nh.getParam("init_y", init_y);
    nh.getParam("init_z", init_z);
    nh.getParam("init_qx", init_qx);
    nh.getParam("init_qy", init_qy);
    nh.getParam("init_qz", init_qz);
    nh.getParam("init_qw", init_qw);
    nh.getParam("min_readings", min_readings);
    nh.getParam("initial_min_readings", initial_min_readings);

    // set variables to look for ball and to move drone


    ball_location.x = NAN;
```

```
139     ball_location.y = NAN;
140     ball_location.z = NAN;
141     // ********************************************** START OF DRONE SETUP
        **********************************************
142
143     //the setpoint publishing rate MUST be faster than 2Hz
144     ros::Rate rate(3.5);
145
146     ROS_INFO("Connecting to drone");
147     // wait for FCU connection
148     while(ros::ok() && !current_state.connected){
149         ros::spinOnce();
150         rate.sleep();
151     }
152     ROS_INFO("Connected!");
153
154
155     geometry_msgs::TwistStamped msg;
156     msg.twist.linear.x = 0;
157     msg.twist.linear.y = 0;
158     msg.twist.linear.z = 0;
159
160
161     //send a few setpoints before starting
162     for(int i = 20; ros::ok() && i > 0; --i){
163         vel_pub.publish(msg);
164         ros::spinOnce();
165         rate.sleep();
166     }
167
168     ROS_INFO("Setting mode to OFFBOARD");
169     mavros_msgs::SetMode offb_set_mode;
170     offb_set_mode.request.custom_mode = "OFFBOARD";
171
172     ROS_INFO("Arming drone");
173     mavros_msgs::CommandBool arm_cmd;
174     arm_cmd.request.value = true;
175
176     ros::Time last_request = ros::Time::now();
177
178
179     geometry_msgs::PoseStamped initial_location;
180     initial_location.pose.position.x = init_x;
181     initial_location.pose.position.y = init_y;
182     initial_location.pose.position.z = init_z;
183
184     initial_location.pose.orientation.x = init_qx;
185     initial_location.pose.orientation.y = init_qy;
186     initial_location.pose.orientation.z = init_qz;
187     initial_location.pose.orientation.w = init_qw;
188
189     geometry_msgs::PoseStamped target_location;
190     target_location.header.frame_id = "base_link";
191
192     geometry_msgs::PoseStamped target_location_last;
193     target_location_last = initial_location;
194
195     while(ros::ok()){
196         if( current_state.mode != "OFFBOARD" &&
```

```
            (ros::Time::now() - last_request > ros::Duration(1.0))){
            if( set_mode_client.call(offb_set_mode) &&
                offb_set_mode.response.mode_sent){
                ROS_INFO("Offboard enabled");
            }
            last_request = ros::Time::now();
        } else {
            if( !current_state.armed &&
                (ros::Time::now() - last_request > ros::Duration(1.0))){
                if( arming_client.call(arm_cmd) &&
                    arm_cmd.response.success){
                    ROS_INFO("Vehicle armed");
                }
                last_request = ros::Time::now();
            }
        }
        // ********************************************** END OF DRONE SETUP
    **********************************************
        // add code for drone movement here

        if (!look_for_ball || isnan(ball_location.x) || isnan(ball_location.y) || isnan(
    ball_location.z)){
            // move_drone = false;
        }
        else if (pow(ball_location_relative.x,2) + pow(ball_location_relative.y,2) + pow(
    ball_location_relative.z,2) > pow(collsion_threshold,2)){

            if (move_drone && reading_count < min_readings){
                reading_count++;
            }
            else if (!move_drone && reading_count < initial_min_readings){
                reading_count++;
            }

            else{
                move_drone = true;
                // if ball coords are still nan, use previous non-nan target
                if (isnan(ball_location.x) || isnan(ball_location.y) || isnan(ball_location.z)
    ){
                    target_location.pose.position.x = target_location_last.pose.position.x;
                    target_location.pose.position.y = target_location_last.pose.position.y;
                    target_location.pose.position.z = target_location_last.pose.position.z;
                    target_location.pose.orientation.x = init_qx;
                    target_location.pose.orientation.y = init_qy;
                    target_location.pose.orientation.z = init_qz;
                    target_location.pose.orientation.w = init_qw;
                    ROS_WARN("Ball out of sight");
                }
                // otherwise use current ball_location and update target_location_last to
    current ball_location
                else{


                    theta = atan2( -1 * ball_location_relative.x, ball_location_relative.z);

                    if (abs(theta) > theta_max && !(ball_location_relative.x ==
    ball_location_relative_last.x &&
                        ball_location_relative.y == ball_location_relative_last.y &&
    ball_location_relative.y == ball_location_relative_last.y)){
```

```
249                         if (theta < 0){
250                             ROS_INFO("Yawing Right");
251                         }
252                         else{
253                             ROS_INFO("Yawing Left");
254                         }
255                         tf2::Quaternion Qoriginal(current_location.pose.orientation.x,
    current_location.pose.orientation.y,
256                         current_location.pose.orientation.z, current_location.pose.orientation
    .w);
257                         tf2::Quaternion Qchange;
258                         Qchange.setRPY(0, 0, theta/2);
259                         Qoriginal *= Qchange;
260 
261 
262 
263                         geometry_msgs::Quaternion Qorigmsg = tf2::toMsg(Qoriginal);
264                         init_qz = Qorigmsg.z;
265                         init_qw = Qorigmsg.w;
266 
267                         // cout << ball_location_relative.x << "," << ball_location_relative.z
     << "," << init_qz << "," << init_qw
268                         // << endl;
269 
270                     }
271 
272                     ball_location_relative_last = ball_location_relative;
273 
274                     target_location.pose.position.x = ball_location.x;
275                     target_location.pose.position.y = ball_location.y;
276                     target_location.pose.position.z = ball_location.z; //ball_location.z;
277                     target_location.pose.orientation.x = init_qx;
278                     target_location.pose.orientation.y = init_qy;
279                     target_location.pose.orientation.z = init_qz;
280                     target_location.pose.orientation.w = init_qw;
281 
282                     target_location_last.pose.position.x = ball_location.x;
283                     target_location_last.pose.position.y = ball_location.y;
284                     target_location_last.pose.position.z = ball_location.z;
285                     target_location_last.pose.orientation.x = init_qx;
286                     target_location_last.pose.orientation.y = init_qy;
287                     target_location_last.pose.orientation.z = init_qz;
288                     target_location_last.pose.orientation.w = init_qw;
289                     //cout << "Moving to location " << target_location.pose.position.x << ","
    << target_location.pose.position.y << "," << target_location.pose.position.z << endl;
290                     // cout << theta << endl;
291                 }
292                 // look_for_ball = false;
293                 // ROS_INFO("Ball Detected");
294                 reading_count = 0;
295             }
296         }
297         else if (pow(ball_location_relative.x,2) + pow(ball_location_relative.y,2) + pow(
    ball_location_relative.z,2) <= pow(collsion_threshold,2)){
298             // ball has definitely reached the target
299             ROS_INFO("Drone has hit target! Resetting Drone");
300             move_drone = false;
301             look_for_ball = false;
302         }
```

```
303         else{
304             ROS_INFO("Error in code");
305         }
306 
307         if (move_drone){
308             local_pos_pub.publish(target_location);
309         }
310         else{
311             local_pos_pub.publish(initial_location);
312         }
313 
314         ros::spinOnce();
315         rate.sleep();
316     }
317 
318     return 0;
319 }
```

Listing A.3: Drone Path Planning (Scenario C)

Bibliography

[1] Z. Ai, M. A. Livingston, and I. S. Moskowitz, "Real-time unmanned aerial vehicle 3d environment exploration in a mixed reality environment," in *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2016, pp. 664–670.

[2] M. Hehn and R. D'Andrea, "A flying inverted pendulum," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 763–770.

[3] F. Augugliaro, A. P. Schoellig, and R. D'Andrea, "Dance of the flying machines: Methods for designing and executing an aerial dance choreography," *IEEE Robotics & Automation Magazine*, vol. 20, no. 4, pp. 96–104, 2013.

[4] UntitledTitle, "Quadrocopter ball juggling, eth zurich," Mar 2011. [Online]. Available: https://www.youtube.com/watch?v=3CR5y8qZf0Y

[5] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[6] F. A. Morrison, "Data correlation for drag coefficient for sphere," *Department of Chemical Engineering, Michigan Technological University, Houghton, MI*, vol. 49931, 2013.