CS5335 Final Project Report

Vinay Srampickal Joseph and Carter Ithier

April 2020

1 Duckie Drone

1.1 Introduction

Our objective was to build an autonomous drone that takes off from the floor and maintains an predetermined altitude. Once, we got that working we planned to make the drone fly in a trajectory and locate a marked position denoting a helipad, and land on it smoothly. For the project we have used the Duckiedrone kit. Duckiedrone is a small autonomous Raspberry Pi drone kit which is part of the Duckietown platform. Duckietown is based on an MIT engineering course (2.166). We would be following the Brown University course, CS1951R, to build the quad-copter and program it. The goal of our final project was as follows:

- Build the drone using the steps in the operation manual and test it with the provided code
- Fine tune the PID sensors
- Do motion planning
- Using the Pi Cam to locate the landing zone and land safely

To accomplish this goal, we came up with several manageable subtasks. We were only able to implement the first five tasks and started progress on the sixth one. The subtasks are as follows:

1. *Getting familiar with ROS*

   Go through tutorials and videos available online to learn ROS

2. *Assemble the drone using the operation manual.*

3. *Configuring the drone*

   Flash the flight controller firmware and configure it

   Performed calibration on the ESCs

   Install Ubuntu and other software on the Raspberry Pi

4. *Connecting to the drone*

   Use SSH to connect to the drone through WiFi

   Connect the drone to the Home Network

5. *Testing the flight of the quadcopter using the web interface*

   Run the script provided with the operation manual to start the quad-copter

   Test the various controls of the drone using the web interface for the quad-copter test

6. *Tuning of the Planar, Altitude and Position Sensor*

   Implement a discretized version of the PID control function

Initialize the PID gain constants so that the drone can take-off smoothly and hover at the correct altitude steadily and without any drift

7. *Robot Motion*

   Get the drone to take off, hover, and land autonomously

   Make the quadcopter fly in a square pattern and land autonomously

8. *Track the landing zone using the PiCam and land on that target*

   Use OpenCV to identify and track the landing zone

   Input: The image stream from the Pi-cam

   Output: The position of the landing zone with respect to the drone

1.2 Approach

The following section describes our approach to completing the various subtasks and the problems we encountered. Both of us worked together on all of the subtasks described here.

1.2.1 Getting familiar with ROS

We had spent about 2 to 3 weeks on this section because we were both new to ROS and we had some time on our hand before we got the duckiedrone kit. There are some tutorials available on the ROS Wiki website [1] that we found extremely helpful. They have a very good section for beginners, in which they have explained everything from how to install ROS to using the robot in simulation. Another course we found useful was the Hello (Real) World with ROS - Robot Operating System program [2] provided by edX. We also briefly looked into a simulation environment for drones [3] on ROS Wiki while we were getting acquainted with ROS.

1.2.2 Assemble the drone using the operation manual.

The operation manual [4] for the duckiedrone is quite elaborate and accurate on how to assemble a drone. Professor Wong had ordered for us all the components mentioned in the Parts and Materials List page on the operation manual as well as the required tools to build the drone. The Pi mount that we used to screw the raspberry pi to the drone body had to be 3D printed. The 3D object file that has to be printed is included with the pidrone package. The 3D printing was done by Ziyi Yang, the Teaching Assistant for the Robotics Science and Systems course.

For future reference, double check the model numbers and specification of the parts before ordering. One of the major setbacks we faced initially was that we had ordered a Raspberry Pi 3 Model B+ instead of the Raspberry Pi 3 model B. The link provided on the operation manual was wrong and pointed us to the incorrect Pi, which resulted in a lot of compatibility issues with software as described in Section 1.2.3. To circumvent these issues, we ended up buying the Raspberry Pi 3 model B and swapping out the B+ model on the drone for that.

Also, be careful of what gauge wires you solder to the ESCs. The directions do not mention which of the two different gauged wires to use. The correct wires to use are the thicker ones.

The actual assembling process took us around 2 full days. The drone being assembled and the finish product can be seen in Figures 1 and 2, respectively. Luckily, we had got all of the parts before the spring break so we were able to spend part of the break assembling the drone. We followed the instructions given in the operation manual [4] and did not have to refer anything else as it had tutorials on how to use a solder iron and multimeter. One of the main thing to remember is to make sure that there is no loose components and all the wires are tied in place using zip ties so that it doesn't get cut by the propellers.

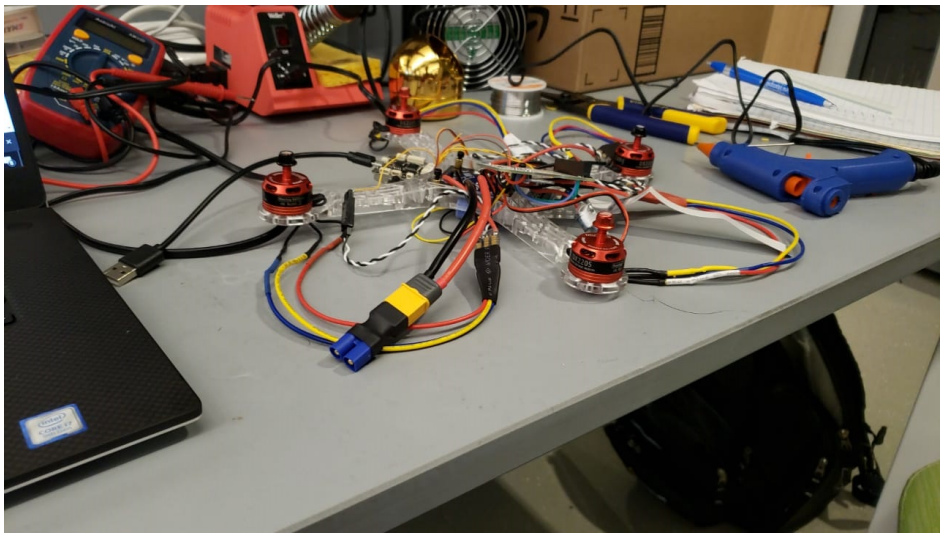


Figure 1: The drone in the process of being assembled

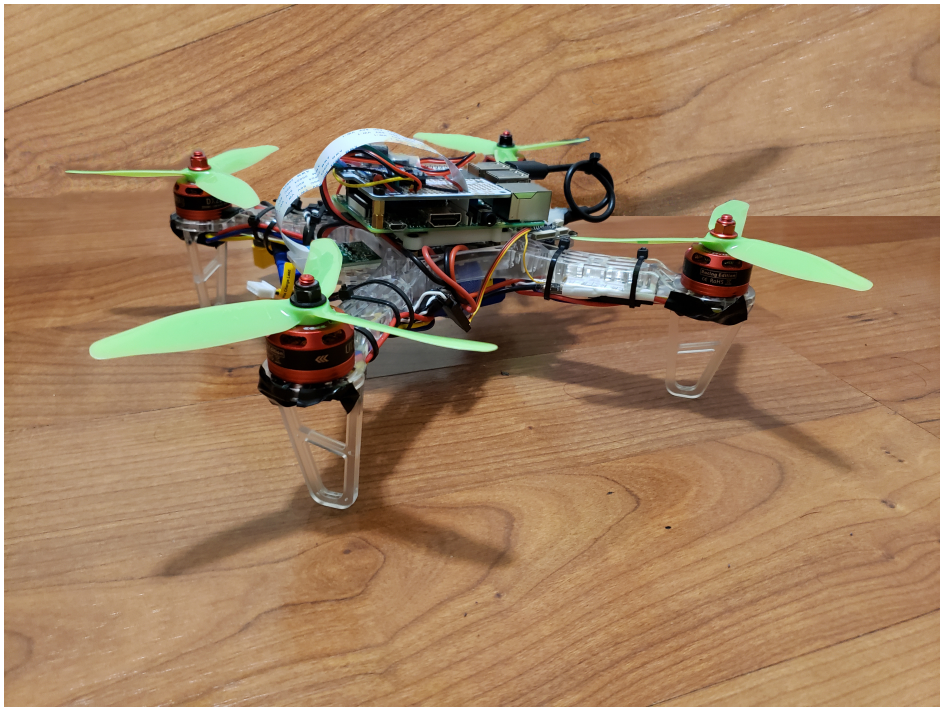


Figure 2: The assembled drone

1.2.3 Configuring the drone

First, we flashed the Flight controller using the program called Cleanflight [5]. The process to be followed is available in the operation manual [4]. Then we configured the Flight Controller again by using the Cleanflight program. Before starting this step always remember to remove the propellers. We don't want the drone flying off with our workstation. The next step was to test the motors, this can be done by running individual motors using the Cleanflight program. To better visualise the direction they are rotating in, we kept a paper clip on the motor instead of the rotor. The operation manual has instructions on how to fix the motors if the directions of rotation are wrong. Adjacent motors should rotate in opposite directions. The motors were all rotating at different speeds when at maximum power which we fixed by calibrating the ESCs, again using Cleanflight program. The steps in the operation manual [4] made us think that this process would take a lot of time, but we got it done easily without any issues.

Next we had to install Ubuntu in our raspberry pi. An image [6] had been provided with the duckiedrone with all the necessary packages pre-installed on it. We used balena Etcher [7], an image flashing tool, to flash the SD card with the OS. We imagined installing an OS to be pretty straightforward such as installing an OS in our workstation. This step held us up for the next 2 full days. The Raspberry Pi 3 model B+ which we had with us at this time would not boot the Ubuntu 16.04 image that was provided with the drone. We attempted installing a fresh image of Ubuntu 16.04 which we download from Ubuntu's official website. This too didn't work. On further researching we found out that official Ubuntu 16.04 does not support the armhf processor used in the newer model B+. We tried installing a Ubuntu 18.04 version, this works, but the code for the quadcopter provided used ROS Kinetic which doesn't work with Ubuntu 18.04. So, we found a version of Ubuntu 16.04 that actually worked on it after many failed attempts. We found a version of Ubuntu provided by Ubiquity Robotics that was compatible with our Pi. Now, we were able to boot into the pi, but some of the packages that had to be installed on the drone were not pre-installed. These three libraries in particular were *Adafruit_ADS1X15*, *web_video_server*, and *filterpy* packages. Two of the required packages mentioned, *Adafruit_ADS1X15* and *filterpy*, had stopped support for python 2.7. *filterpy* was used by the Kalman filter for the drone and *Adafruit_ADS1X15* was used for obtaining readings from the IR sensor. We were not able to find any solutions to intsalling these two packages. We were, however, able to install *web_video_server* which is needed for streaming the video over HTTP. This could be installed using the command *sudo apt-get install ros-kinetic-web-video-server*. This laid out a very simple problem in front of us, keep using the model B+ and keep troubleshooting all the issues that come our way or order a new model B raspberry pi. We went ahead with the easier option.

The new Raspberry Pi which we ordered took 2 to 3 days to arrive, and the Ubuntu image provided with the drone booted on the Pi without any hindrance.

1.2.4 Connecting to the drone

The steps for connecting to the drone were a bit vague in the operation manual [4]. By default the SSID of the drone is *defaultdrone* and the password is *bigbubba*. Actually, every password used in the drone is by default set as *bigbubba*. This information was not available in the operation manual and took some time to find out. After connecting to the WiFi network we had to SSH into the drone using the command *ssh duckiesky@192.168.42.1*. The password that has to be entered at the prompt is the default one. We can change the name of the drone's WiFi network by following the steps in the operation manual. To connect the drone to our home network the steps in the operation manual didn't seem to work for us. Given below are the steps we followed for this.

1. In the folder */etc/wpa_supplicant* create a new file *wpa_supplicant.conf*. In the file write ssid="NETWORK NAME" on one line and psk="PASSWORD" on the following line with the appropriate values
2. Run roscd pidrone
3. Run the python script *networking/Connect_to_user_wifi.py*

After completing these steps the drone's WiFi network will be disabled and we have to connect to the same home network as the drone to again SSH into the drone. These steps have to be run each time we restart the drone. We only needed to connect to the home network when we needed to update the packages

or download something onto the drone from the web. There is no other advantage of connecting to the home network than to get internet access on both the workstation and the robot. It should also be noted that we often found it useful to connect the Raspberry pi to a display using a HDMI cable to edit the files on it, rather than solely relying on SSH'ing.

1.2.5 Testing the flight of the Quadcopter using the web interface

Once we were connected to the drone via WiFi, we were ready to test the flight capabilities of the drone using the web interface already included in the package. This was done by following the operation manual [4]. But, when we ran the shell script we were not able to get the drone running and we were not able to see the video stream. We fixed this by updating Ubuntu on the Pi. We did this by running the following scripts in the terminal:

1. *sudo apt-get update*
2. *sudo apt-get upgrade*

Then we updated the *pidrone_pkg* which was already cloned on the image which was provided. We did this by pulling the changes of the *pidrone_pkg* from the GitHub repository [8] of the course from Brown University.

To fly the drone we need to open the web page *index.html* in the *pidrone_pkg-master/web* folder. The web interface can be seen in Figures 3 and 4. If the connection between the drone and the webpage is successful it shows a message *Connected* in green. Once it's connected we can arm the drone by clicking the *;* button on the keyboard and to disarm the drone click *space bar* to prevent any accidents. Always be ready to disarm the drone as we never know what happens when we take off and perform other operation. We faced situations where the drone rocketed off to the roof and drifted off and crashed on the walls. We always had a hand on the disarm button while flying the drone. Additionally, we tied the drone with a wire so that it doesn't fly off uncontrollably. We observed that sometimes there is a delay in the communication between the drone and the workstation. We didn't want to risk the drone getting out of our control, so we always armed and disarmed the drone before actually taking off. After all these checks, to take off the drone we press *w*. If all goes well the drone is supposed to hover at 0.3m from the floor (although it didn't for us). The rest of the actions with their controls are visible by scrolling down on the web page.

While playing with the drone we observed that the camera feed was not visible on the web page. After playing a lot with the webpage we observed that the feed from the camera is available only when we are running the drone in *position mode*. On the right side of the web interface under the title "Controls" is a button that allows you to put it in *position mode*.

When we tried to take-off the drone from the floor, we saw that the drone was not able to lift up from floor and was only drifting sideways on the floor. We initially thought this may be due to some problem with the flight controller and recalibrated the ESC's. This didn't fix it, so we guessed it may be some issue that will be fixed by tuning the altitude sensor. This too ended without any success. With guidance from Prof. Wong, we figured out that the PID controller for the Throttle was initialized with wrong values. We made this fix in the *pid_class.py* file. We changed the value of *midpoint* from 1300 to 1400 in the initialization of *throttle* and *throttle_low*. With this change we were able to get the quadcopter taking off. During take-off we observed that the drone was not hovering at 0.3m but kept on climbing. It should have got disarmed automatically at 0.6m, but still kept on lifting fast. We guessed that this may be due to the high speed with which the drone was taking off. We confirmed this by checking whether the drone disarms automatically at 0.6m, when we manually lift the drone with the propellers removed. This was fixed by tuning the PID controllers.

1.2.6 Tuning of the Planar, Altitude and Position Sensor

For this section we followed the Duckiesky Learning Material [9]. We followed the instructions mentioned in the learning material to tune the PID sensors. Note that the direction say that the drone should be flying over a "highly textured planar surface." This may be important. We were having significant drift when it was not over a textured surface which affected our ability to be able to tune it.

The planar tuning step was straight-forward and although we never got it quite perfect, we got decent results as can be seen in the Results section. However, we had many issues with the altitude tuning. We

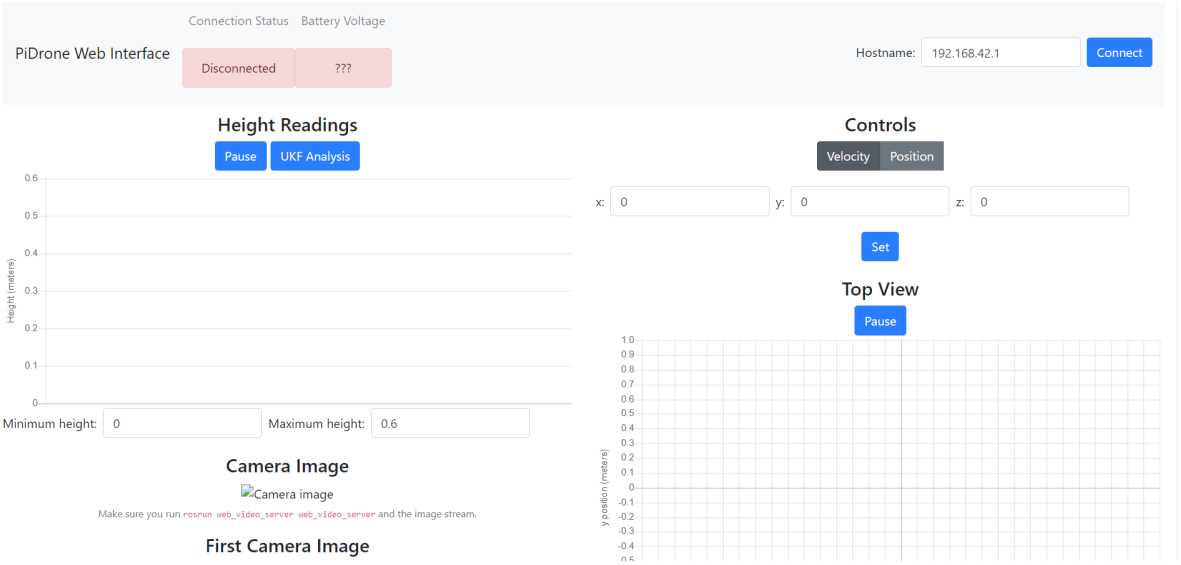
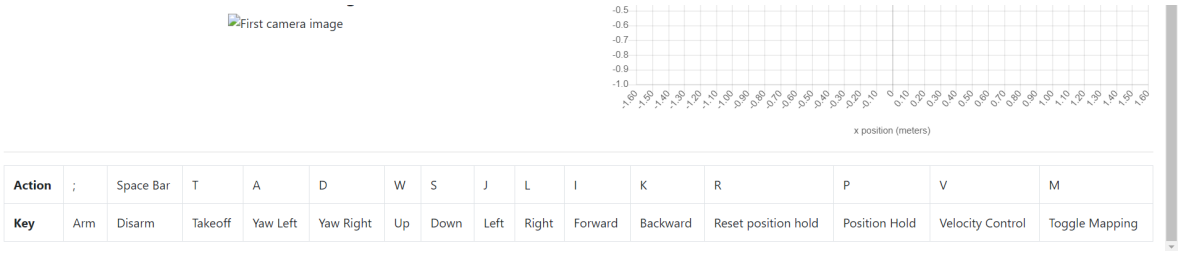


Figure 3: The web interface used to control the drone



Action	;	Space Bar	T	A	D	W	S	J	L	I	K	R	P	V	M
Key	Arm	Disarm	Takeoff	Yaw Left	Yaw Right	Up	Down	Left	Right	Forward	Backward	Reset position hold	Position Hold	Velocity Control	Toggle Mapping

Figure 4: The keyboard command directions for using the drone web interface

were never able to get it to maintain altitude at 0.3 m (which was the set point). It was either at a height that was too low or its flight would become uncontrolled and it would fly very high and end up crash landing because its safety feature would disarm the drone. Our progress on this subtask was cut short when the drone became damaged. If it had not broken, we think that simply more time with trial and error would result in us having good tuned values.

Before we broke the drone, we wrote our own PID class that we intended to tune for altitude control. We did this following the discretized form of a control function given by the following equation from [9]:

$$u(t) = K_p e(t_k) + K_i \sum_{i=0}^{k} e(t_i)\Delta t + K_d \frac{e(t_k) - e(t_{k-1})}{\Delta t} + K \tag{1}$$

In Equation 1, K_p is the proportional gain constant, K_i is the integral gain constant, K_d is the derivative gain constant, e is the error, and K is the control offset term.

The implementation of this equation in code is done through the use of a constructor for the PID class and two methods– step and reset. In the constructor, the p, i, and d terms are initialized to 0. These terms correspond respectively to the first term, second term, and third term in the summation given by Equation 1. The last error value is initialized to None and the k_p, k_i, and k_d gain constants are set the values passed into the constructor. The k value (for the control offset) is also initialized to a value passed into the constructor. The reset function resets the p, i, and d terms to 0 and the last error term to None. The actual PID logic exists in the step function. This function is called at each time step interval (which is determined by the *student_pid_controller* and *pid_controller* scripts). The step function takes in the error for that particular time step and the duration of how long that time step was. From there you can calculate the appropriate p, i, and d terms using the equation and you can set the last error term to be the current error after this is done. The actual control command returned by the function is the sum of the p, i, d, and k terms. Note that this control value is adjusted so that it is always in the range [1100, 1900].

In the portion of the Duckiesky manual for tuning the student PID class, it recommends starting with a value of 1250 for K. It is important to note that this value is for the CONTROL OFFSET, not any of the gain constants such as K_p. We mistakenly thought this was for K_p and the drone took off extremely fast and uncontrolled before crashing.

We tried tuning our throttle PID controller following the directions in the manual. We were almost able to complete the first step of this, which was to get the drone to oscillate by changing K_p and having the other gains set to 0. However, it was not consistently oscillating and its set point was too low resulting in it hitting the floor multiple times while oscillating. Looking back, we probably should have set the control offset, K, higher and tried with that. Before tuning, it had much worse performance than the PID controller in the pidrone package so we decided to return to this tuning at a later point and first focus on tuning the pidrone package's PID. However, as mentioned above, we damaged the drone before being able to complete this step.

1.2.7 Robot Motion

This was one of the most interesting section as we got to see the drone working autonomously. In this section, we made the drone hover and land on its own, without any inputs from the webpage. However, its landing was always a crash landing (achieved by cutting the motor power). We later intended to try getting the drone to land gracefully and getting it to fly in a square trajectory, but it become too damaged to attempt this.

After getting the drone to stay stable in a position without any drift using the web interface, we experimented on moving the drone autonomously. We wanted to arm the drone, take-off, and then hover at 0.3 m for a short period of time. In our initial attempt at arming the drone we published the command "ARMED" to the */pidrone/desired/mode* topic. However, this did not arm the drone. We could not for the longest time figure out why this was the case. We troubleshooted by recording rosbags and analyzing the message on the */pidrone/desired/mode* topic. This showed that the message was in fact being published, but by looking at the contents of */pidrone/mode* we saw that the mode was never being changed. After several hours of debugging with no success, we decided to further dive into the pidrone package and try to find the code for the web interface since arming it was successful with that interface. Once we found this file, we noticed that

for arming, flying, and disarming the drone, either a pose or twist message had to be sent over the topics */pidrone/desired/pose* and */pidrone/desired/twist*, respectively, IN ADDITION to sending a desired mode message. When we implemented this in our solution we were now able to successfully arm, take off, and land. The corresponding desired mode messages for each of these actions was a Mode message with the mode field either being "ARMED", "FLYING", or "DISARMED." To get the drone to hover after taking off, we simply had to enable position control by publishing a True Bool message to the topic */pidrone/position_control* and publish a pose message with position and orientation set to zero. Then to land we would disarm the drone. Using this methodology, we were able to take off, hover, and land.

However, landing by simply disarming the drone led to it crashing to the ground. We were worried (rightfully so) that this would damage the drone. So our next step was to try and get the drone to land gracefully. To do this we tried to publish a pose with a small negative value for the z position and rest all zeros. We used *-0.5* for this. Then, when the drone met the new set point in the z axis, it was supposed to publish a pose with a negative z value again and repeat this until it was only slightly off the floor, at which point it would disarm. The code to achieve this can be seen in the file hover.py in the zip file we included.

Unfortunately, this approach did not work. We are not entirely sure why. Our best guess is that this was due to the fact that we never got the previous substask working correctly (tuning the altitude control). Without having good control of altitude, the drone was not able to change the height set point used in the PID controller. As such, it was not able to make successive progress of traveling to lower altitudes until it was safe to disarm.

After working on the hover script, the quadcopter got damaged, probably due to the excessive number of times it crashed while disarming. One of the problems was a small issue with a short circuit. We attempted to fix this by enclosing the exposed wires in tape. However, it appeared that much of the exposed wire had become disconnected from the board and did not have a good electrical connection. For a more substantial fix we needed to solder it in place, but with the university closed due to the Covid-19 outbreak, we were unable to do this. Perhaps related to the short circuit and poor electrical connections, we also noticed that the flight controller was not always being properly powered and the on-board wifi component was damaged because we could not reliably connect to the drone to operate it.

Without a working drone, we were unable to further troubleshoot the altitude PID tuning to see if resolving this would result in our landing code working. This also prevented us from testing a script we wrote that would result in the drone taking off and moving in a square. In this script, we published each of the 4 actions (forward, right, left, and back) for an equal time interval, so the drone would reach the starting point again. For performing these actions we had to publish the desired changes in the desired twist */pidrone/desired/twist* topic. To move left we published twist with the x value of linear as a small negative value and the rest all zeros. To move right we published twist with the x value of linear as a small positive value and the rest all zeros. To move the drone forward and backward we published twist with the y value of linear as a small positive and negative value, respectively, and the rest all zeros.

1.3 Results

Most of the times, the drone never flew as expected. There were a lot of times when it just shot up towards the roof and fell down crashing. For the most part though, we were able to resolve this by modifying the code in the pidrone package.

Here we have show our results of a test flight, which we had recorded data from in a rosbag file. In this example, the drone took off, hovered, and then disarmed. The results (in particular the altitude control) were not great but this shows part of the progress that we were making. This flight was controlled using the web interface when this was recorded. We have got better results in the later part of the project but those haven't been documented.

From the flight it was observed that the drone had good latitude control but poor altitude controls. Looking at the graphs in Figures 6 and 7 it can be seen that the z velocities were fluctuating whereas the x and y velocity were almost constant. The altitude control of the flight was horrible as is visible in the graph in Figure 5. The altitude kept oscillating about the 0.2 m mark. This is lower than the 0.3 m set point which is the height we have set the drone to hover on take-off in the PID controller class. This is probably because we had not successfully completed altitude control tuning. In the figures referenced above, each data point in the graph is approximately 0.0015s apart. The flight was very short as can be observed in the graphs.

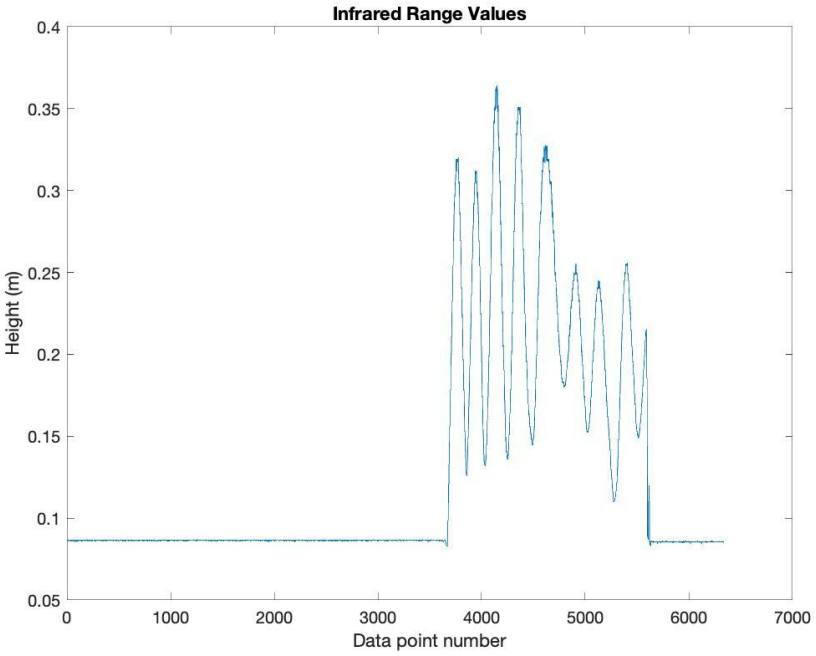


Figure 5: Infrared readings from a successful hover

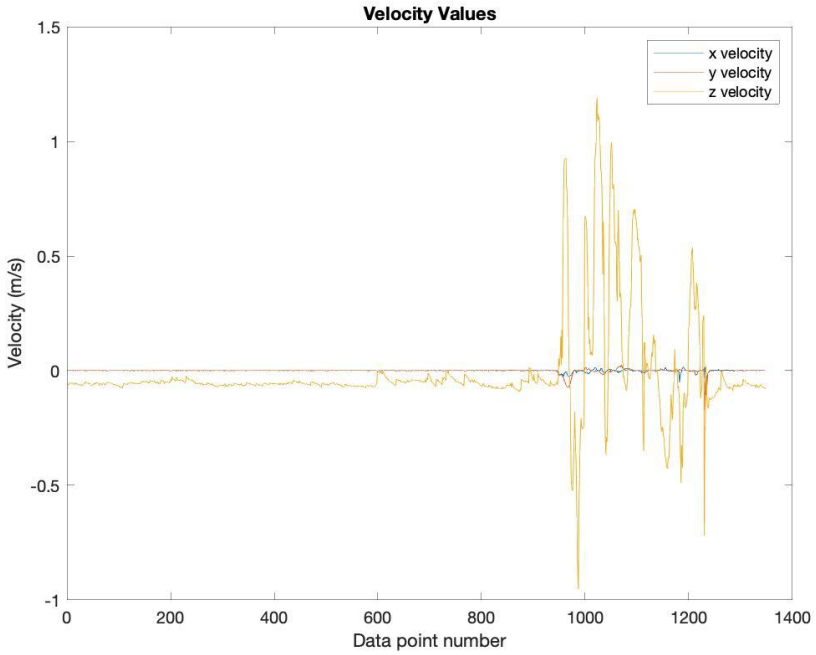


Figure 6: x, y, and z velocity values from a successful hover

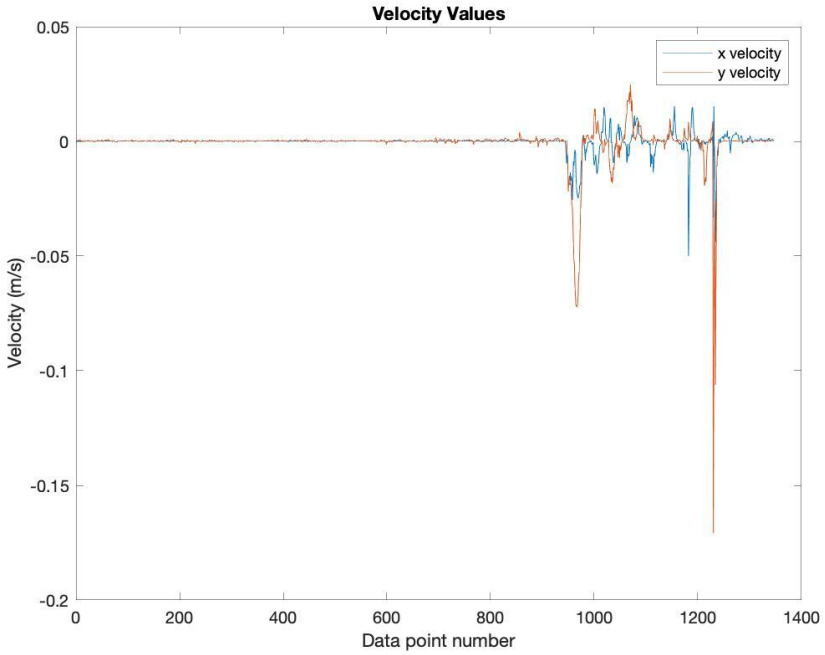


Figure 7: x and y velocity values from a successful hover

1.4 Things We Wish We Knew Ahead of Time

There were a lot of things that if we knew ahead of time would have saved us a lot of time and effort. When ordering the parts we had no idea that the Raspberry Pi 3 model B+ would cause so much issues. Also, we had no idea that the link provided in the operation manual [4] was the wrong model of Pi. We also wished that the password (*bigbubba*) for everything in the Ubuntu image, was provided in the operation manual. We ended up trying random passwords with no success and then we found a draft version of the manual in which the password was provided. When we were working on making the drone hover and fly in a square trajectory, we wanted to refer to the code that was used to control the drone from the web interface. We ended up opening each file in the pidrone package searching for the script. It would have been very easy if the location of the code was known to use beforehand. For future users, this file is *web/js/main.js*.

We never knew that a robot had so many things that could go wrong, and it did go wrong most of the times. Professor Wong used to say during his lectures that working with robots are very difficult, which we obviously took lightly until we faced the harsh truth.

1.5 Discussion

Overall, the experience of building our own robot and working on it was exciting, especially during the beginning before we moved on to the software part. We were pretty much following the schedule we had made for working on the drone. There were some frustrating times when we had no clue about how to proceed, but with timely help by Prof. Wong we were able to overcome these. We felt very good when we got the sensors working and the drone flying. There were a lot of things we learned while working on this project like flight dynamics, using a flight controller, and ROS. The main thing we took away from this project is the experience of working on hardware. It was very fulfilling to move away from just writing code or working on simulation environments for our project.

What was not at all expected during the course of the project was the lock-down of the University due to Covid-19. We could not make use of the robotics lab for working on the drone and we could not meet and work on the drone because of the self distancing. We had to meet online on Teams and discuss the progress that had been made.

Robotics was not what we expected, we started with high expectations of what we could accomplish, but in the end, it showed us that whatever can go wrong, will go wrong.

1.6 Future Work

We had some subtasks which we were unable to complete because of damage to the drone. We would like to work on them further in the future and make the drone truly autonomous. Another thing to work on would be true motion planning with the drone because as of now, the only thing the drone could autonomously do is hover. It would be cool to have the drone creating its own trajectory and flying.

Another idea we had during the beginning of the project was to have the drone fly through hoops. We had dropped it because we were worried about the amount of time it would take program the drone to detect the hoop and fly through it. We worried this would be too much in addition to simply learning ROS and how to control the drone autonomously. We would have required some extra sensors for this and would have had to make the drone fly very slowly as the processor may not be able to keep up with the amount of data. We would surely like to work more on this in the future.

In the second part of the project, we worked on a perception project, which can be read below in Section 2. Here we worked on tracking a rice box in the video and determining its position and pose. What could be done further would be to integrate that script with the drone to give it perception capabilities. This would be challenging as the error would be higher because of the vibration of the drone.

1.7 Resources

The below list has the resources we used broken down by topic. We relied mostly on the sources listed under the topics "Building the drone" and "Programming the drone." One of these sources, [8], is the Duckie Drone Github repo which we built upon as an existing code base. It would have been extremely hard to make any progress without many of the files in this repo.

- Using ROS: [1], [2]
- Building the drone: [4], [6], [5]
- Programming the drone: [9], [8]
- Drone simulation: [3]

2 Perception Project

2.1 Introduction

Our initial goal was to recognize a specific object in a picture and determine its distance from the camera and angle of rotation. Most of the images we did this in had the box in isolation. For this problem, we decided to use a red rice box and always have it in the x-y plane so that the rotation calculated would be a rotation about the z axis. Once we finished this task, we decided to make the problem more challenging by building upon the work we had already done. Our final project goals were as follows:

- Track the red rice box in a video file where there may be occlusion and other objects in the background
- Determine the pose of the box (this being the x, y, z coordinates of the AR tag on the box and a quaternion representing the orientation of the box relative to the camera)
- Compare our calculated pose to ground truth (which would be determined using the ROS package AR track alvar) and compute errors

To accomplish this goal, we came up with several subtasks that were needed to address the problem. We were able to address all of the substasks throughout the semester. The subtasks were as follows:

1. *Create a library of sample photos to use for testing the development of our algorithm and a ROS bag file with video information to test our final solution*

Take pictures of the rice box laying flat on the floor at different angle increments of 45°

Take pictures of the box in a scene at different distances away from the camera

Record a ROS bag file of an image stream with the box being moved at different positions and orientations

2. *Read through OpenCV python tutorials to get a better understanding of the library functions at our disposal*

3. *Be able to extract contours from an image*

   Input: the image to extract contours from and HSV values that when used on the image results in a mask with roughly the region of interest in white

   Output: a list of contours from the image

   Algorithms used for step: cv2.cvtColor (convert bgr image to hsv values), cv2.inRange (make a mask image of lower and upper bounded hsv values), cv2.threshold (thresholds a mask to make a binary image), cv2.findContours (finds contours in a binary image)

4. *Determine which contour (out of a list of possibilities), if any, is most likely to belong to our object of interest*

   Input: a list of possible contours and a contour that has the right dimensions of the box to be tracked

   Output: a contour that is most likely the box to be tracked or None if one does not exist

   Algorithms used for step: cv2.convexHull and cv2.minAreaRect (make a best fit rectangle of the contour), cv2.matchShapes (get a number that represents how close of a match a contour is to the desired contour)

5. *Perform feature matching*

   Inputs: Two grayscale images to do feature matching (an image that the box needs to be tracked in and a reference image that is only of the box)

   Outputs: Two lists– the first list contains the (x, y) values of a pixels in the image that represent a certain feature and the second list has (x, y) values for pixels in the second image that correspond to the same features

   Algorithms to be used in step: Either ORB, SURF, or SIFT

6. *Use the results of feature matching to calculate a homography matrix*

   Inputs: two lists of corresponding points for matched features between the two images

   Outputs: a homography matrix

   Algorithms to be used in step: the equation/process used in [10]

7. *Use the homography matrix to determine a bounding box for the object of interest*

   Inputs: the homography matrix and the pixel locations of the four corner points of the box in the template image (the known image of the box with 0 rotation)

   Outputs: a mapped set of four points corresponding to the four corners of the box

   Algorithms to be used in step: cv2.perspectiveTransform (perform a perspective transformation to points give a homography matrix) or our implementation of that function

8. *Decompose the homography matrix into a rotation matrix and translation vector*

   Inputs: homography matrix

   Outputs: 3x3 rotation matrix and 3x1 translation vector

   Algorithms to be used in step: cv2.solvePnP (finds an object pose from point correspondences) or our implementation of that function

9. *Get the quaternion represented by the rotation matrix that was extracted*

   Inputs: a 3x3 rotation matrix

   Outputs: a corresponding quaternion

   Algorithms to be used in step: scipy.spatial.transform.Rotation.as_quat()

10. *Calculate the position of the box in the frame*

    Inputs: the homography matrix, the bounding box for the rice box, the coordinates of the center of the ar tag in the known image, dimensions of the image

    Outputs: x, y, z coordinates of position

    Algorithms to be used in step: cv2.perspectiveTransform (to get the projected points of the AR tag in a known image to the frame in question) or our implementation of it, trigonometric functions to estimate position relative to pixel coordinates

11. *Project a set of axis for the pose of the box on the frame as well as the bounding box*

    Inputs: bounding box, axis points, and matrix that has the translation and rotation information for the pose

    Outputs: a frame with axis and bounding box drawn on

    Algorithms to be used in step: cv2.drawContours (to draw the bounding box), cv2.projectPoints (to project axis points in correct orientation onto frame) or our implementation of it

12. *Use the AR track alvar ROS package to calculate ground truth for a series of selected images*

    Inputs: video with box being published over an Image topic, camera calibration parameters being published over a CameraInfo topic

    Outputs: Pose message

    Algorithms to be used in step: the AR track alvar pose estimation algorithm

13. *Use our algorithm to calculate pose for a series of images*

    Inputs: an array of images containing the rice box

    Outputs: the pose (position and orientation) for the box in each image

    Algorithms to be used in step: algorithm developed by us using loosely the above steps

14. *Compare our results to ground truth*

    Inputs: pose for frame calculated by AR track alvar and pose for corresponding frames calculated by our algorithm

    Outputs: distance between position and orientation for each frame

    Algorithms to be used in step: Matlab's norm function and the quaternion distance equation described in [11]

Notes:

- In all the above steps (and throughout the report) cv2 refers to the OpenCV Python library
- All the development code for this project can be found at the following Github Repo: https://github.com/dithier/CS5335_ROS where the final code products exsiting in the src/scripts folder

2.2 Approach

2.2.1 Creating a Data Set

Carter created the data set. The initial data set was a series of pictures taken on a phone. The majority of the images were of the rice box flat on the floor, in isolation, at different angles of rotation. There were a few photos taken of the box being in a scene.

Once our algorithm was working decently at detecting angle of rotation, we decided to make it more challenging by trying to estimate the pose. To do this we needed more data. We needed to create information that had ground truth of the pose so we used the package AR track alvar to generate an AR tag that we printed out and taped to the rice box [12]. We then took a video using a computer webcam in ROS and recorded a rosbag of the Image and CameraInfo topics being published. We did this using the Video Stream OpenCV package [13]. Due to lighting issues that become apparent during algorithm development, we did have to take a second video in order to have usable data.

We initially had issues getting the webcam to communicate with our virtual machines. We first started using an Ubuntu virtual machine from VirtualBox, but could not get the machine to recognize the host computer's webcam. There was much trial and error and we were not able to resolve the issue. As a workaround, we installed VMware and found that the webcam worked with it without needing any additional configuration.

We anticipated that testing our algorithm on the video file would be difficult because we wanted to be able to test small changes on one frame at a time. As a result, we wrote a ROS node that allowed us to view the image from the Image topic of the rosbag and save select frames to a file. We saved 16 frames that had different positions and orientations to test our algorithm on.

2.2.2 Learning OpenCV

We both devoted approximately one week to learning the basics of the OpenCV Python library. This was done by following the official online tutorials [14]. The relevant sections of the tutorials that we read were GUI Features in OpenCV, Image Processing in OpenCV, Feature Detection and Description, and Camera Calibration and 3D Reconstruction.

In order to follow along with the tutorials we actually had to install OpenCV. The library comes with ROS so it was already installed on our virtual machines, but we also wanted to be able to run it on our host computers. We found that the easiest way to install it on our host computer was through pip [15]. [15] is for OpenCV 4 which is a different OpenCV version than the one installed with ROS Kinetic, but we found the differences between the two versions to be minor.

2.2.3 Learning ROS

As described in Section 1.2.1 we spent approximately 2 to 3 weeks getting familiar with ROS. Please see that section for further details.

2.2.4 Extracting Contours From an Image

Both of us worked on extracting contours from an image. Extracting contours from a thresholded image is relatively straight-forward when using OpenCV– you simply call the function cv2.findContours(). The challenge we experienced in this step was actually obtaining a thresholded image that contained our region of interest (the rice box). The box is a bright red color so using this fact and from seeing examples in the OpenCV tutorials we figured that HSV filtering would be a good option.

To create a mask with the appropriate HSV values using OpenCV's function cv2.inRange(), we needed to determine the lower and upper bounds for the HSV values. We started off by following the suggestion in [16] for finding the bounds. We used OpenCV to get the hsv value for a BGR (0, 0, 255) color and used a lower bound of [H - 10, 100, 100] for the lower bound and [H + 10, 255, 255] for the upper bound. The results were very poor; the box was not well isolated. We spent a lot of time playing with these values by hand with little success which made us realize that we needed a more systematic approach to finding the bounds.

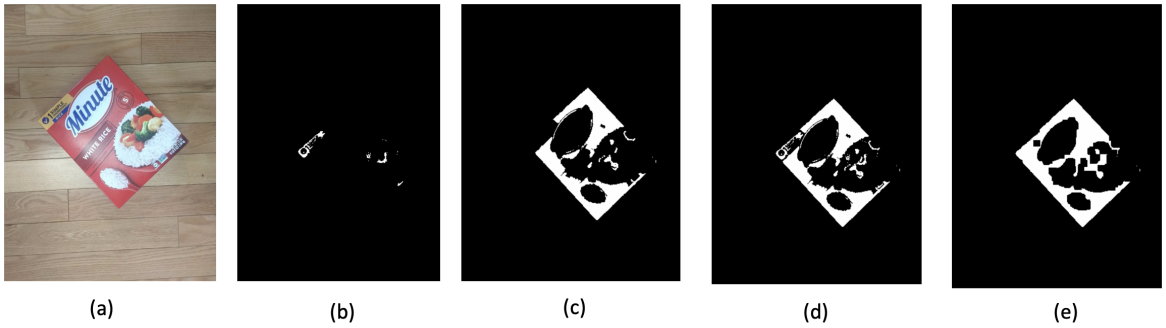


Figure 8: (a) Original image (b) mask of orange in the box (c) mask of red in the box (d) combined orange and red mask (e) mask after closure morphological operation

We decided to modify the tutorial in [17] which resulted in us having a trackbar with an image. There were sliders for H low, H high, S low, S high, V low, V high, and an on off switch that when turned on showed the thresholded image and when off showed the original image. We then used this trackbar implementation to find the HSV boundaries. There were still some difficulties with this approach, however. For instance if you loaded on image, it was often easy to find the ideal values, but then when you applied those values to a different image they were not as good. To workaround this we found the values that worked well on one image. Then we would find at least two other images in our data set that the values were not ideal for. We created a new image containing those three images side by side. Then we tuned the HSV values using the trackbar on that combined image. This allowed us to see the affects on more than one image at a time and find middle ground in getting good values.

There were a few more things we experimented with regarding HSV values. While we eventually found values that worked well for the majority of the box, we found that the bottom right corner (which was a more of a crimson color than a red color) and the top left corner (which was orange) was getting cut off. As a result we decided to find HSV values for those two colors as well and then blend the masks of those three thresholded images. After much experimentation we found that the crimson color overlapped too much with different parts of the background in images where the box was not isolated thus giving us poor results. In turn, we decided in our final product to ignore the crimson color of the box and just use a blended mask of the orange and main red part of the box.

The last thing we experimented with for creating a mask was different morphological operations. The mask from the orange part of the box was sometimes unconnected from the mask of the red part of the box. We needed to somehow blend these two parts together. We played with different operations such as erosion, dilation, closing, and opening. In our final product we used a closing operation on the combined masks.

Once we had our mask we were able to threshold it using the cv2.threshold() function and finally extract the contours using cv2.findContours(). A visualization of the outputs of the different steps to get a mask of which to find contours can be seen in Figure 8.

2.2.5 Performing Feature Matching

Carter completed this subtask. We started by following one of the feature matching tutorials from the OpenCV tutorials [18]. However, rather than using SIFT for computing keypoints and descriptors like the tutorials, we used ORB. This decision was made because SIFT does not come with the OpenCV installed with ROS.

We first tried doing brute force matching with the cv2.BFMatcher object and then sorting the matches in order of their distance. Then we chose the matches that were a threshold distance away. This threshold was chosen by experimentation. We tried multiple images and determined which distances found corresponded to "good" fits and which ones corresponded to "bad fits." We also took into consideration how many matches would get filtered out. For instance, some thresholds resulted in good matches, but there were very few matches total (less than 10).

However, we still found that too many bad matches were not getting filtered out. Examples of some bad

Figure 9: Bad matches made between features in the known image of the rice box and the image the box is to be located in

matches can be seen in Figure 9. On the left shows the rice box in isolation where features on the spoon the the bottom part of Minute label are matched to wrong parts of the box in the other image. The image on the right shows how much worse this problem becomes when the box is not in isolation. Out of all of the matches drawn, none of them are correct. Simply playing with the distance threshold could not resolve this issue.

Our next attempt was to use a different algorithm for feature matching. Following [19], we performed K-Nearest Neighbors matching and determined which matches were good by applying a ratio test between the distances of each pair of matched features. However, the results did not seem remarkably better than the previous attempt; there were still bad matches.

One thing we noticed in both these methods is that there were very few matches made in general. As can be seen in Figure 9, there were only a handful of matches. Our first suspicion was that this was occurring because there were not enough features being computed. To change this we changed the nfeatures parameter in the OpenCV ORB constructor. Figure 10(a) shows the initial amount of features detected on the box and (b) shows how many more features were computed when we increased nfeatures to 100000 (the default is 500). This increased the number of matches, but not by much.

We noticed that by increasing the number of features, some of the features detected were in parts of the image that we wanted to ignore (such as walls, chandeliers, and cabinets). To prevent this, and thus lower the chance of having more poor matches, we passed in a mask of the largest contour found in the previous step (Section 2.2.4) to try and restrict the area of the image that keypoints were found.

We next tried changing other parameters in the ORB constructor. We used [20] as a resource to what parameters could be changed as the OpenCV tutorials did not have this information. Figure 11(a) shows feature matching when using the default ORB constructor parameters. The number of matches slightly increased when we changed the number of features, changed the feature scoring type from Harris Score to Fast Score, and changed the fastThreshold parameter from 20 to 10. Figure 11(b) shows the results of these changes. We were still not satisfied because of the low number of matches. In our next iteration we also changed the edgeThreshold (which is the size of the border where features are not detected) and the patchSize (which is the size of the patch used by the BRIEF descriptor). We decreased both the edgeThreshold and patchSize (which are by default 31) to 20. The results can be seen in Figure 11(c). As can see, this dramatically increased the number of matches that were found which resulted in us keeping these settings.

The results from Figure 11 were from matches generated by brute force matching. When we tried KNN matching, we got much fewer matches because it seemed that the ratio test was too stringent. As a result in our final product we used the brute force matching method. Note that although we got many more matches, we did not successfully find away to get more accurate matches while still retaining a high number of matches. This is an area of future work and is described more in the Future Work section.

2.2.6 Selecting the Best Contour

Both of us worked on selecting the best contour. In our first attempt, we sorted the provided contours by area from largest to smallest and chose the contour with the largest area. This approach worked the majority

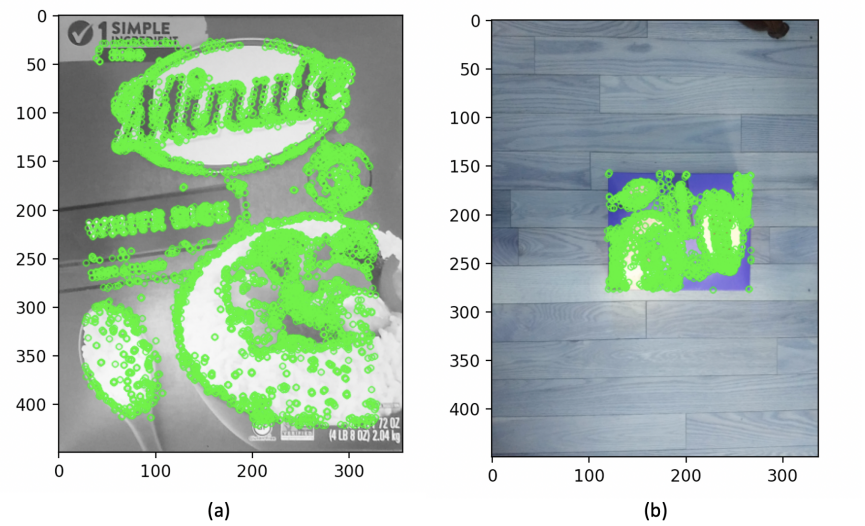


Figure 10: (a) Doing feature extraction with the default number of features (b) Feature extraction used with an increased number of features

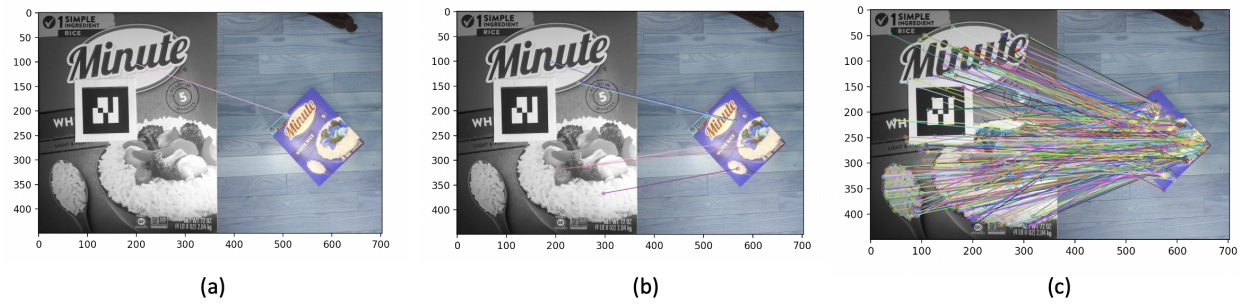


Figure 11: (a) Feature matching using the default settings of ORB during object creation (b) feature matching with changed settings for nfeatures, scoreType, and fastThreshold values (c) feature matching with changed settings for nfeatures, scoreType, fastThreshold, edgeThreshold, and patchSize

(a) (b)

Figure 12: (a) Box with a contour overlaid in green (b) box with a contour overlaid in green and the best fit rectangle of the convex hull of that contour drawn in yellow

of the time, but it would also result in contours getting chosen that were not similar in shape at all to the box. This was undesirable because we wanted to be able to rule out objects in the image that were not the rice box.

In our next attempt, we decided to try to include information we obtained from feature matching and to take into account what the actual shape of the box was. We again sorted the contours from largest area to smallest. We started with the largest contour and this time calculated the number of keypoints found from feature matching (between a known image of the box and the frame in question) that resided inside the contour. We also decided to do shape matching between the contour we were examining and a desired contour. The desired contour was created by us to be a perfect contour of the box (a contour that had the correct shape and aspect ratio). We compared the similarity of these two contours using the OpenCV function cv2.matchShapes(). In order for the results to be most accurate for the matching of shapes, we chose to match the best fit rectangle of the convex hull of the contour in question to the desired contour. An example of a contour and the best fit rectangle of the convex hull used for matching shapes can be seen in Figure 12. If the contour in question had more than a certain threshold of keypoints residing inside it and met a certain threshold for similarities in shape to the desired contour then we considered it a valid contour and returned that contour. If it did not, we would then begin the same process for the next largest contour. We would repeat this process a maximum of ten times. If none of those contours met the requirements we returned None indicating there were no valid contours.

This approach worked extremely well on all of the test images from the cell phone that we used. We had rather strict requirements for the keypoint and shape matching thresholds and this led to very reliable results. Note, however, that the test images from the cellphone were mainly of the box in isolation. Only a handful of the images had the box in a scene. As a result, the thresholds were too strict when applied to images with the box in the scene. After relaxing the thresholds based on tuning with multiple images from the bag file, we were able to get good performance. Because the constraints are less stringent there is a higher probability that false positives could occur than previously. This may be an area of future work.

2.2.7 Calculating a Homography Matrix

Carter completed this subtask. Initially we found the homography matrix using the OpenCV function cv2.findHomography(). This worked extremely well right away. As the project progressed, we decided to

re-implement the function ourselves.

Our first step in re-implementing the function was to search online for examples of how to do this. We came across many helpful resources for this (see Section 2.6.4). We ended up using the following equations from [21]:

$$A \cdot x = b \tag{2}$$

$$A = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -u_1 \cdot x_1 & -u_1 \cdot y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_1 \cdot x_1 & -v_1 \cdot y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -u_2 \cdot x_2 & -u_2 \cdot y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -v_2 \cdot x_2 & -v_2 \cdot y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -u_3 \cdot x_3 & -u_3 \cdot y_3 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_3 \cdot x_3 & -v_3 \cdot y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -u_4 \cdot x_4 & -u_4 \cdot y_4 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_4 \cdot x_4 & -v_4 \cdot y_4 \\ & & & & & \vdots & & \end{bmatrix} \in \mathbb{R}^{2N \times 8}, \; x = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} \in \mathbb{R}^8, \; b = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \\ \vdots \end{bmatrix} \in \mathbb{R}^{2N} \tag{3}$$

where $[u, v]^T$ are the pixel coordinates in the reference image and $[x, y]^T$ are the corresponding coordinates in the image in question and the homography matrix contains the elements of x:

$$H = \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & 1 \end{bmatrix} \tag{4}$$

This equation requires a minimum of four corresponding points to work. In our first attempt we decided to use only four points. For each image we received a long list of corresponding points from the feature matching step described in Section 2.2.5 so we needed to determine a way to pick just four of them. We decided that the points to be picked should be the first four points that had the closest match distance (as determined by feature matching), lied inside the contour returned from the process in Section 2.2.6, and were not colinear. The results of this process can be seen in Figure 13(a). As can be seen, the results were horrible. Note that on other test images the process worked slightly better, but none of them worked great.

Our next attempt was to use more points returned from feature matching. We started with the top 25 points of the closest match distance that were inside the most likely contour and when this improved results we changed to using all points returned from feature matching. This dramatically improved the results, but the orientation was still wrong. An example of a bounding box determined from a calculated homography using all points can be seen in Figure 14. As can be seen in the figure, the dimensions of the bounding box are approximately correct, but the bounding box is rotated out of the plane incorrectly. For some images, such as Figure 13(b), the bounding box and pose were still completely wrong.

After researching online, it appeared that our problem was that having bad matches between features in our images could greatly impact results. We needed a solution that would be more robust to outliers. Online resources suggested that the RANSAC algorithm would work well for this. Psuedocode of our implementation of the RANSAC algorithm can be seen in Algorithm 1.

Figure 13: (a) Pose decomposed from homography obtained from using the four matched features with the closest distances (b) pose decomposed from homography obtained from using all matched features (c) pose decomposed from homography obtained using RANSAC

Figure 14: Bounding box determined from homography calculated using all feature matched points

Algorithm 1: RANSAC for Homography

Input : Two list of points corresponding to the same features
Result: A homography matrix calculated without outliers
p ⟵ 4 ; `// number of points to be used in model`
d ⟵ distance threshold;
N ⟵ number of iterations to run algorithm;
T ⟵ number of nearby points required to assert a model fits well;
for $i \leftarrow 1$ **to** N **do**
  draw a random sample, S, of p points from the data ;
  if *points in S are colinear* **then**
    continue;
  end
  find homography matrix for points in S;
  for *each point from the input not in S* **do**
    calculate inliers and number of inliers ; `// points less than` d `distance from model`
  end
  if *number of inliers is higher than best previous ones* **then**
    store the inliers
  end
end
if *best number of inliers* $>= T$ **then**
  return homography calculated using all of the inliers found from the best model
else
  no homography found;
end

In this algorithm, the distance metric used for determining whether a point was an inlier was the sum of square differences:

$$\sum_i d(p'_i, H * p_i)^2 \tag{5}$$

where d is the distance between the two values, p_i is the point in the known image, H is the homography matrix, and p'_i is the corresponding point in the image you are calculating the homography for.

Applying this algorithm to the homography calculated worked extremely well; results can be seen in Figure 13(c). However, getting such results (and making the results be consistent despite the random nature of the algorithm) required a great amount of time tuning values such as the distance threshold, number or nearby points required to assert a good fit, and parameters for feature matching discussed in Section 2.2.5. A sample of different parameters tried in this step can be seen in the RANSAC_tuning.csv file in our submitted work. There were a few photos in our data set that OpenCV's homography function worked better on than ours, but for the most part ours worked just as well.

2.2.8 Determining a Bounding Box

Vinay worked on this portion of the code. In our initial data set, which consisted of images taken on a phone and mostly had the box in isolation, we computed the bounding box as the best fit rectangle of the convex hull of the most likely contour (where the contour was selected using the process explained in 2.2.6). This worked extremely well, as can be seen by the yellow bounding box in Figure 12(b).

However, when we started testing this process on images from the bag file where there were lots of other objects in the background, this method became extremely unreliable. The yellow box in Figure 15 shows the resulting bounding box using the method. This problem occurs because we were not able to tune the hsv values well enough to completely isolate the red box in all of the frames from the bag file.

Our solution was to use the homography matrix we computed in the previous step. We found the pixel coordinates of the four corners of the box in the reference image and used the homography matrix to determine the pixel locations of these corners in the frame in question. At first we used the OpenCV function cv2.PerspectiveTransform() to accomplish this, but later Carter implemented this function himself

Figure 15: Frame with the convex hull of the most likely contour in green and the best fit rectangle of this hull in yellow

for the final product. This was done by using the following equation:

$$\begin{bmatrix} \lambda x' \\ \lambda y' \\ \lambda \end{bmatrix} = H * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6}$$

where (x, y) is the pixel coordinate of a corner in the reference image and (x', y') is the corresponding coordinate of the pixel in the image in question.

This solution that used the homography matrix consistently worked better than our first attempt. In terms of overall performance, it worked pretty well as can be seen in the Results section. The main issues in performance occurs when there is a poor homography estimate.

2.2.9 Computing Orientation

Carter worked on this subtask of the project. We first extracted rotation and translation for the image in question using the OpenCV function cv2.solvePnP by following the tutorial from [22]. This process was straight-forward and immediately gave great results. However, we wanted to further challenge ourselves and rely on OpenCV less so we decided to re-implement this function ourselves.

Most of the completion of this subtask consisted of doing research. From the function signature of solvePnP we knew that the camera distortion coefficients, camera calibration matrix, and known correspondences between points in the reference image and image in question were needed. The last observation gave us the idea that somehow the homography matrix was involved or could at least be of use. This led to several hours of researching how to decompose a homography matrix into rotation and translation. Most of the articles and websites we came across were dead ends; they either talked about actually calculating the homography matrix (not decomposing it) or they were tutorials that utilized already implemented functions from libraries such as OpenCV. Eventually, we came across [23] which demonstrated an approximate way to decompose the matrix. We implemented this and it worked well without any modification.

2.2.10 Get the Quaternion From Rotation Matrix and Make It Have Same Axis Convention as Ground Truth

Carter worked on this section of the project. Thanks to the Rotation class from the Scipy spatial transform library, getting a quaternion from a rotation matrix is very straight-forward [24]. You simply create a Rotation object from the rotation matrix and call the function as_quat().

When we first tried this and were comparing our results to ground truth, we noticed the quaternions were very different even for images that the axis looked to be appropriately positioned on. Further research revealed that this was because AR track alvar, the package used to determine ground truth, used a different axis convention than our set up. Specifically, the AR track alvar axis could be obtained by rotating our

coordinate system 180°about the y axis. More discussion is in the Results section regarding our trial and error in converting our axis system to the convention ground truth used, as well as results that were obtained in our first failed attempt. Once we determined that Equation 19 was the correct transformation we multiplied the two matrices in that equation using the Numpy library and used Scipy to extract the quaternion.

2.2.11 Calculate Position of the Box in the Frame

Vinay wrote this portion of the code. The distance of the object from the camera can be computed with knowledge about the original size of the target and the camera optics. We use the information about the field of view of the camera.

The first step is to find the width of the frame in inches. For this we use the following equation:

$$width = \frac{(original\ target\ width\ (in) * width\ of\ frame\ (pixels))}{width\ of\ box\ (pixels)} \tag{7}$$

The original target width is the known width of the rice box we are using, width of the frame is the number of columns in the image array, and the width of the box is the width in pixels of the bounding box found.

After finding the width, we can use trigonometry to find the distance from the camera to the target. One thing to remember here is that the width we found out earlier has to be halved. We used the below equation to get the distance.

$$distance = (width/2)/\tan(\theta) \tag{8}$$

Here, θ is half the field of view of the camera. The camera we used had a field of view of 55°. A visualization of these calculations can be seen in Figure 16 which is taken from [25]. [25] was used as a reference for all calculations in this subtask.

To calculate the value of the x position, we made use of the width of the image calculated earlier. The value of x is given by the following equation.

$$x = \frac{X * width}{width\ of\ frame\ (pixels)} \tag{9}$$

where X is the pixel x coordinate of the center of the AR tag.

To calculate the value of y position, we first calculated the height of the target in the image similar to method for calculating the width. The equations are as follows:

$$height = \frac{(original\ target\ height\ (in) * height\ of\ frame\ (pixels))}{height\ of\ box\ (pixels)} \tag{10}$$

$$y = \frac{Y * height}{height\ of\ frame\ (pixels)} \tag{11}$$

where Y is the pixel y coordinate of the center of the AR tag.

The x and y axis of the image frame in OpenCV is such that (0,0) is located at the top left corner of the image and the positive x axis points to the right and the positive y axis points down. But, in the coordinate system used by AR track Alvar, the center of the image is (0,0), the positive y axis points up, and the positive x axis is towards the right. In our final solution, we have converted the positions calculated into this coordinate system and used the distance calculated from Equation 8 as the z value. We also converted all position measurements to meters to be consistent with the values returned by AR track alvar.

2.2.12 Project Axis and Bounding Box on Frame

Both of us worked on this subtask. Drawing the bounding box was very easy; it only required a call to the OpenCV function cv2.drawContours().

To draw the axis on the frame we first had to determine where the end points of the axis lie. The end point for the x axis was the coordinate [L, 0, 0] in the original frame, the y end point was [0, L, 0], and the z end point was [0, 0, L] where L denoted a length long enough to easily see the axis on the image in question.

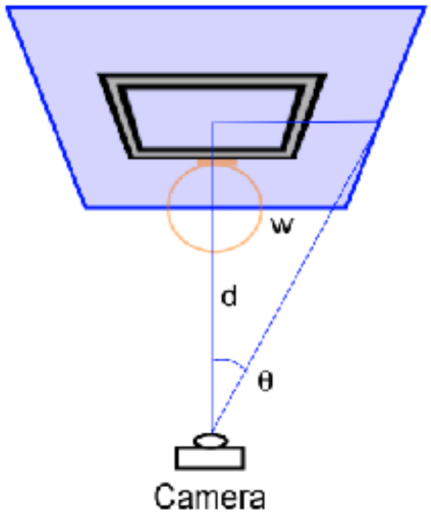


Figure 16: A visualization from [25] for calculating distance

Then these three points were projected using the rotation and translation information that was calculated by the methods discussed in Section 2.2.9. This was initially done using the function cv2.projectPoints(), but we later re-implemented this function ourselves.

In our first attempt of a project points function, we used an equation similar to Equation 12. However, this did not give good results. We later realized that this was because we were not taking into account the intrinsic camera calibration matrix. This, and further online research, led us to the following equation to project a point:

$$\begin{bmatrix} \lambda x' \\ \lambda y' \\ \lambda \end{bmatrix} = K * \begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{12}$$

where (x', y') is the projected point, K is the camera intrinsic matrix, the matrix to the right of it is a combined matrix with the rotation and translation (or camera extrinsics), and (x, y, z) is the coordinate of the point in 3D space to be projected. This equation led to the results we expected.

Once the the projected points were determined, drawing them on the image was simple thanks to inspiration from an OpenCV tutorial [22]. We simply chose an origin point (which we decided to be the top left hand corner of the box) and drew different colored lines from that point to each axis end point.

2.2.13 Get Predictions and Ground Truth For Set of Images

This subtask was completed by both of us. Once we finished our algorithm for predicting pose, we had to predict the pose for a series of images and determine the true pose for those images so we could later compare them. We extracted true pose using the ROS package AR track alvar. Getting the package to publish poses to a topic was relatively simple by following the directions in [12] and using the pr2_indiv.launch file example in [26].

The first issue we had when trying to obtain ground truth was that we were not publishing a CameraInfo message which resulted in the package not being able to track the object. We could not figure out why it was not tracking the object, but then after looking at online resources we realized it was because it needed the camera calibration information. As such, we calibrated the camera (see the camera calibration resources in Section 2.6.4) and wrote a node that published the camera info. Then, AR track alvar worked perfectly.

The second issue we had was timing issues. We initially intended on playing the rosbag that had our video data and writing a subscriber node. This subscriber node subscribed to the raw image topic and the topic with pose information from AR track alvar. In the callback for the raw image topic we had code that

ran our algorithm and then published a pose message with the results. In the callback for AR track alvar messages, it extracted the pose from the message (which was of type AlvarMarkers) and published a pose message. We did this because it was easier to work with a Pose message then an AlvarMarkers message for our data anlaysis later. Our plan was to record the two topics published by this node in a rosbag that we could later analyze in Matlab. This did not work though because the two callback functions ran at rates too slow to process all of the images. For example, out of the 1213 image messages in the rosbag, our node only published a little over 300 of AR track alvar related messages and only a handful for our solution.

Our first attempt to fix this issue was to play with the queue size of the subscribers, but this did not work. We then started playing the frequency that the rosbag messages were played at. By slowing the publishing down to a rate of 0.7 Hz, we were able to get AR track alvar to publish the same amount of messages as there were images in the rosbag. However, the callback for our pose estimation solution was still far too slow to keep up.

As a work around, we decided to process our images outside of ROS. Due to the length of time it took to process our images, we decided it would take too long to analyze all 1213 images so we decided to only analyze some of them. This meant we had to randomly select a subset of images. To do this, we wrote a node that subscribed to the raw image topic of the rosbag and stored all of the images that were published in an array. Once no more images were being published, we saved this array to a Numpy npz file. Then in a python script, we loaded the npz file with the images, randomly selected 50 of them, and saved this to a new npz file. We wrote a driver script that loaded the 50 images from the npz file, ran our algorithm to predict pose on each image, saved an image of the pose output for each image, and saved the position and orientation results to a csv file using the Pandas library.

Next we used ROS to get the ground truth. We wrote a node that loaded the npz file and published each image to the */webcam/image_raw* topic at a rate of 0.7 Hz. We also added a short delay from after the node was initialized to before it started publishing because we noticed that sometimes the first image or two were not being published. We then started recording a rosbag and started the image publisher node, the node we wrote earlier that published CameraInfo messages, the node that published a modified message of pose extracted from AR track alvar, and ran the launch file for AR track alvar.

Once we had the rosbag we now knew the ground truth for 50 images. We extracted this information from the rosbag in Matlab by reading the messages from the */Ar_Pose* topic that contained the pose. This was done using the functions rosbag(), select(), and readMessages(), and cellfunc(). We put all of the data in a matrix and then wrote the results to a csv file.

2.2.14 Comparing Our Results to Ground Truth

Carter implemented this portion of the project. Once all of the previous subtasks of the project were finished, the last step was to compare our results to those obtained from AR track alvar. This subtask was straightforward and had no complications. First we created an excel sheet that had both our results and the ground truth. Each row had data for a different frame that was analyzed. Column 1 had the image number, columns 2-8 had our position and orientation prediction, and columns 9-end had the position and orientation from AR track alvar. We loaded the data into Matlab and removed the rows from our predictions and AR track alvar's predictions where AR track alvar was unable to calculate a pose. We then calculated the error in position and orientation between our estimate and the truth and graphed the errors for each frame. Afterwards we found the average errors as well as the standard deviations. We noticed that the standard deviations were very high for the data due to some extremely inaccurate pose estimations. To get a better idea of our accuracy for the average case of our algorithm, we filtered out data from frames whose results we considered to be outliers and redid the above calculations. These results, as well as more discussion on how errors were calculated and what we considered to be an outlier, are described in the Results section.

2.3 Results

In this section we will show both our positive and negative results for pose prediction of a red rice box. To obtain these results, we recorded a bag file containing Image and CameraInfo messages pertaining to the box in various positions and orientations. The file was approximately 40 seconds long and contained 1213 messages in the topics for raw camera image and camera info. Running our algorithm takes approximately

Figure 17: (a) Pose detection from AR track alvar (b) Pose detection from our original solution

18 seconds per frame. This implies it would take approximately 6 hours to process all of the frames in the video. We thought that a smaller sample of images that would take less time to process and would still give us insight into how well our algorithm performed. As such we decided to randomly select 50 frames from the 1213 total frames to run our analysis on.

Before showing the final results, however, we wanted to show results for intermediary results that were incorrect. We were able to come up with a solution that would come to a reasonable pose estimation, but this pose estimation had different axis conventions than AR track alvar. This meant that we could not directly compare our quaternion result to the ground truth. Figure 17 shows AR track alvar's axis convention on the left and ours on the right. In both pictures the x axis is red, the y axis is green, and the z axis is blue. (Note that our z axis is projected in the negative z direction.) It becomes apparent from looking at the pictures that AR track alvar's convention could be obtained from ours by rotating our axis 180°about the y axis. To achieve this we knew we had to multiply the rotation matrix from our solution by a rotation matrix that represented a rotation about the y axis. This rotation matrix (where β is 180°) can be seen below:

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{13}$$

In our first attempt of applying this rotation matrix, we did the multiplication in the wrong order. Our equation for the new rotation matrix (which we then derived the quaternion from) was

$$R_{solution} = R_y(\beta) \times R_{init} \tag{14}$$

where R_{init} was the rotation matrix for the pose using our convention of axis, $R_y(\beta)$ is given by Equation 13, and $R_{solution}$ is the rotation matrix for the pose using AR track alvar's convention for axis.

However, when using Equation 14 we got very high errors in the orientation estimation. These errors can be seen if Figure 18. The error was calculated by finding the the angle of rotation needed to get from our predicted quaternion to the one provided by AR track alvar. This is given by the formula

$$\theta = \cos^{-1}(2(q1 \cdot q2)^2 - 1) \tag{15}$$

where $q1 \cdot q2$ is the inner product of the quaternions [11]. We noticed that the errors were very high even for specific frames where the axis from the rotation matrix before it was rotated about the y axis appeared fairly accurate. This led us to believe that we were doing something wrong in the conversion from our axis convention to AR track alvar's. To confirm this, we drew the axis computed after the conversion on frames that we knew the pose estimation was accurate. This revealed that there was in fact something wrong as

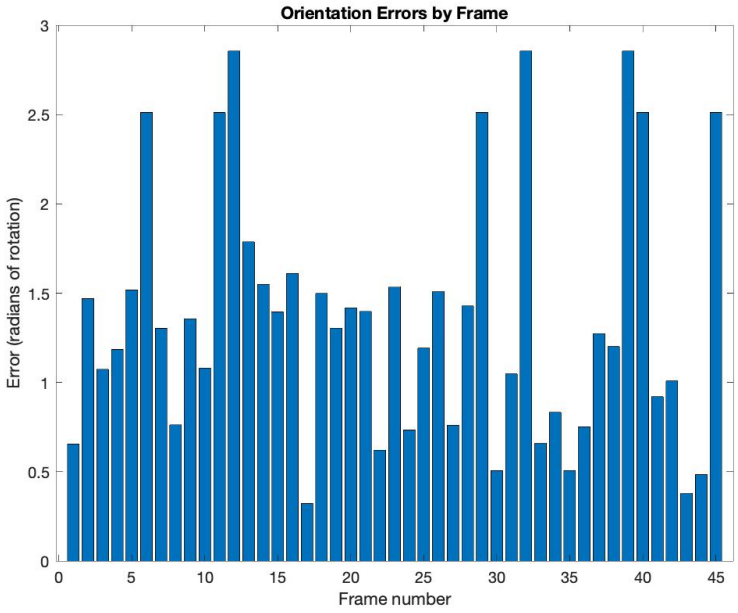


Figure 18: Orientation errors by frame for intermediary results that had incorrect quaternions

can be seen in Figure 19. In Figure 19(a) a wildly incorrect pose estimate can be seen. The bounding box is distorted and the axis are not in the right orientation. For comparison, Figure 19(b) shows the pose estimation from our algorithm for the same frame when the conversion of axis convention is calculated correctly.

To get to the correct conversion of axis convention, we realized that we had multiplied the matrices in Equation 14 in the wrong order. The correct equation is

$$R_{solution} = R_{init} \times R_y(\beta) \tag{16}$$

The remainder of the section demonstrates our results after applying Equation 16.

Figure 20 shows six different frames that had good pose estimations as determined by looking at the axis drawn on the image. Figure 21 shows three other frames with good estimates and the corresponding pose estimates calculated by AR track alvar. The top row of images are the poses determined by AR track alvar and the bottom row is our poses. The position of the axis is on the center of the target for AR track alvar, while we chose to display the axis on the top left corner of the box. However, you can still directly compare orientations of the axis between the images.

Figure 22 shows that our algorithm works reasonably well even when there is some occlusion to parts of the box.

Figure 23 shows six frames where our pose estimation was poor. Most of these frames had the bounding box and axis rotated out of the plane at the wrong angle or had the wrong size for the bounding box. Two of the photos (the top right and bottom left) had bounding boxes and axes that were not remotely similar to the truth. Figure 24 shows the pose predicted by AR track alvar on the left and the pose predicted by our algorithm for the same frame on the right. It is clear to see that our prediction is very wrong.

These poor results can most likely be contributed to poor feature matching. There were probably either not enough features matched between the reference image and the query image to generate a correct homography. Or, there were many matched features but so many of them were incorrect matches between the two images that our RANSAC algorithm was not able to ignore enough outliers when calculating the homography. If we had more time to address these problems we could look into different feature matching algorithms that produce better matches or play with the threshold for what we consider a "good" enough match to include in the data fed to the RANSAC homography algorithm. Additionally, we could play with the threshold for the RANSAC algorithm of what is considered a good fit for a model so that if we have too

Figure 19: (a) Incorrect intermediary result when trying to convert our axis convention to match AR track alvar's (b) Correct conversion of our axis convention to AR track alvar's

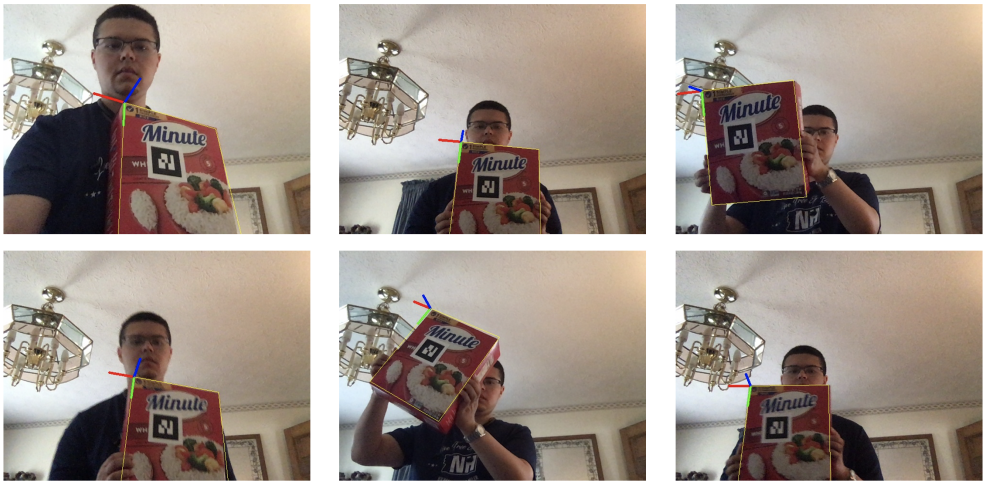


Figure 20: A sample of six analyzed frames that had good pose estimation

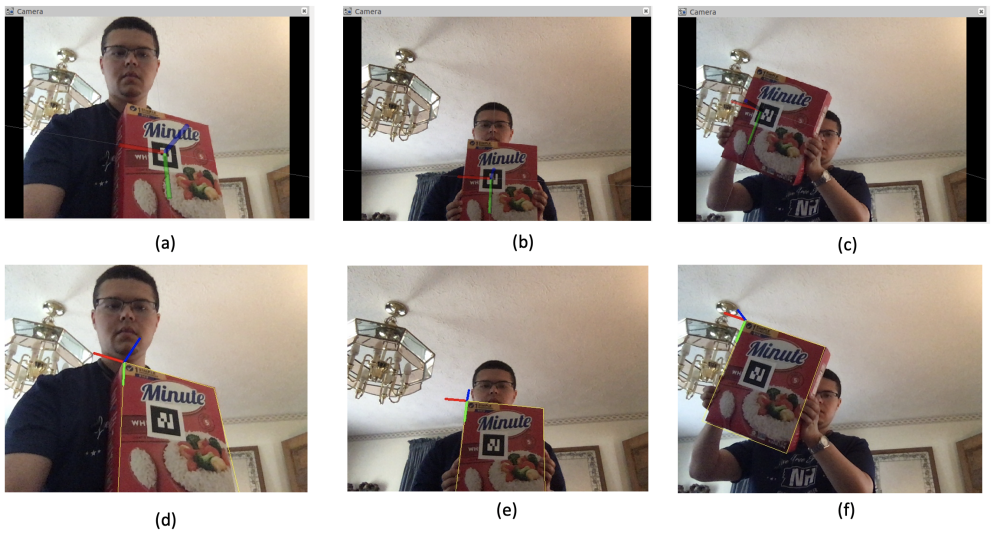


Figure 21: A sample of three analyzed frames that had similar pose estimations to those predicted by AR track alvar. The top row of images (a-c) has pose estimation determined by AR track alvar and the bottom row (d-f) has pose estimates for the corresponding frames calculated by our algorithm

Figure 22: Our results for object tracking with some occlusion

Figure 23: A sample of six analyzed frames that had bad pose estimation

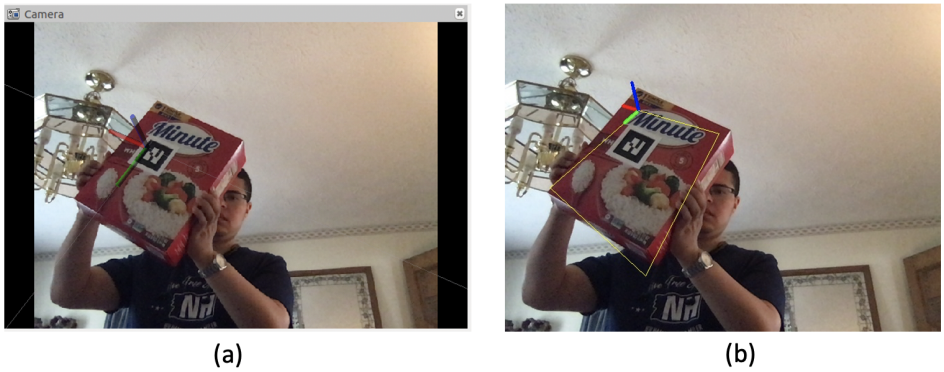


Figure 24: One frame that had poor pose estimations compared to those predicted by AR track alvar. (a) AR track alvar's pose estimation visualized in rviz (b) our pose estimation

many bad matched points, we simply do not predict a pose at all. We also might change how our video data is recorded. For instance, we could have better lighting and use a camera that has higher resolution.

Out of the 50 frames analyzed, there were a couple in which AR track alvar was not able to predict position and orientation because the AR tag was out of view of the camera. Figure 25 shows our results for those frames. The picture on the left seems to have the right orientation, but the bounding box is too large. For the picture on the right, the bounding box is approximately the right size but it is rotated out of the plane that the actual box lies in. However, these results are relatively good considering half of the box is occluded. Future work could consist of making the estimation more robust in situations like these.

Figure 26 shows the orientation error for our final results. This was calculated by Equation 15 as described previously. The average error was 0.6620 rad with a standard deviation of 0.9223 rad.

Figure 27 shows the position error for our final results. This was calculated by finding the Euclidean distance between our predicted position and what AR track alvar predicted. The average error was 1.0539 m with a standard deviation of 2.2983 m.

The averages and standard deviations for position and orientation error are greatly impacted by outliers in the data where our estimation was completely wrong (such as the top right and bottom left photos in Figure 23). If we filter out these outliers (where we consider an outlier being an error greater than 3 m in position or an error greater than 1 radian for orientation) we get the results in Figures 28 and 29. The

Figure 25: Our results for object tracking for frames that AR track alvar had no pose prediction

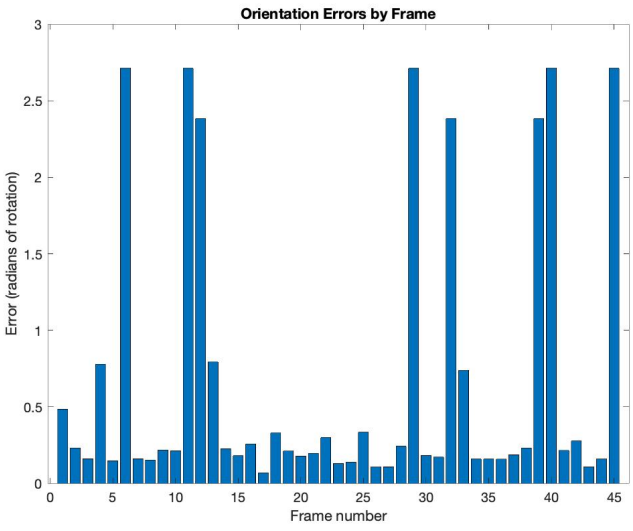


Figure 26: Orientation errors by frame for final results

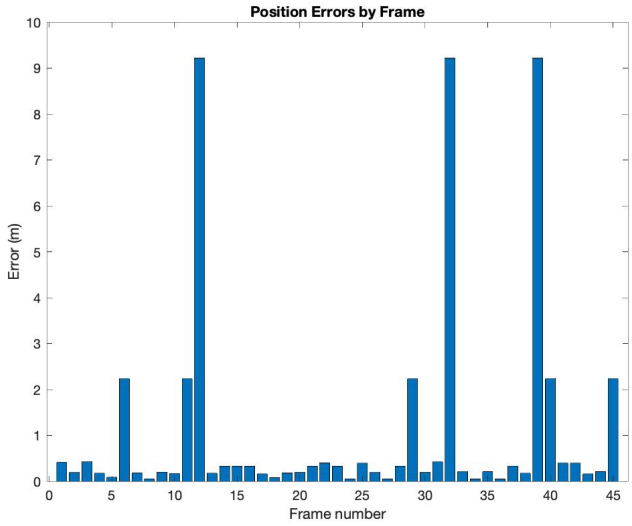


Figure 27: Position errors by frame for final results

average position error becomes 0.2325 m with a standard deviation of 0.1209 m and the orientation error becomes 0.2454 rad with a standard deviation of 0.1755 rad.

Visualization of the pose prediction by our algorithm for every frame analyzed can be seen in the results folder in our zip file. There is also an excel spreadsheet in that folder that has our position and orientation prediction (columns 2-8) and the position and orientation prediction of AR track alvar (columnes 9 - end) for every frame analyzed. Note that the rows for the AR track alvar pose calculations that are filled with zeroes are images that the package was not able to calculate a pose for.

2.4 Discussion

Overall we have positive thoughts about the project and its process as a whole. Considering the time constraint we had because our project initially started off as a drone project, we think we made reasonable progress and got decent results. Although there could definitely be improvements (as discussed in the Future Work section below), we ended up with a working product. Although frustrating at times, we found it rewarding when we were able to accurately determine pose on many of the images.

There are two main pieces of advice we would give ourselves before the project started. The first would be to read the vision related chapters of Peter Corke's *Robotics Vision and Control* textbook. We spent a great deal of time looking at lecture slides from different schools and reading tutorials online for how to accomplish things in the project when much of the base knowledge could have been obtained from reading the textbook. We did not realize this until after we finished the project when we were doing Ex 4. Having read this would have filled in some gaps in our knowledge from the slides– especially for some of the OpenCV functions we re-implemented. The second piece of advice would be to be more thorough when looking through OpenCV documentation and tutorials. We followed the Python tutorials for our project, but towards the end of the project we realized that the C++ tutorials had better descriptions of the concepts used in the functions implemented in the OpenCV library. Additionally, paying more attention to the method signatures of functions we re-implemented would have helped with our implementation. The most obvious example of this is the function cv2.findHomography. One of the parameters taken into the function is a method for the calculation (such as RANSAC). When we were implementing a homography finding function we couldn't figure out why our results were so different and poor until we realized the key was that OpenCV used RANSAC and we did not.

We learned a lot during this project. Both of us had varying degrees of experience with OpenCV prior, but working on this project cemented skills with that library. We also learned about feature matching algorithms, homography, and pose detection– all things both of us had little experience or knowledge of previously. We

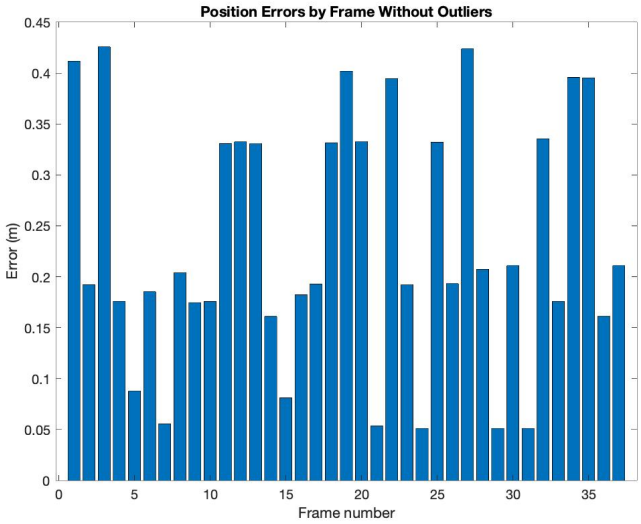


Figure 28: Position errors by frame for final results when outliers (errors greater than 3 m) are removed

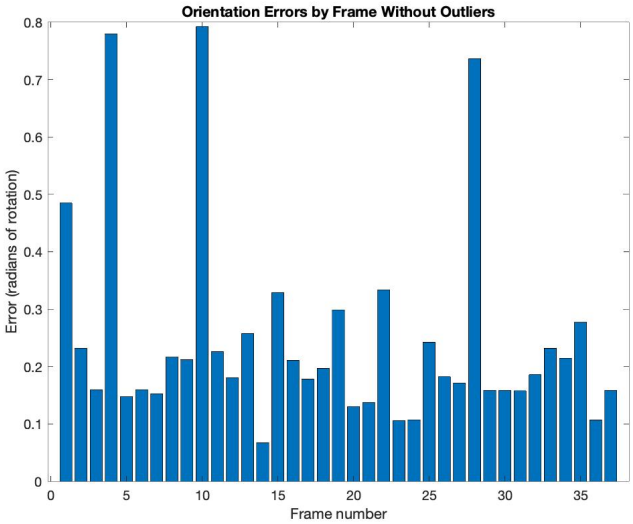


Figure 29: Orientation errors by frame for final results when outliers (errors greater than 1 rad) are removed

are excited by future robotics projects that may include some of the work we did. Most exciting would be being able to recognize multiple objects of different shapes and colors, putting this algorithm on a robot, and performing SLAM. In a more simple application of our algorithm, it would be cool if we were able to get the drone we worked with in the first half of the semester to search for the box and land on it.

2.5 Future Work

Although we finished all of the subtasks we aimed to, there is still much work that could be done on this project. The main thing that could be done is further work to increase the accuracy of the bounding box and pose estimation. Both the bounding box and the pose estimation are calculated based off the homography matrix so this seems to be the main bottleneck for accuracy.

There are two things that come to mind for things to explore in order to get a more accurate homography matrix. The first is to further tune the RANSAC algorithm parameters such as the number of iterations to run the algorithm, the number of nearby points required to assert a good model fit, and the distance threshold which determines whether a point is an inlier. A significant amount of time was spent tuning these parameters and while this led to more consistent results for each image, there is still some variance in getting the best homography matrix when running the algorithm on the same image multiple times. OpenCV, on the otherhand, always returns the same result for a particular image. If we had more time we would devote some of this to further playing with these parameters. Another thing with RANSAC that we did not have time to try that might be interesting is playing with the model for the distance which is used to determine whether a point is an inlier or not. Right now, the distance model is the sum of squared errors for the transformed points calculated using a homography matrix and the true values in image (see Equation 5). This distance could be changed to a symmetric epipolar distance which accounts for both the forward and reverse directions:

$$\sum_i d(p_i', H * p_i)^2 + d(pi, H^{-1} * p_i')^2 \tag{17}$$

Tuning the RANSAC algorithm for homography will help get better and more consistent results by ignoring outliers, but an additional approach is to simply give the RANSAC algorithm matching feature points with fewer outliers. This means that feature matching would need to be more accurate. With our current setup there are currently many bad matches when the box is in a scene and not isolated. Some things that may address this is trying SURF or SIFT instead of ORB and trying different matching algorithms for the keypoints and descriptors returned from these algorithms. Other options include getting better data, such as images that have better illumination.

Other possible directions of future work include making the detection more robust to occlusion and background noise and finding a better way to isolate regions of interest. Making the algorithm more robust to occlusion and background noise may require use of machine learning and deep learning methods to better recognize the object. In regards to determining regions of interest (roi), currently the roi is determined by using an HSV filter, but this requires tuning the lower and upper bounds of the HSV values which is time consuming and different depending on the lighting and the device used to record video.

A different direction to take the project in would be to implement an Extended Kalman Filter to determine the pose of the rice box. This could be implemented if we took the video from robot where we knew something like the odometry measurements. We could then perform EKF for the pose of the box similar to how we used EKF to predict landmark locations in Ex3.

2.6 Resources

2.6.1 Existing Packages Used

A lot of resources were used in the development of this project. The main libraries/packages used were OpenCV, Numpy, Scipy, and AR track alvar. OpenCV, which is an open source computer vision library in C++ and Python, was used for most of the image tasks and was a source of inspiration for functions that we re-implemented (namely cv2.perspectiveTransform, cv2.solvePnP, cv2.findHomography (with RANSAC), and cv2.projectPoints). It (and its documentation and tutorials [14]) were the most heavily used resources for this project. Numpy was used to do the matrix operations needed in the re-implementation of OpenCV

functions and Scipy was used to calculate the quaternion from the rotation matrix. AR track alvar was the ROS package described earlier that determines a ground truth pose based on the AR tag on an object.

2.6.2 Tutorial On Rviz Visualization for AR Track Alvar

With the aid from the resources described in Section 2.6.4, the use of most of existing packages utilized in the project were relatively straight-forward. It should be noted that for AR track alvar, a pose cannot be computed without a topic being specified that has CameraInfo messages. As such, camera calibration is required (see resources for this in Section 2.6.4). One part of using AR track alvar that was hard to figure out was visualizing the pose axis for a frame in rviz. The remainder of this section has some basic instructions on how to do this.

1. Install *rviz* using *sudo apt-get install rviz*.
2. We can launch the rviz application from the terminal using the command *rosrun rviz rviz*. If we have saved a configuration file earlier we can run this by navigating to this path and then using the command *rosrun rviz rviz -d filename.rviz*.
3. Next, we edit the settings for the Display's *Global Options*. Set the *Fixed Frame* as the topic published by the *ar_track_alvar*. In our case, it was *ar_marker_2*.
4. Now, we need to add a camera to rviz. This can be done by clicking on the *add* button and then from the *rviz* dropdown in the *By Display Type* tab select *Camera* and click *Ok*.
5. You can see a new setting that is visible in the *Displays* window. Here we need to change the *Image Topic* to *webcam/image_raw*. This should be the topic on which the image_raw is published. You should be able to see the video feed on the *Camera* window.
6. The next step is to view the Pose as 3D axi es. To add this, click on the *add* button and then from the *rviz* dropdown in the *By Display Type* tab select *Axes* and click *Ok*.
7. Now you should be able to visualize the pose using the 3D axes on the AR Tag in the video feed. We could resize the length and radius of the axes by changing the values for *Length* and *Radius* in the settings under *Axes* in the *Display* window.
8. If we want to save the configuration we should click on *File* and select *Save Config As*. In the save window select the required location and provide a name for the config file. Next click save button.

An example of the final configuration can be seen in Figure 30.

2.6.3 Resources Created By Us

In terms of what we can contribute, a zip file has been submitted with all our code used for this project. The results folder contains spreadsheets with our calculated pose and the true pose, as well as images with the calculated bounding box and axis drawn on. The results_attempt1 folder contains the same information, but its spreadsheet results do not have position calculated and the results_attempt2 has the correct axis drawn on the images but the quaternions for the pose in the csv file are incorrect. In the scripts folder are all relevant files to the project. It includes tracking_class.py which has all of the logic for tracking the box and pose_class.py which has our version of OpenCV functions that we implemented. The Matlab files used to analyze our results are included in the scripts folder as well. The folder also has files for several ROS nodes that do things like publish 50 select messages, publish camera info, and save select frames from an Image topic.

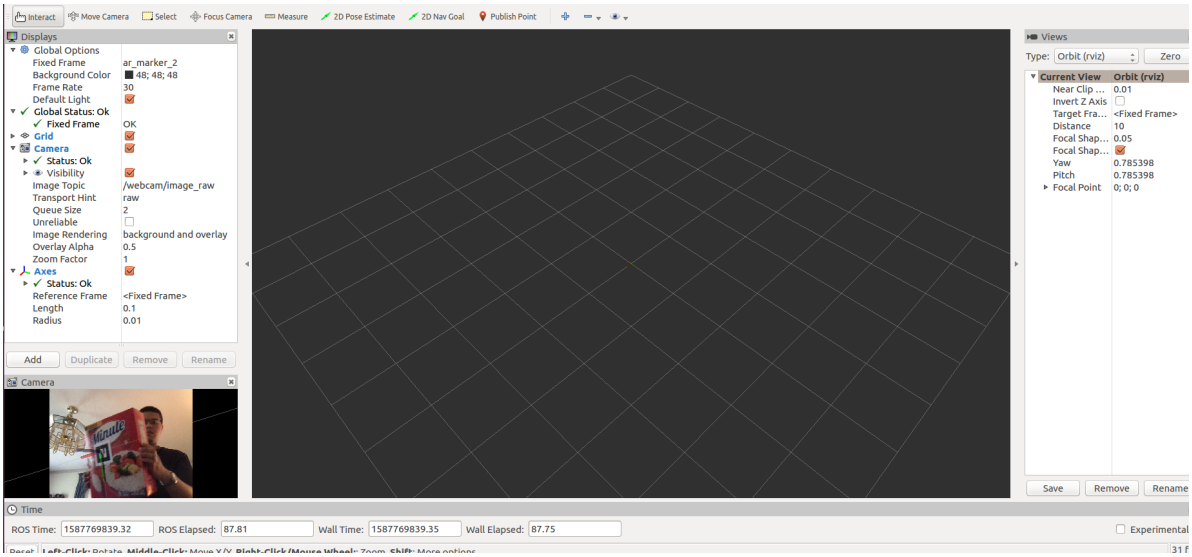


Figure 30: Final rviz configurations for displaying axis from AR track alvar on frame

2.6.4 Resources Broken Down By Topic

The below breakdown shows what resources where used for which portion of the project:

- Using ROS: [1]
- Using OpenCV: [14], [27], [28]
- Using AR Track Alvar: [21], [29], [30], [31]
- Using Numpy and Scipy: [32]
- Feature Matching: [20]
- Camera Calibration: [33], [34], [35]
- RANSAC: [36], [37], [38], [39], [40]
- Homography and Pose Estimation: [41], [24], [21], [42], [43], [23], [44], [36]
- Estimating Position: [25]

References

[1] Ros tutorials. http://wiki.ros.org/ROS/Tutorials.

[2] Hello (real) world with ros – robot operating system course. https://www.edx.org/course/hello-real-world-with-ros-robot-operating-system/.

[3] Ros wiki drone simulation. http://wiki.ros.org/hector_quadrotor/.

[4] Duckiedrone operation manual. https://docs.duckietown.org/daffy/opmanual_sky/out/index.html/.

[5] Cleanflight flight controller flashing tool. http://cleanflight.com/.

[6] Duckiedrone ubuntu image. https://drive.google.com/file/d/1ogPrxXBpXa6Tbv3xpYZcvCc_7EXD-w7S/view.

[7] Sd card image etcher. https://www.balena.io/etcher/.

[8] pidrone_pkg github repo. https://github.com/h2r/pidrone_pkg.

[9] Duckiesky learning materials. https://docs.duckietown.org/daffy/doc-sky/out/index.html.

[10] Calculating homography. http://www.csc.kth.se/~perrose/files/pose-init-model/node17.html.

[11] Quaternion distance. https://math.stackexchange.com/questions/90081/quaternion-distance.

[12] Ar track alvar. http://wiki.ros.org/ar_track_alvar.

[13] Video stream opencv. http://wiki.ros.org/video_stream_opencv.

[14] Alexander Mordvintsev and Abid K. Opencv-python tutorials. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.htm.

[15] Opencv python pip. https://pypi.org/project/opencv-python/.

[16] Changing colorspaces. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html#converting-colorspaces.

[17] Trackbar as the color pallete. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_trackbar/py_trackbar.html#trackbar.

[18] Feature matching and homography. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html#feature-homography.

[19] Feature matching. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html#matcher.

[20] Opencv orb class reference. https://docs.opencv.org/3.4/db/d95/classcv_1_1ORB.html.

[21] Ee106a lab 4: Image manipulation, camera calibration, and ar tags. https://bcourses.berkeley.edu/files/65410901/download?download_frd=1.

[22] Pose estimation. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_pose/py_pose.html#pose-estimation.

[23] Basic concepts of the homography explained with code. http://man.hubwiz.com/docset/OpenCV.docset/Contents/Resources/Documents/d9/dab/tutorial_homography.html.

[24] Scipy rotation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.html.

[25] Vision target white paper. https://forums.ni.com/t5/FIRST-Robotics-Competition/FRC-2012-Vision-Target-White-Paper/ta-p/3522837?profile.language=en.

[26] Ar track alvar launch files. https://github.com/ros-perception/ar_track_alvar/tree/melodic-devel/ar_track_alvar/launch.

[27] Opencv-python tutorials c++. https://docs.opencv.org/master/d9/df8/tutorial_root.html.

[28] Camera calibration and 3d reconstruction. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findhomography.

[29] Austin Buchan. Ar tag tutorial. https://piazza.com/class_profile/get_resource/hysvddrwjpvg5/i252vexju0u5tb.

[30] Tracking ar tags with ros: Monocular vision. http://projectsfromtech.blogspot.com/2017/09/tracking-ar-tags-with-ros-monocular.html.

[31] Output frame on ar track alvar. https://answers.ros.org/question/152866/output_frame-on-ar_track_alvar/.

[32] Numpy and scipy documentation. https://docs.scipy.org/doc/.

[33] How to calibrate a monocular camera. http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration.

[34] Yaml to camera info publisher. https://gist.github.com/rossbar/ebb282c3b73c41c1404123de6cea4771.

[35] Camera calibration package. http://wiki.ros.org/camera_calibration.

[36] Mit csail homographies and ransac. http://6.869.csail.mit.edu/fa12/lectures/lecture13ransac/lecture13ransac.pdf.

[37] Pseudocode for the ransac algorithm. http://learning.eng.cam.ac.uk/pub/Public/Turner/Teaching/ransac.pdf.

[38] Cs 4495 computer vision ransac. https://www.cc.gatech.edu/~afb/classes/CS4495-Fall2014/slides/CS4495-Ransac.pdf.

[39] Robust estimation: Ransac. http://www.inf.u-szeged.hu/~kato/teaching/computervision/Lab05-RANSAC.pdf.

[40] Cse486 lecture 15 robust estimation: Ransac. http://www.cse.psu.edu/~rtc12/CSE486/lecture15.pdf.

[41] Pose estimation. http://users.isr.ist.utl.pt/~jsm/teaching/piv/Pose.pdf.

[42] Kenji Hata and Silvio Savarese. Cs231a course notes 1: Camera models. https://web.stanford.edu/class/cs231a/course_notes/01-camera-models.pdf.

[43] Cmu homography slides. http://16720.courses.cs.cmu.edu/lec/transformations.pdf.

[44] How does the perspectivetransform() function work? https://answers.opencv.org/question/54886/how-does-the-perspectivetransform-function-work/.