Maritime Recovery of a Quadrotor

by

Midshipman 1/C Francis Cunniff

A Capstone Project Report Submitted to the Faculty of
The Weapons, Robotics and Control Engineering Department
United States Naval Academy, Annapolis, Maryland

Faculty Advisor: Prof Matthew Feemster

Department Chair: Prof Brad Bishop

30 April 2019

Contents

Maritime Recovery of a Quadrotor

Authors: Midshipman 1/C Francis Cunniff
Contact Information: fxcunniff@gmail.com

***Abstract*—Increased functionality in commercial-off-the-shelf quadrotors allows for more advanced uses in research purposes. The design of a framework that allows for the integration of sensors as well as the implementation of controllers is necessary to enable the full potential of these quadrotors. Using the Robot Operating System, and the scenario of a quadrotor landing autonomously on an unmanned surface vessel, a process for implementing the sense, act, decide principles of Robotics and Controls Engineering can be developed and serve as a baseline for future applications of these quadrotors beyond what they are designed to do from the factory.**

INTRODUCTION

Motivation

The increased availability of commercial-off-the-shelf quadrotors opens up several opportunities for their use in situations that typically would require a purpose built quadrotor. One example of this is operations over open water in which the operator of the quadrotor may not have line of sight or a first person view from the quadrotor. In this scenario, being able to land the quadrotor in a highly accurate manner is crucial to ensuring that the quadrotor can be retrieved and used to fly again. This situation and the assumptions that come with it will be the basis for the design and experimentation in this project, however the principles used apply very generally to mobile robotics and autonomous vehicles as a whole.

Problem Statement

The goal of this project is to develop an autonomous system that promotes recovery of an aerial vehicle in a maritime environment. This part of the project will focus on the required software development to allow for autonomous landing of a commercial off the shelf (COTS) quadrotor

Related Work

Team M-SCAPE: [2] One of the goals of this project involved the recovery of the quadrotor at the end of the flight. A problem with this project as it relates using a COTS quadrotor is that it became a very aerospace engineering focused project and the design of the quadrotor and sensors used could never be used on a scale like inexpensive, preassembled commercial quadrotors can. Additionally, with focus on design, finals controls implementation was developing but ultimately lacking for a full on autonomous landing. The highest reported accuracy towards landing on a fixed point was 3m, which is now set as a baseline from which to improve upon.

Design Process

Customer Interviews and Requirements Generation

The ultimate goal based of the customer interviews is to develop a system that could both land and takeoff from an unmanned surface vessel autonomously. Considerations early in the project limited the scope to quadrotors so as not to turn the project into comparing various types of unmanned aerial vehicles. The use of commercial-off-the-shelf components meant that the quadrotor was limited to onboard sensors and would run using the stock firmware, meaning that only minimal modification would be required to turn a quadrotor out of the box into one compatible with the system.

Functions and Morphological Chart

In order to successfully execute the tasking, the hardware and software would be selected based on their utility in fulfilling the functional requirements outlined above. Fitting into the broader concepts of Robotics and Controls Engineering, the system will need to be able to sense, act, and decide. Action will come in the form of the physical quadrotor. Sensing will be the sensing package that best accomplishes the goal. Decision will be handled by a controller designed to optimize accuracy. Communications between these functions is often an overlooked aspect of the process, and so it is also included as a relevant function. Any subsystem capable of sensing, acting, or deciding will need adequate communication capabilities to perform properly.

Table 1: Morphological chart

	Concept 1	Concept 2	Concept 3	Concept 4
Communications	Wi-Fi	Bluetooth	Xbee	Radio/RC
Position Sensor	GPS	IMU	Computer Vision	Motion Capture
Processor Location	USV	Quadrotor	Shore Station	
End-Course Controllable Mobility	USV only	Quadrotor only	USV and Quadrotor	Dynamic positioning USV and Quadrotor

Decision Matrix and Preliminary Design

Table 2: Decision Matrix

	Weight	Tail hook Fixed Wing/Yard Patrol Craft	Kingfisher/Floatplane	Quadrotor/USS Feemster
Recovery Probability	4	2/8	4/16	4/16
Speed of Recovery	2	3/6	1/ 2	4/8
Algorithm Speed	2	2/4	2/4	2/4
Redundant Systems	2	2/2	2/4	3/6
Total		20/40	26/40	34/40

The initial decision matrix was developed at a point in the project prior to narrowing the focus to quadrotors specifically. This allowed for greater focus of on a particular problem rather evaluating the merits of fixed wing vs. rotary wing aircraft which was outside the scope of this project. Additionally, while mentioned in previous tables, the unmanned surface vessel is not subject to any design considerations or action in this project.

Ethical Considerations

The ethical implications of this research are fairly limited, but a system like this could be used for minimally invasive monitoring of wildlife areas without requiring human contact. The research methods and processes will all take place in a laboratory setting and all equipment used is vetted by TSD to ensure it meets standards of safety.

Engineering Analysis and Prototype Development

The design involved occurred almost entirely at the software level. Due to the nature of using COTS components, hardware design was intentionally kept to a minimum. For initial testing, a Parrot Ardrone 2.0 was available for use and experimentation to allow for a more practical understanding of the characteristics of a quadrotor and what could be done to interact with them on the software side. Whatever quadrotor was used would have to be able to communicate with a PC over Wi-Fi. The controller developed would have to be able to read data from the motion capture system.

Component Selection

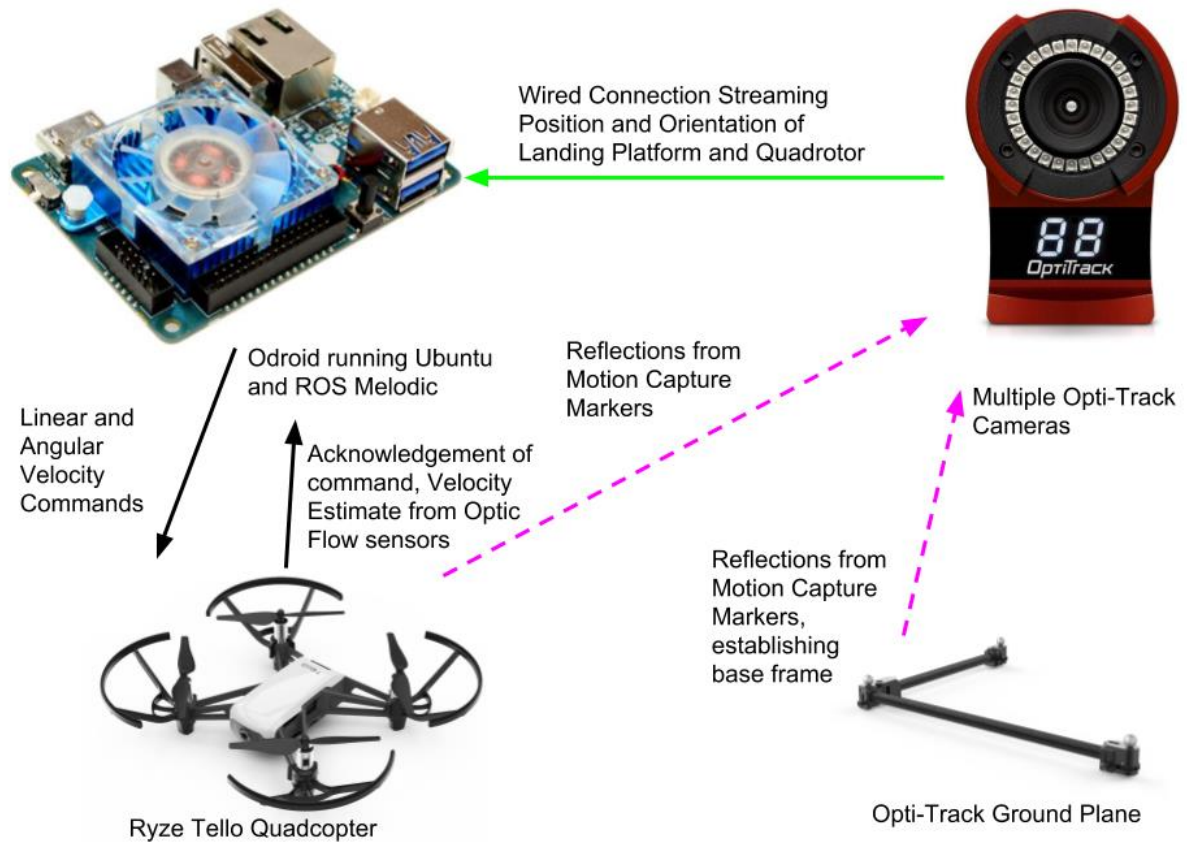
The selection of components was based on both how well suited a given component would perform its task as well as how reliably it could be implemented into the overall system that was forming. Motion capture as the sensing subsystem was performed by the Opti-Track system because it works, requires only slight, cosmetic modifications to the quadrotor, and was available. The Tello quadrotor used was selected based off its incredibly stable hover and the availability of a recently published software development kit form the manufacturer. The unmanned surface vessel, which was not to be designed in this project per the initial design requirements, would be assumed to be operating in a dynamic positioning mode. This means that for testing purposes we could assume that the USV or its stand in would have sufficient sensors and actuators to remain fixed at a point on the earth's surface.

Design Evolution

The component selection above did not list the Parrot Ardrone 2.0 intentionally. The design eventually moved away from the Ardrone as all software development for it ceased in 2012, which caused some issues inherent to the Ardrone such as not holding position reliably. Interfacing with the 2012 software would require using legacy versions of software, and so the final decision involved getting a newer and better supported drone so that the most relevant and capable features of the other systems could be used to their full potential.

Final Design

Overview



Wired Connection Streaming Position and Orientation of Landing Platform and Quadrotor
Odroid running Ubuntu and ROS Melodic
Reflections from Motion Capture Markers
Multiple Opti-Track Cameras
Linear and Angular Velocity Commands
Acknowledgement of command, Velocity Estimate from Optic Flow sensors
Reflections from Motion Capture Markers, establishing base frame
Ryze Tello Quadcopter
Opti-Track Ground Plane

Figure 1: Diagram visualizing how components relate to each other. Green lines are wired, black is wireless, purple is visual.

Mechanical Subsystem

For this phase of the project the assumption is that whatever unmanned surface vessel will be used is operating under dynamic positioning, meaning that internal processes are allowing the USV to maintain its position relative to a fixed point on the earth’s surface. This assumption is not baseless, as many commercially available trolling motors have GPS based dynamic positioning that allows small craft that do not desire to put down anchor to stay in a small area and not drift. With the assumption that an unmanned surface vessel with some dynamic positioning system will be used, any stationary platform can be substituted for it within the lab. The Tello drone itself is already designed to fly from the factory. It required no assembly or modification other than the Opti-Track markers that are taped on.

With regards to the operation of the Tello drone, the commands send indicate changes in orientation and not position. This meant that the drone could not simply be told to go forward/backward/left/right, but instead would be told to pitch forward 5 degrees or increase vertical velocity by 0.5m/s. The Tello drone was chosen specifically because it had all the onboard sensing capabilities desired.

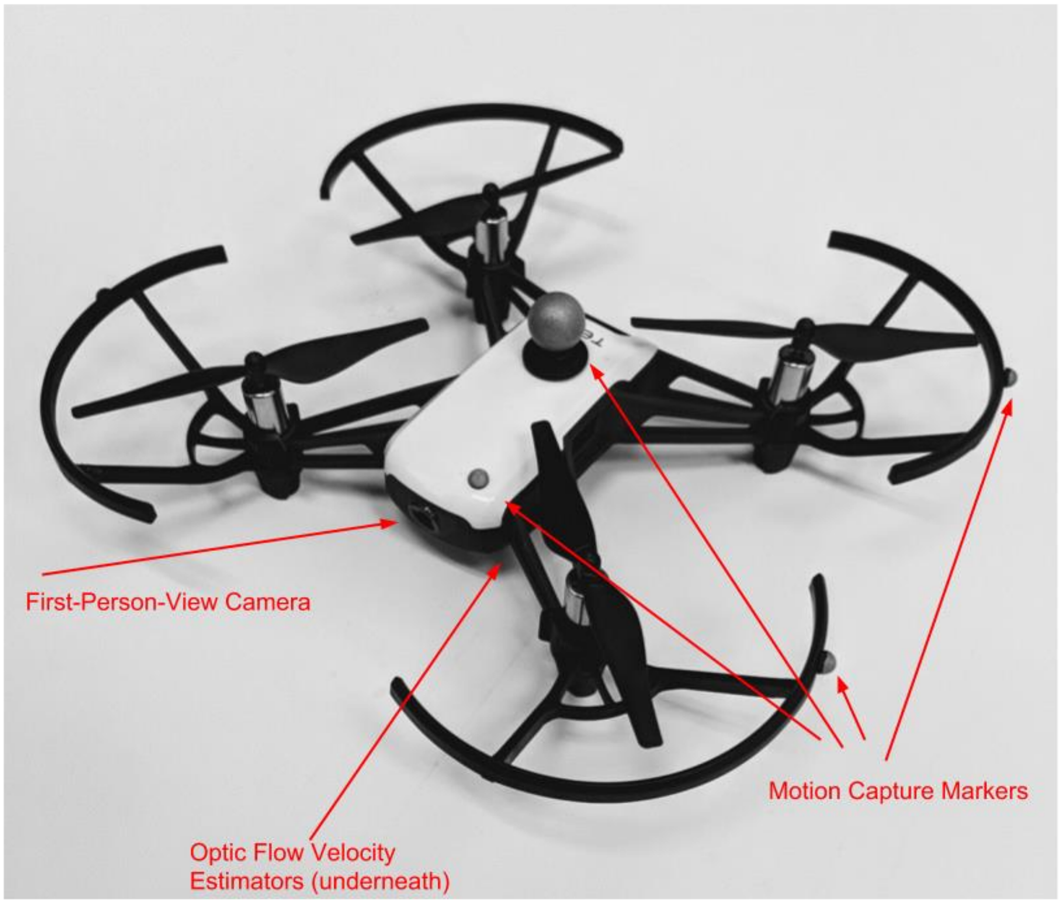


Figure 2: Ryze Tello quadrotor as modified for use with Opti-Track System.

Electrical Subsystem

All electrical components were left in stock configuration and no electrical components were developed specifically for this lab. The Tello quadrotor operates its DC motors off 3.8V, 1Ah detachable batteries that are swapped through periodically. All other components operate off outlet power from the lab.

Software Subsystem

The software subsystem consisted of several machines running different operating systems. To mitigate the issues associated with communications between these machines, the Robot Operating System (ROS) was used to allow for coherent and quick sharing of information between the various processes, sensors, and actuators in the system. ROS operates in terms of nodes, which are programs that handle information. Information in ROS is in the form of a standard message type appropriate for the information being conveyed. These messages that can include everything from Booleans that give information such as the power status of the Tello, to something like a “twist” data type, which includes linear and angular velocity in the X, Y, and Z axis.

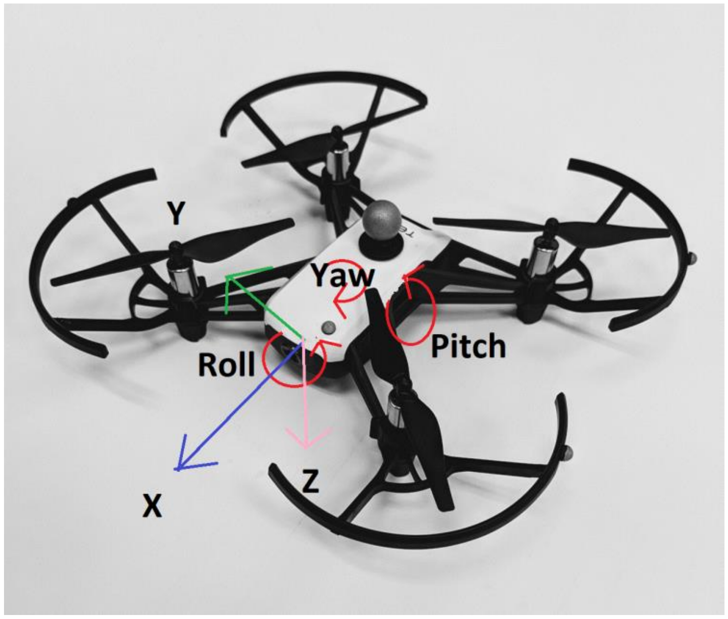


Figure 3: Tello with world frame coordinates labeled. Initial control algorithm assigns variable. Body frame aligning with world frame is first control operation. Note that Opti-Track defaults to a "Y up" coordinate frame instead of a more traditional "Z down"

ROS topics, consisting of one or more of these messages, are passed into nodes based on what topics the node is subscribed to. This allows for all information to be available to all nodes running on the same connection, but does not slow computing time as a given node is only receiving information from the relevant topics it is subscribed to. A node that performs operations on the information it receives can then publish the data back into ROS. Multiple topics can be published to and subscribed from a single node. The benefit of this is that once a node is known to function correctly through testing, it becomes simpler to add additional processes to the system as long as the arguments into and out of the node remain unchanged. In this project, the control loop took place from the initial input in the form of a Pose Stamped message from the vrpn_client_node.

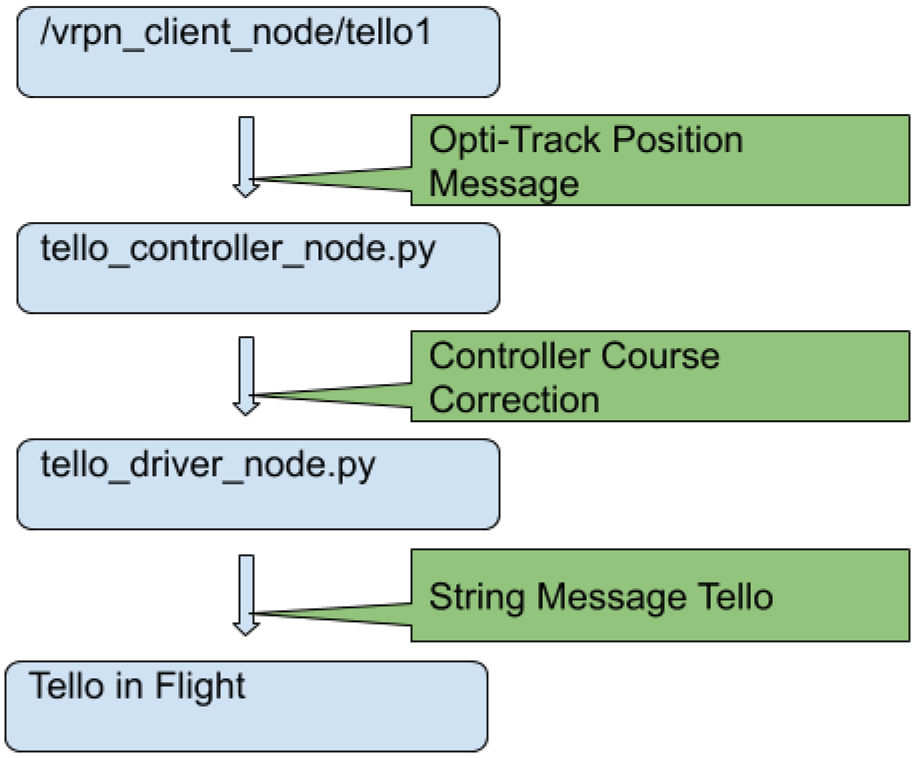


Figure 4: Graphic demonstrating the path and conversion of topics. Nodes are indicated by "_.py", topics begin with a "/"

This information included the position of the Tello drone in the room as determined by Opti-Track, as well as its roll, pitch, and yaw. Any desired landing platform would be labelled with markers and similarly sensed. The controller node is subscribed to the information regarding the location of the Tello and landing platform, and inside of the control node the error between the desired position (the landing platform) and the measured position (the Tello drone) is determined. Inside the same node this information is compared to thresholds to apply a proportional change in position if not in the desired area. The information regarding what changes must be applied to the Tello's location does not go from this node directly to the drone. Instead, this information then goes to the tello_driver_node [1], which receives controls solution, and generates a string that it will send over Wi-Fi to the Tello to allow for the commanded changes in orientation and position to take place.

Feedback Control

The controller used is a proportional controller that uses the difference between the desired and measured position and implements a roll or pitch to cause the necessary translation. First the yaw is controlled to align the body frame of the quadrotor with world frame of the lab. At this point the primary axes of the coordinate frame and the ability of the drone to translate are aligned. The roll and pitch are variable on the Tello between -10 and 10 degrees, and so with all measurements in motive being on the order of millimeters, a gain, Kp of .01 was chosen so that the roll and pitch would be fully saturated until within 1 meter of the desired landing area. The onboard flight controller does not allow the Tello to lose altitude when it pitches or rolls, so all altitude adjustments are saved for last. The maximum upward acceleration of the Tello occurs during takeoff, and the fastest downward acceleration is the force of gravity when the electric motors cut out. In order to get an expeditious but controlled descent when over the desired landing area, vertical velocity is set to -0.5m/s, which usually only operates for a second before the drone is in the threshold for the landing command to be sent

$$Yaw = Ktheta(\Theta_{des} - \Theta) \tag{1}$$

$$Roll = Kp(x_{des} - x) \tag{2}$$

$$Pitch = Kp(y_{des} - y) \tag{3}$$

$$Altitude = Kalt(z_{des} - z) \tag{4}$$

The code is written in python, and is similar to this pseudocode:

```
while(1)

Read Landing Platform Position

Read Quadrotor Position from Opti-Track

Calculate error between positions

Send horizontal velocity command to
Quadrotor

Send position hold command to Landing
Platform

if error < margin{

     Attempt to land quadrotor

else{

     Return quadrotor to safe altitude
     and await next horizontal velocity
     command}}

end and send landing confirmation
```

Figure 5: Control Loop Pseudocode

The control solution chosen was a simple proportional controller for several reasons. For testing the interactions of multiple processes running on this system at very high data rates, proportional control was simple and fast. The location at which the controls solution is implemented in ROS lends itself to python code, and in order to ensure the most reliable performance from an inexperienced python coder relatively simple operations are used. Later testing will mention use of a joystick which is using a human being to close to the loop, please note that this is just a diagnostic tool and not an actual attempt to add a human operator to a supposedly autonomous vehicle.

Results and Analysis

Demonstration Plan

The demonstration will take place in the MU201 lab where the fixed Opti-Track system is mounted. A preliminary calibration is performed to ensure the greatest accuracy and account for any sag or shifting in the camera mounts since the last time the system was used. A calibration that is accurate to within 0.5mm requires about 15 minutes of "wanding" the room with a reference wand. After this is completed, two objects must be established in the Motive software as rigid bodies. This is accomplished by taking the objects intended to track and placing the motion-capture markers on them in such a way as to avoid symmetry so that distinct patterns of the markers emerge. In the case of this lab, the ground plane provided with the Opti-Track System will serve as the landing platform or notional unmanned surface vessel. The Tello drone has markers attached to it, and both objects are selected and represented in motive as relationships of the markers floating in space. When the objects are initialized, the ground plane will align along the X and Y axis indicated on the floor of the lab. The Tello drone will be aligned with its camera facing in the position of the positive X axis as well so as to align the body frame of the Tello drone and world frame initially. Establishing rigid bodies in Motive turns the markers from orange to blue, and draws lines between all markers associated with an object. After establishing a rigid body, it is named appropriately with no spaces or punctuation. With an accurate calibration and well-fixed markers, the other settings associated with rigid bodies in Motive should be left default. Ensure that vrpn streaming is selected in the streaming settings.

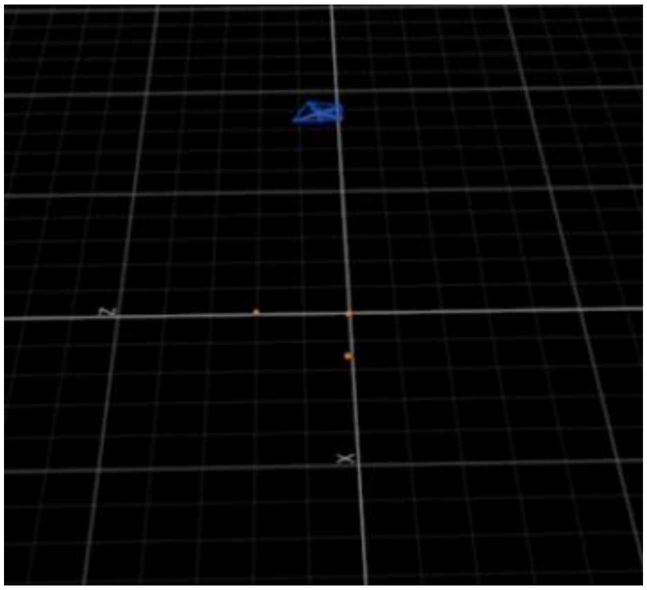


Figure 6: Tello as a rigid body in Opti-Track (blue). Ground plane as 3 orange markers.

Both the PC in the MU201 lab as well as whatever additional Linux PC is running ROS nodes should be connected to a common network. Pinging the various machines on the network ensures that there are no duplicate IP addresses. Due to the subnet mask and relative few machines on the network, an IP address change may be in order so as not to conflict with existing machines on the system. Verify connection by pinging 10.1.1.200, the Odroid's IP address. Upon successful return, open a window in an SSH utility such as Tera Term. SSH into the odroid@10.1.1.200. This step is easily performed on the MU201 lab PC which has Tera Term installed and a hardwired connection into a router that serves the Odroid. Once SSH is successful a terminal will open with $odroid@odroid as the username. The command roscore will start ROS's core services and allow the ROS master to be started. This will be responsible for routing the communications through the system and enabling the functionality of ROS. This window should be left open to enable monitoring later.

On the Linux PC on which controls are running, open a terminal. The standard process for opening nodes in ROS making topics available is to indicate

```
rosrun desired_package desired_node
```

A list of all ROS topics operating on the system are available at any time by opening a new terminal in Linux and typing the command

```
rostopic list
```

Which should populate with data relevant to all processes that have been started on that particular node. In the case of the specific nodes utilized for this purpose, within the tello_driver package, the following nodes must be launched, vprn_client, tello_driver_node, and either tello_joy or tello_controller, depending on whether the input is desired to come from a manual joystick or the controller implemented. It is good practice to open another terminal after running these commands to ensure the relevant topics are available before progressing any further with the process.

For the commanded control input, first initialize the tello_joy node. This will require manual inputs into an attached joystick to send data to the tello_driver and ultimately to the Tello. This step is useful as a diagnostic when the tello_controller_node is not responding or giving errors causing the flight of the Tello to become unpredictable. The current mapping developed for the Logitech 3D Pro controller indicates button #8 for takeoff and #7 for landing. Pitch, roll, and yaw are controller with the forward, lateral, and rotational motions of the joystick.

At this point, the controller node will be generating messages in a "Twist" format, a standard message type that the tello_driver_node is configured to receive and convert into the strings that are then pushed via Wi-Fi to the Tello to be executed. The Tello should respond to these commands, changing pitch first, then roll, in order to translate across the room while maintaining altitude. Once over the landing area, yaw will adjust to align the body and world frame. Once the orientation and position are within the thresholds established in the controller, vertical velocity will decrease until the internal sensors of the Tello prevent further descent due to ground effect, at which point the controller will reach the boundary at which it issues the land command. This should occur about 0.5m above the ground.

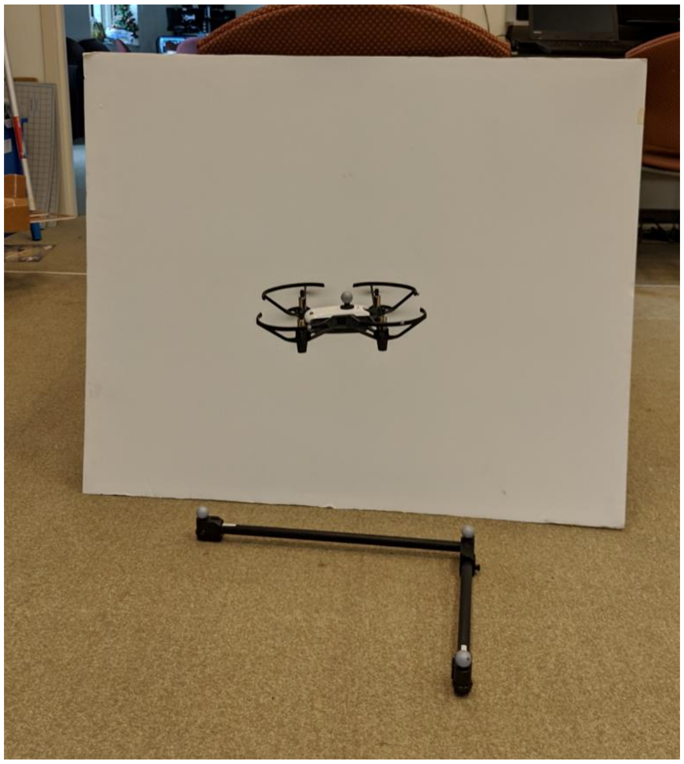


Figure 7: Tello begins landing sequence inside of ground plane.

Performance Measures

The ultimate lack of a reliable controller does not allow for any conclusive results as to the accuracy of the controller being sufficient to recovery a quadrotor over open water with any degree of confidence. Joystick teleoperation proved that the resolution of the actuators is fine enough that if the correct controls inputs can be generated, the Tello is capable of moving very small distances reliably. The onboard Optic-Flow technology also allows for the Tello to maintain position without any additional input from the Opti-Track system. These findings suggest that in the event a sufficient controller was developed, the Tello drone has the hardware capabilities to allow it to land autonomously on and unmanned surface vessel.

Project Management

Life Long Learning

There are four components of this lab that required additional research in order to effectively use. The Opti-Track system required an introduction with an instructor to understand the basics of how to calibrate the system as well as establishing individual objects. Linux Ubuntu was explored briefly in EC312, and so a review of how to operate from the terminal in Linux was also required. These two aspects of the life-long learning were relatively intuitive once introduced. Robot Operating System (ROS) Melodic was one of the more challenging aspects of the program. It had advantages that made it worth learning, but the system requires a period of familiarization in order to effectively understand exactly what ROS is doing based off commands specific to ROS. Python was a completely new programming language that was used both in existing ROS nodes as well as the controller node written for this lab, which did not require a full understanding of all of the particulars of python, but did require a thorough understanding of functions and the required libraries specific to mobile robotics applications in ROS.

Cost analysis and Parts List

Table 3: Material Costs

Ryze Tello Quadrotor	$120
Spare Batteries/Propellers	$60
Opti-Track and Motive	$16000
Controls PC	$1200
Odroid XU4Q	$80
Motion Capture Markers	$20
Materials Total	$17480

Table 4: Labor Costs

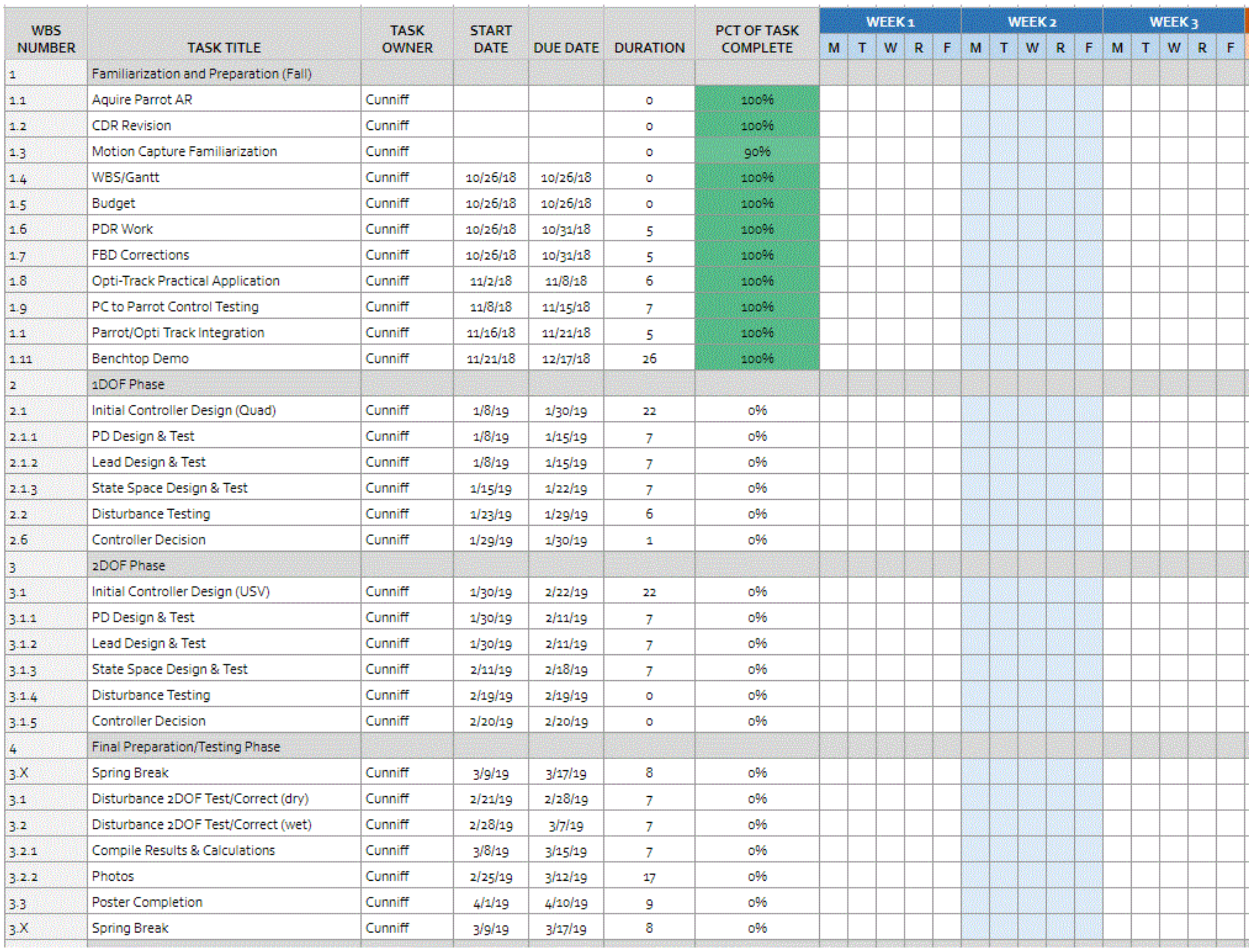
Midshipmen	$2400
Faculty	$1920
Shop and TSD	$1280
Direct Cost Subtotal	$5600
Fringe Benefits (Estimated 35% of direct costs)	$1960
Facilities (Estimated 50% of direct cost)	$2800
General Services (Estimated 15% of direct cost)	$840
Indirect Cost Subtotal	$5600

Note that of the $17480 total for materials, most equipment was available within the Robotics and Controls Engineering department. As a result, funding provided was $600 and covered the purchase of the Tello drone, spare parts for the drone, and the Odroid required to run the mobile camera setup.

Timeline

The original project proposal included a Gantt chart that outlined both Phase 1 and 2 of the project.

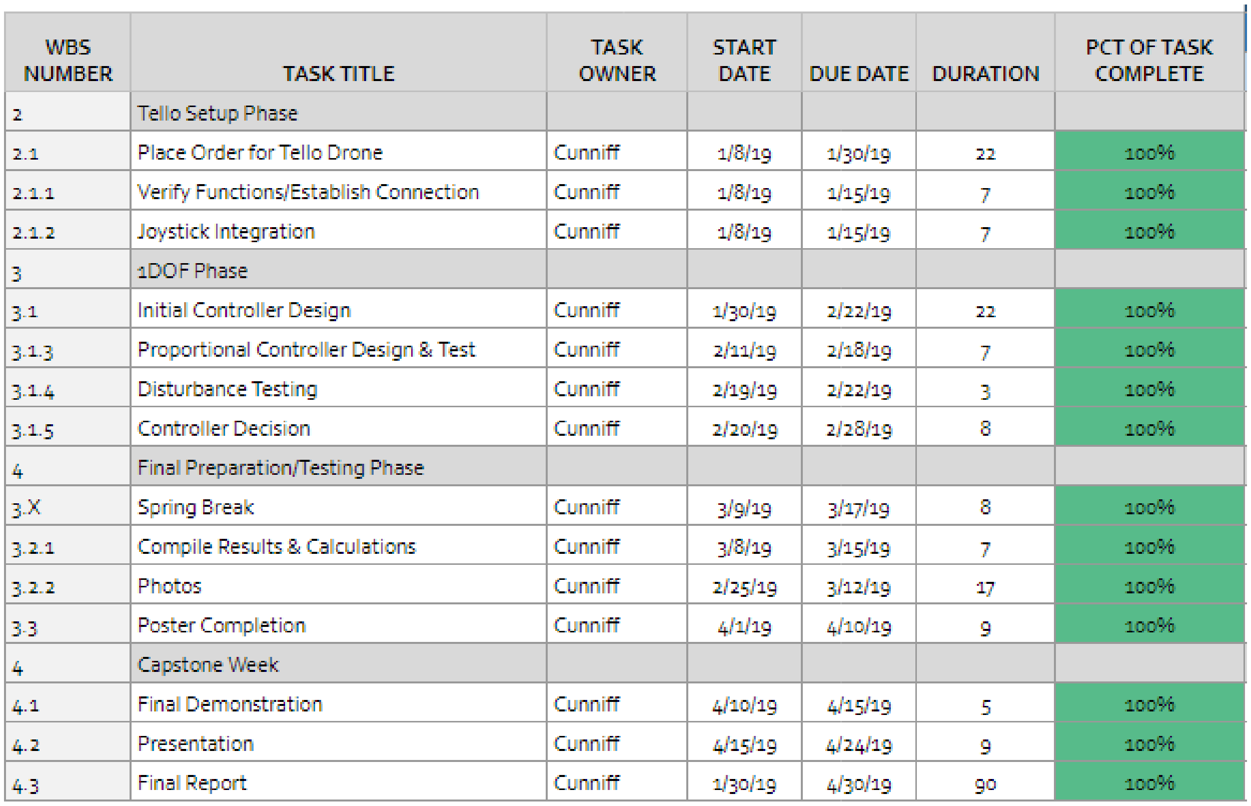
Table 5: Initial Gantt chart



							WEEK 1					WEEK 2					WEEK 3				
WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE	M	T	W	R	F	M	T	W	R	F	M	T	W	R	F
1	Familiarization and Preparation (Fall)																				
1.1	Aquire Parrot AR	Cunniff			0	100%															
1.2	CDR Revision	Cunniff			0	100%															
1.3	Motion Capture Familiarization	Cunniff			0	90%															
1.4	WBS/Gantt	Cunniff	10/26/18	10/26/18	0	100%															
1.5	Budget	Cunniff	10/26/18	10/26/18	0	100%															
1.6	PDR Work	Cunniff	10/26/18	10/31/18	5	100%															
1.7	FBD Corrections	Cunniff	10/26/18	10/31/18	5	100%															
1.8	Opti-Track Practical Application	Cunniff	11/2/18	11/8/18	6	100%															
1.9	PC to Parrot Control Testing	Cunniff	11/8/18	11/15/18	7	100%															
1.1	Parrot/Opti Track Integration	Cunniff	11/16/18	11/21/18	5	100%															
1.11	Benchtop Demo	Cunniff	11/21/18	12/17/18	26	100%															
2	1DOF Phase																				
2.1	Initial Controller Design (Quad)	Cunniff	1/8/19	1/30/19	22	0%															
2.1.1	PD Design & Test	Cunniff	1/8/19	1/15/19	7	0%															
2.1.2	Lead Design & Test	Cunniff	1/8/19	1/15/19	7	0%															
2.1.3	State Space Design & Test	Cunniff	1/15/19	1/22/19	7	0%															
2.2	Disturbance Testing	Cunniff	1/23/19	1/29/19	6	0%															
2.6	Controller Decision	Cunniff	1/29/19	1/30/19	1	0%															
3	2DOF Phase																				
3.1	Initial Controller Design (USV)	Cunniff	1/30/19	2/22/19	22	0%															
3.1.1	PD Design & Test	Cunniff	1/30/19	2/11/19	7	0%															
3.1.2	Lead Design & Test	Cunniff	1/30/19	2/11/19	7	0%															
3.1.3	State Space Design & Test	Cunniff	2/11/19	2/18/19	7	0%															
3.1.4	Disturbance Testing	Cunniff	2/19/19	2/19/19	0	0%															
3.1.5	Controller Decision	Cunniff	2/20/19	2/20/19	0	0%															
4	Final Preparation/Testing Phase																				
3.X	Spring Break	Cunniff	3/9/19	3/17/19	8	0%															
3.1	Disturbance 2DOF Test/Correct (dry)	Cunniff	2/21/19	2/28/19	7	0%															
3.2	Disturbance 2DOF Test/Correct (wet)	Cunniff	2/28/19	3/7/19	7	0%															
3.2.1	Compile Results & Calculations	Cunniff	3/8/19	3/15/19	7	0%															
3.2.2	Photos	Cunniff	2/25/19	3/12/19	17	0%															
3.3	Poster Completion	Cunniff	4/1/19	4/10/19	9	0%															
3.X	Spring Break	Cunniff	3/9/19	3/17/19	8	0%															

Early in the semester it was determined that the Ardrone 2.0 that was used for research in EW401 was not adequate to fulfill the tasks this project set forth. The process of ordering the Tello drone and completing the same tasks required to operate the Ardrone using a joystick took more time than expected due to the Ardrone having old but comprehensive resources available that allowed it to perform well within a very specific set of circumstances. This required the generation of a new timeline modified to fit the additional task of setting up a new drone with relatively little documentation available as to its operation.

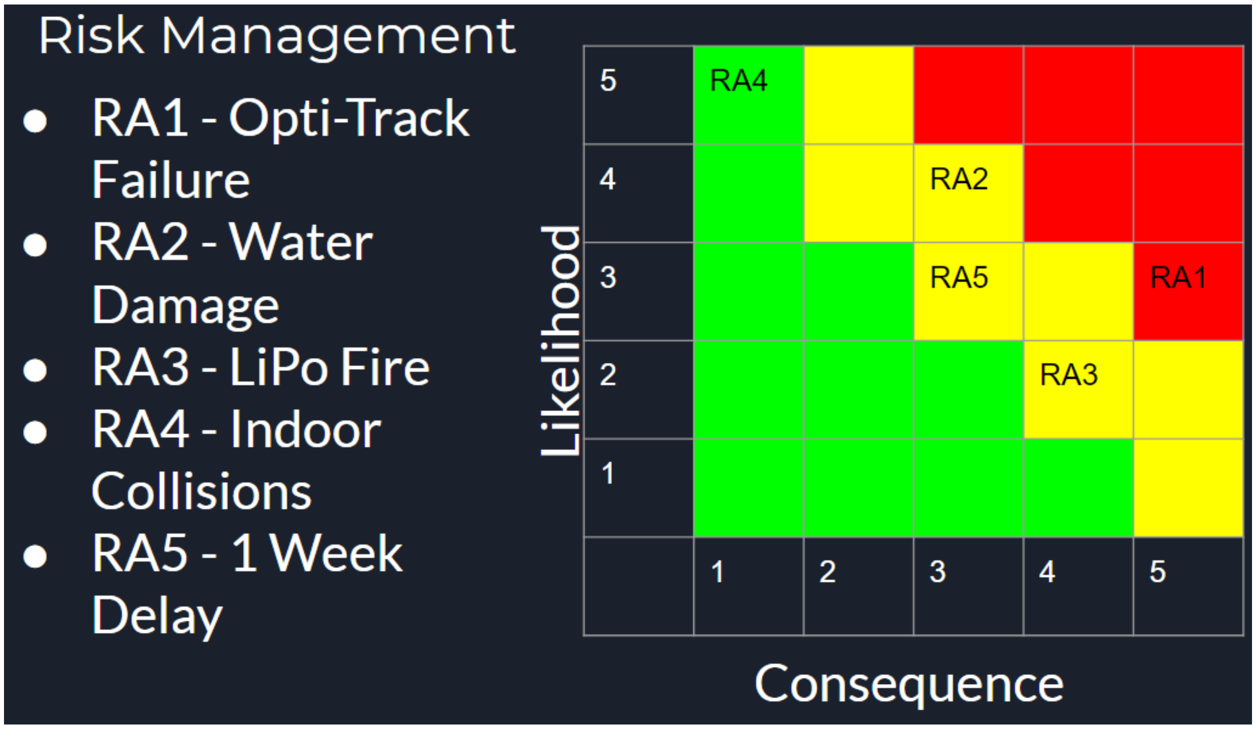
Table 6: Revised Gantt chart



WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
2	Tello Setup Phase					
2.1	Place Order for Tello Drone	Cunniff	1/8/19	1/30/19	22	100%
2.1.1	Verify Functions/Establish Connection	Cunniff	1/8/19	1/15/19	7	100%
2.1.2	Joystick Integration	Cunniff	1/8/19	1/15/19	7	100%
3	1DOF Phase					
3.1	Initial Controller Design	Cunniff	1/30/19	2/22/19	22	100%
3.1.3	Proportional Controller Design & Test	Cunniff	2/11/19	2/18/19	7	100%
3.1.4	Disturbance Testing	Cunniff	2/19/19	2/22/19	3	100%
3.1.5	Controller Decision	Cunniff	2/20/19	2/28/19	8	100%
4	Final Preparation/Testing Phase					
3.X	Spring Break	Cunniff	3/9/19	3/17/19	8	100%
3.2.1	Compile Results & Calculations	Cunniff	3/8/19	3/15/19	7	100%
3.2.2	Photos	Cunniff	2/25/19	3/12/19	17	100%
3.3	Poster Completion	Cunniff	4/1/19	4/10/19	9	100%
4	Capstone Week					
4.1	Final Demonstration	Cunniff	4/10/19	4/15/19	5	100%
4.2	Presentation	Cunniff	4/15/19	4/24/19	9	100%
4.3	Final Report	Cunniff	1/30/19	4/30/19	90	100%

Risk Management

Table 7: Risk Management Matrix



Risk Management

- RA1 - Opti-Track Failure
- RA2 - Water Damage
- RA3 - LiPo Fire
- RA4 - Indoor Collisions
- RA5 - 1 Week Delay

Likelihood					
5	🟩 RA4	🟨	🟥	🟥	🟥
4	🟩	🟨	🟨 RA2	🟥	🟥
3	🟩	🟩	🟨 RA5	🟨	🟥 RA1
2	🟩	🟩	🟩	🟨 RA3	🟨
1	🟩	🟩	🟩	🟩	🟨
	1	2	3	4	5
	Consequence				

An Opti-Track failure would likely have been catastrophic in this lab and would have required attempting to use the mobile setup or the older Vicon motion capture systems in the department. This would have necessitated relearning a new motion capture UI and finding existing ROS nodes that would interpret the data from the different system. Water damage was very avoidable when the project was modified to eliminate phase 2 testing. LiPo fire was a concern, and TSD approved LiPo bags were used to safely contain the batteries while charging or in storage. Indoor collisions rarely resulted in more than a broken propeller which was planned for and easily replaced. The risk that was not able to be avoided in this project was the 1 week project delay. Due to the hardware changes early in the spring semester, the process of acquiring a new drone and getting it operating using relatively little available information added time to the project and required modification to the end objectives. Additionally the process of acquiring a quadrotor is slow even once the funding is available due to the inspection process. The delay caused the project to turn from a controls design project in which various controllers would be implanted and evaluated, to a systems integration project in which a framework was developed in which any controller could be tested.

Discussion and Conclusion

The strength of the final design is the decentralized nature of the existing system, and the fact that the code is not looking for any particular sensor so long as the sensor is sending a standard message type. The benefit of this is that future work on the project including the phase 2 field testing will likely occur outside of the Opti-track lab and not have the benefit of a highly accurate monitoring system. Integration of an additional system such as an IMU, GPS, or other positioning system based in computer vision can feed directly into inputs for the controller with relatively minor changes in the code. Comparing this to using a language such as python without ROS to handle communications, and the process of sending messages that are quickly interpreted by other programs on the system can become tedious and slow. The course of action to connect to the Tello drone to a PC has been documented, and now for any group that wants to approach this project in the future there is a body of work to refer to in order to spend more time working on the control design aspects of the project. The teleoperation of the Tello drone using a joystick has been explored to allow for rapid testing and can run in lieu of the controller as situations determine. The primary critique of this project is that the proportional controller developed to test the controls principles this project did not function properly, and so any future group attempting this project or something similar should have a background in python as well as ROS. Construction of additional libraries within the source files for the executable scripts will be necessary any time a new sensing element is added. This will ensure that whatever controller is developed will work properly when it comes to communication between nodes.

Acknowledgment

Professor Jeremy Dawkins provided a great deal of time explaining ROS and troubleshooting the python scripts produced for or modified to fit the needs of this project. Professor Kutzer provided assistance solving technical difficulties with the Opti-Track system and provided the resources to use the Opti-Track lab independent of instructor supervision. Professor Feemster guided this project from fall semester through the end of the year.

References

[1] Anqi Xu, “anqixu/tello_driver,” GitHub, 06-Nov-2018. [Online]. Available: https://github.com/anqixu/tello_driver. [Accessed: 02-Mar-2019].

[2] Cully, Niewoehner, Bloom, Evertson, Partlow, Speir, and Webb, “Mangrove Forest Survey UAS Pack Mule,” rep.

[3] Jonathan Bohren, “joy Wiki,” ros.org, 13-Mar-2009. [Online]. Available: http://wiki.ros.org/joy. [Accessed: 01-Apr-2019].

[4] M. Watterson and V. Kumar, "Safe receding horizon control for aggressive MAV flight with limited range sensing," 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, 2015, pp. 3235-3240.

[5] Paul Bovbel, “vrpn_client_ros Wiki,” ros.org, 22-Jun-2015. [Online]. Available: http://wiki.ros.org/vrpn_client_ros. [Accessed: 20-Jan-2019].

[6] Portions of this document were adapted from the IEEE Conference template found at http://www.ieee.org/conferences_events/conferences/publishing/templates.html and the ES200 Lab Report Format document.

Appendix

Joy_teleop.launch

```
<?xml version="1.0" ?>
<launch>
  <arg name="joy_dev" default="/dev/input/js0" />

  <arg name="namespace" default="Tello" />

  <node pkg="joy" type="joy_node" name="joy_node">
    <param name="dev" value="$(arg joy_dev)" />
    <param name="deadzone" value="0.2" />
  </node>

  <group ns="$(arg namespace)">
    <node pkg="tello_driver" type="gamepad_marshall_node.py" name="joy_teleop" />
  </group>

</launch>
```

tello_node.launch

```
<?xml version="1.0"?>
<launch>
    <arg name="tello_ip" default="192.168.10.1" />
    <arg name="tello_cmd_server_port" default="8889" />
    <arg name="local_cmd_client_port" default="8890" />
    <arg name="local_vid_server_port" default="6038" />
    <arg name="namespace" default="Tello" />

    <group ns="$(arg namespace)">
        <node pkg="tello_driver" name="Tello" type="tello_driver_node.py" output="screen">
            <param name="local_cmd_client_port" value="$(arg local_cmd_client_port)" />
            <param name="local_vid_server_port" value="$(arg local_vid_server_port)" />
            <param name="tello_ip" value="$(arg tello_ip)" />
            <param name="tello_cmd_server_port" value="$(arg tello_cmd_server_port)" />
            <param name="connect_timeout_sec" value="10.0" />
            <param name="stream_h264_video" value="false" />
        </node>

        <node pkg="image_transport" name="image_compressed" type="republish" args="raw
in:=image_raw compressed out:=image_raw" />
    </group>
</launch>
```

tello_controller.py

```
#!/usr/bin/env python2
import time
import traceback
import tellopy
import rospy
from geometry_msgs.msg import PoseStamped
from sensor_msgs.msg import Joy
from std_msgs.msg import Empty, UInt8, Bool
from geometry_msgs.msg import Twist



def distance(deltax,deltay,deltaz)
        #makes more sense when landing platform is not at (0,0,0)
        deltax = x - 0
        deltay = y - 0
        deltaz = z - 0
        #pub = rospy.Publisher('chatter',
def callback(msg)
        x = msg.pose.position.x
        y = msg.pose.position.y
        z = msg.pose.position.z
        #rospy.loginfo('x: {}, y: {}, z:{}'.format(x, y, z))
def controller(deltax,deltay,deltaz)
        bound = 50
        if deltax <= bound or deltax >= -bound
                pitchcmd = 0.0
        elif deltax < bound
                pitchcmd = 1.0 * deltax
        elif deltax < bound
                pitchcmd = -1.0 * deltax
        if deltay <= bound or deltay >= -bound
                rollcmd = 0.0
        elif deltax < bound
                rollcmd = 1.0
        elif deltax < bound
                rollcmd = -1.0
        if deltax == 0.0 and deltay == 0.0
                vcmd = -0.6

def main():
   rospy.init_node('tello_controller')
   rospy.Subscriber("/vrpn_client_node/tello1/pose",PoseStamped,callback)
   pub = rospy.Publisher('chatter', Twist, queue_size=10)
   rospy.spin()

if __name__ == '__main__':
    main()


def test():
    drone1 = tellopy.Tello(
        local_cmd_client_port=8890,
        local_vid_server_port=6038,
        tello_ip='192.168.10.1',
    )
    try:
        drone1.connect()
        drone1.wait_for_connection(10.0)
        print('Connected to drone')

        # Test raw video
        if True:
            drone1.start_video()
            while True:
                time.sleep(1.0)

        # Test video decoding
        if False:
            import av
            import cv2
            import numpy

            container = av.open(drone1.get_video_stream())
            for frame in container.decode(video=0):
                image = cv2.cvtColor(numpy.array(
                    frame.to_image()), cv2.COLOR_RGB2BGR)
                cv2.imshow('Frame', image)
                key = cv2.waitKey(1)
                if key and chr(key) in ('q', 'Q', 'x', 'X'):
                    break

        # Test flying
        if False:
                #cmd = Twist()
            #cmd.linear.x = self.joy_state.LY
            #cmd.linear.y = self.joy_state.LX
            #cmd.linear.z = self.joy_state.RY
            #cmd.angular.z = self.joy_state.RX
            #self.pub_cmd_out.publish(cmd)
                time.sleep(2.0)
            drone1.takeoff()
            time.sleep(3.5)

            drone1.set_vspeed(1.0)
            time.sleep(0.6)
            drone1.set_vspeed(0.0)

            time.sleep(2.0)

            drone1.set_pitch(pitchcmd)
                time.sleep(0.5)
                drone1.set_yaw(yawcmd)
            drone1.set_pitch(0.0)
            drone1.set_yaw(0.0)

            time.sleep(2.0)

            drone1.flip(1)

            time.sleep(2.0)

            drone1.set_vspeed(-0.5)
            time.sleep(0.6)
            drone1.set_vspeed(0.0)

            time.sleep(1.0)

            drone1.palm_land()
            time.sleep(3.0)
                  #test controller (comment out test flying)
                  if False
                        time.sleep(2.0)
            drone1.takeoff()
            time.sleep(3.5)


    except BaseException:
        traceback.print_exc()
    finally:
        if False:
            drone1.reset_cmd_vel()
            drone1.land()
        drone1.quit()


#if __name__ == '__main__':
#    test()
```

gamepad_marshall_node.py

```
#!/usr/bin/env python2
import rospy
from std_msgs.msg import Empty, UInt8, Bool
from sensor_msgs.msg import Joy
from geometry_msgs.msg import Twist


class GamepadState:
  def __init__(self):
    self.A = False
    self.B = False
    self.X = False
    self.Y = False
    self.Start = False
    self.Select = False
    self.Sync = False
    self.L1 = False
    self.L2 = False
    self.L3 = False
    self.R1 = False
    self.R2 = False
    self.R3 = False
    self.DL = False
    self.DU = False
    self.DR = False
    self.DD = False
    self.LX = 0.  # +: left
    self.LY = 0.  # +: top
    self.RX = 0.  # +: left
    self.RY = 0.  # +: top
    self.LT = 0.  # 1.0: idle, -1.0: depressed
    self.RT = 0.  # 1.0: idle, -1.0: depressed

  def parse_ps3_usb(self, msg):
    if len(msg.buttons) != 12 or len(msg.axes) != 6:
      raise ValueError('Invalid number of buttons (%d) or axes (%d)' % (
        len(msg.buttons), len(msg.axes)))
    self.A = msg.buttons[0]
    self.B = msg.buttons[1]
    self.X = msg.buttons[2]
    self.Y = msg.buttons[3]
    self.L1 = msg.buttons[4]
    self.R1 = msg.buttons[5]
    #self.L2 = msg.buttons[11]
    #self.R2 = msg.axes[7]
    self.Select = msg.buttons[6]
    self.Start = msg.buttons[7]
    self.L3 = msg.buttons[9]
    #self.R3 = msg.buttons[10]
        self.R2 = msg.buttons[10]
        self.DU = msg.buttons[11]
    #self.DD = msg.buttons[14]
    #self.DL = msg.buttons[15]
    #self.DR = msg.buttons[16]
    self.LX = msg.axes[0]
    self.LY = msg.axes[1]
    #self.LT = msg.axes[2]
    self.RY = msg.axes[5]
    self.RX = msg.axes[2]
    #self.RT = msg.axes[5]


def parse_ps3_usb2(self, msg):
    if len(msg.buttons) != 19 or len(msg.axes) != 27:
        raise ValueError('Invalid number of buttons (%d) or axes (%d)' % (
            len(msg.buttons), len(msg.axes)))
    self.Select = msg.buttons[0]
    self.L3 = msg.buttons[1]
    self.R3 = msg.buttons[2]
    self.Start = msg.buttons[3]
    self.DU = msg.buttons[4]
    self.DR = msg.buttons[5]
    self.DD = msg.buttons[6]
    self.DL = msg.buttons[7]
    self.L2 = msg.buttons[8]
    self.R2 = msg.buttons[9]
    self.L1 = msg.buttons[10]
    self.R1 = msg.buttons[11]
    self.Y = msg.buttons[12]
    self.B = msg.buttons[13]
    self.A = msg.buttons[14]
    self.X = msg.buttons[15]
    self.Sync = msg.buttons[16]
    self.LX = msg.axes[0]
    self.LY = msg.axes[1]
    self.LT = msg.axes[12]
    self.RX = msg.axes[2]
    self.RY = msg.axes[3]
    self.RT = msg.axes[13]


def parse(self, msg):
    err = None
    try:
        return self.parse_ps3_usb(msg)
        except ValueError, e:
            err = e
        try:
            return self.parse_ps3_usb2(msg)
        except ValueError, e:
            err = e
        raise err


class GamepadMarshallNode:
    MAX_FLIP_DIR = 7

    def __init__(self):
        # Define parameters
        self.joy_state_prev = GamepadState()
        # if None then not in agent mode, otherwise contains time of latest enable/ping
        self.agent_mode_t = None
        self.flip_dir = 0

        # Start ROS node
        rospy.init_node('gamepad_marshall_node')

        # Load parameters
        self.agent_mode_timeout_sec = rospy.get_param(
            '~agent_mode_timeout_sec', 1.0)

        self.pub_takeoff = rospy.Publisher(
            'takeoff', Empty,  queue_size=1, latch=False)
        self.pub_throw_takeoff = rospy.Publisher(
            'throw_takeoff', Empty,  queue_size=1, latch=False)
        self.pub_land = rospy.Publisher(
            'land', Empty,  queue_size=1, latch=False)
        self.pub_palm_land = rospy.Publisher(
            'palm_land', Empty,  queue_size=1, latch=False)
        self.pub_reset = rospy.Publisher(
            'reset', Empty,  queue_size=1, latch=False)
        self.pub_flattrim = rospy.Publisher(
            'flattrim', Empty,  queue_size=1, latch=False)
        self.pub_flip = rospy.Publisher(
            'flip', UInt8,  queue_size=1, latch=False)
        self.pub_cmd_out = rospy.Publisher(
            'cmd_vel', Twist, queue_size=10, latch=False)
        self.pub_fast_mode = rospy.Publisher(
            'fast_mode', Bool,  queue_size=1, latch=False)
        self.sub_agent_cmd_in = rospy.Subscriber(
            'agent_cmd_vel_in', Twist, self.agent_cmd_cb)
        self.sub_joy = rospy.Subscriber('/joy', Joy, self.joy_cb)
        rospy.loginfo('Gamepad marshall node initialized')
```

```
def agent_cmd_cb(self, msg):
    if self.agent_mode_t is not None:
        # Check for idle timeout
        if (rospy.Time.now() - self.agent_mode_t).to_sec() > self.agent_mode_timeout_sec:
            self.agent_mode_t = None
        else:
            self.pub_cmd_out.publish(msg)

def joy_cb(self, msg):
    self.joy_state = GamepadState()
    self.joy_state.parse(msg)

    # Process emergency stop
    if not self.joy_state_prev.B and self.joy_state.B:
        self.pub_reset.publish()
        #rospy.logwarn('Issued RESET')
        return

    # Process takeoff
    if not self.joy_state_prev.Start and self.joy_state.Start:
        self.pub_takeoff.publish()
        #rospy.logwarn('Issued TAKEOFF')

    # Process throw takeoff
    if not self.joy_state_prev.DU and self.joy_state.DU:
        self.pub_throw_takeoff.publish()
        #rospy.logwarn('Issued THROW_TAKEOFF')

    # Process land
    if not self.joy_state_prev.Select and self.joy_state.Select:
        self.pub_land.publish()
        #rospy.logwarn('Issued LAND')

    # Process palm land
    if not self.joy_state_prev.DD and self.joy_state.DD:
        self.pub_palm_land.publish()
        #rospy.logwarn('Issued PALM_LAND')

    if not self.joy_state_prev.X and self.joy_state.X:
        self.pub_flattrim.publish()
        #rospy.logwarn('Issued FLATTRIM')

    if not self.joy_state_prev.Y and self.joy_state.Y:
        self.pub_flip.publish(self.flip_dir)
        #rospy.logwarn('Issued FLIP %d' % self.flip_dir)
        self.flip_dir += 1
        if self.flip_dir > self.MAX_FLIP_DIR:
            self.flip_dir = 0
        # Update agent bypass mode
        if self.joy_state.L2:
            self.agent_mode_t = rospy.Time.now()
        else:
            self.agent_mode_t = None

            print self.agent_mode_t
        # Manual control mode
        if self.agent_mode_t is None:
            if not self.joy_state_prev.R2 and self.joy_state.R2:
                self.pub_fast_mode.publish(True)
            elif self.joy_state_prev.R2 and not self.joy_state.R2:
                self.pub_fast_mode.publish(False)

            cmd = Twist()
            cmd.linear.x = self.joy_state.LY
            cmd.linear.y = self.joy_state.LX
            cmd.linear.z = self.joy_state.RY
            cmd.angular.z = self.joy_state.RX
            self.pub_cmd_out.publish(cmd)

        # Copy to previous state
        self.joy_state_prev = self.joy_state

    def spin(self):
        rospy.spin()


if __name__ == '__main__':
    try:
        node = GamepadMarshallNode()
        node.spin()
    except rospy.ROSInterruptException:
        pass
```