Autonomous Building Inspection Using Unmanned Aerial Vehicles

Submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering
by
Joseph Adamson
June 18, 2020
Baskin School of Engineering
University of California, Santa Cruz

The Master's Project of Joseph Adamson is approved:

Gabriel Elkaim Electrical and Computer Engineering	Date
Patrick Mantey Electrical and Computer Engineering	Date
Quentin Williams Acting Vice Provost and Dean of Graduate Studies	Date

Contents

1 Background and Motivation **2**

2 Proposed Solution **3**
2.1 Machine Learning . . . 3
2.2 Unmanned Aerial Vehicle . . . 3

3 Payload Design **4**
3.1 Computing Power . . . 4
3.2 Cameras and Lighting . . . 4
3.3 Distance Sensors . . . 5
3.4 Pixhawk Flight Controller . . . 5

4 Software Design **6**
4.1 Localization Solution . . . 6
4.2 Robot Operating System . . . 8
4.3 Nodes, Architecture, and Setup . . . 10
4.4 Pixhawk Firmware (PX4) and Ground Control Station . . . 16

5 Handheld Prototype Platform **20**
5.1 Construction . . . 21

6 UAV Platform **22**
6.1 Standard UAV Components . . . 22
6.2 Payload Packaging . . . 24
6.3 System Operation . . . 27

7 Testing and Results **28**
7.1 Prototype Tests . . . 28
7.2 Flight Tests . . . 32

8 Future Work **35**

9 Conclusion **36**

Appendices **39**

A PX4 Guides **39**

B ROS Installation Steps **40**

1 Background and Motivation

NASA research centers are home to a wide range of industrial test facilities critical to the success of their missions. Among these facilities is the National Full-Scale Aeronautics Complex (NFAC) which houses some of the largest wind tunnels in the world. Wind tunnels are typically much smaller, dealing with scaled-down models; however, with its 40' x 80' and 80' x 120' test sections the complex allows for the testing and validation of aircraft models which are, true to the name, full-scale. The tunnel is driven by 6 astoundingly huge fans, each 40 feet in diameter and requiring an onsite power plant to start the motors before switching to the power grid. To support this scale of testing, the NFAC is the size of several city blocks, with large swaths of warehouse-like empty space often reaching heights of 10 stories (see figure 1 below.)

Given that the structure is composed largely of panels fastened to a metal skeleton, there can be several types of wear/defect including rusted/missing rivets and cracked panels. These can be due to mishaps, stress, or *Foreign Object Damage*.[1] These issues are exacerbated by its age; initial construction began as far back as the 1940s.

Inspection and maintenance of such a large facility is cumbersome. Current inspection techniques involve scaffolding, rope-climbers, telephoto and by-eye inspection. This results in many work-hours of skilled and dangerous labor required for ensuring the NFAC is in a workable state. Thus, there is a strong motivation for automating or aiding the inspection process of the NFAC and other facilities. Unmanned Aerial Vehicles (UAV's, or simply 'drones') are strong candidates for such automation, especially as onboard computing power grows ever stronger, smaller, and more efficient.

Figure 1: Aerial view of the NFAC at NASA Ames[1]

[1]Consider the destructive force of a penny or sand traveling at 120mph

2 Proposed Solution

David Murakami, an aerospace engineer in the Fluid Mechanics Laboratory (Code AOX) at NASA Ames Research Center, submitted a grant proposal exploring the possibilities of automating the NFAC inspection process. Machine learning and drones are two technologies currently seeing steady rises in popularity and ability; as such, combining them might yield strong automation technology. Each half of the proposed project is discussed below.

Clearly the administration agreed, as the proposal was accepted and given funding through NASA's Internal Research and Development (IRAD) program. Indeed, there is strong interest in this technology across many divisions, given that NASA holds many large mission-critical facilities (for example, other wind tunnels, testing facilities, and launch silos.) Furthermore, the US Air Force is also contributing to this project, as the NFAC is currently under their jurisdiction.

Dr. Murakami included provisions in his proposal for two graduate-level interns to assist with the project, at which point I became involved as a team member on this collaboration between UCSC and NASA.

2.1 Machine Learning

In recent years, machine learning techniques have enjoyed a surge in popularity fueled by modern processor technology, which brings viability to algorithms previously requiring excessive training times and computational power. This branch of computer science has myriad applications for such tasks as facial recognition, object detection, and other arbitrary function-fitting tasks. As such, there is strong potential for applying such techniques to the detection of defects in the NFAC structure using high-resolution diagnostic images.

The student in charge of this portion, Nick Califano (Computer Science MS, University of South Florida,) has demonstrated strong results using image classification, advanced computer vision, 3D photogrammetry, and other techniques for determining defects and their positions within the tunnel. Early proof of concept runs were taken with a handheld camera rig and provided the initial dataset for seeking out cracked panels and missing fasteners.

2.2 Unmanned Aerial Vehicle

The goal of this project was to develop a drone and payload which would fly inside the wind tunnel to capture the necessary diagnostic images. Emphasis was placed upon development of a prototype platform for testing localization and imaging viability. UAV systems aren't limited to the ground and require far less safety equipment and work-hours as compared with manual inspection. My role in this project was that of developing such an autonomous system. The payload primarily includes the high-resolution camera necessary for taking the pictures, but also must incorporate processing power, lighting, and other aspects pertinent to autonomous flight. The ultimate goal is to fly a 'lawnmower' grid pattern up and down the various walls while taking images at fixed points; the images should be taken at a distance of 1 meter from the wall, which is close enough for our focused camera to capture relevant details.

The NFAC, and indoor flying in general, presents a great deal of challenges with regard to autonomous flying vehicles. Among these issues, one primary concern for such flying is the lack of GPS. A discussion of these challenges, and the solution we are using, can be found in section 4.1.

The two core components of such a robotic system, hardware and software, were developed during the course of the project. Presented here is a description of those aspects, usage scenarios for the drone, as well as a discussion of the test results and future work.

3 Payload Design

3.1 Computing Power

The Intel NUC line is a series of small and powerful computing units. Like other units becoming common in the market, they are full computers running an operating system, unlike classical microcontrollers. Furthermore, this class of mini-pc is much more muscular than boards such as the Raspberry Pi, and boasts specifications equivalent to a strong laptop. This is achieved while retaining a form factor suited for small robotic systems (with a square case measuring roughly 4.5" to a side.)

Since the payload needs to perform a variety of intensive calculations (12MP image capture, computer vision, localization, communication, etc) the NUC was a natural choice as the brains of the operation. Our model is configured with a 2.2GHz dual-core i5 processor and 16GB of RAM. It also provides the necessary IO options for the payload (primarily, 4 USB 3.0 ports.) This computing unit was used for early proof of concept work and thus allowed us to move straight to payload design from the concept phase.

3.2 Cameras and Lighting

Basler daA1280-54uc ('Slam Cam')

This 1 megapixel global-shutter camera was used during proof-of-concept work to capture images of an interior portion of the NFAC, and was later repurposed in the final payload as a localization sensor. It is capable of capturing images at 54 frames per second. A lens attachment allows adjustment of the focal length and aperture, which improves both field of view and depth of field.

Basler acA4112-20uc ('Wall Cam')

This 12 megapixel camera is capable of capturing images at 20 frames per second and is used as the high-resolution camera to take the diagnostic images of the walls. The lens for this camera is fixed focal length, and focused at 1 meter. Therefore, the drone should keep a distance of 1 meter between itself and the walls.

Both cameras, manufactured by Basler, feature GPIO pins for interacting with other hardware, and communicate with the NUC via USB 3.0. A software suite provided by

Basler allows writing settings to the firmware such as exposure time and gain, but settings can also be written in custom software as described in section 4.3.

Lighting

Since the NFAC is a dark environment, the system needs to carry its own onboard lighting. Furthermore, the light needs to be as intense as possible so that diagnostic images can be taken with a very low exposure time, resulting in minimal motion blur. A high-intensity LED array was constructed, including driver circuitry, in order to provide the necessary light. Following a few troubled attempts to drive the LEDs, a working driver board was constructed and interfaced to the cameras. The GPIO of the cameras is configured to send a high logic-level pulse whenever the shutter is open; that signal triggers the enable pin of the LED driver, resulting in synchronized shots without damaging the lights with continuous operation.

3.3 Distance Sensors

The sensor package includes two distance sensors, which allow the system to sense in absolute units (ie, meters) to scale and augment the Orb Slam2 localization discussed in section 4.1. By using a forward-looking rangefinder, the system can make adjustments to its path in order to maintain the desired 1-meter distance from the wall. A downward-looking rangefinder can augment the localization algorithm's height estimate and provides much less variance than, for example, a barometer.

- Garmin lidarlite v3hp: This rangefinder unit can sense distances up to 40 meters, and is used as the downward-facing sensor.
- Sparkfun VL53L1x module: This unit is a breakout board for the VL53L1x chip, which is a short-range laser ranging module. It allows millimeter-precision up to a maximum range of 4 meters, and serves as the forward-looking sensor.

Both modules communicate over an I2C connection, which is read using an Arduino. The Arduino is a popular microcontroller board which acts simply to interface the sensors to the Intel NUC and performs very little processing. The short-range sensor operates at 3.3 volts, while the Arduino and long-range sensor operate at 5 volts. Therefore, a bidirectional level shifter was used in order to allow safe communication via I2C. The Arduino is connected to the NUC via USB, and takes sensor readings at set intervals as dictated by the ROS software.

3.4 Pixhawk Flight Controller

Though not strictly part of the custom payload, the flight controller is a vital portion of aerial systems and requires some discussion. Flight controllers are computing units responsible for receiving commands, reading onboard sensors, and running feedback control calculations in order to drive the actuators and maintain stable control of the vehicle. There are many flight

controllers on the market; of these, the Pixhawk was chosen for this project. The Pixhawk and associated firmware are open source and widely supported by a huge community.

Like other flight controllers, the Pixhawk is a specialized microcontroller with several serial ports and pwm outputs suitable for transmitting/receiving radio and commanding 4 or more propeller motors. It contains internal inertial sensors (gyroscope and accelerometer) as well as a magnetometer. Additionally, there is ADC functionality which allows battery voltage to be monitored. A full wiring guide is given in [2]

The Pixhawk is connected to the NUC with a serial-to-usb (*FTDI*) converter using its TELEM2 port, exposing its messaging system (mavlink) to the computer.

4 Software Design

Robotic systems must properly combine software and hardware in a manner that lets them interface with the outside world semi or fully autonomously. Robotic and embedded software must deal directly with the hardware, which results in several layers necessary for programming such a system. For example, a typical setup will have, at minimum, the software layers:

- Low-level IO drivers are required for reading sensors, implementing communication protocols, controlling actuators (motor circuitry) and other hardware functions related to physically interfacing with the world
- Mid-level feedback controllers use the drivers to gather/estimate state information and enact control schemes accordingly (PID, full-state, or other feedback control methods) to drive the system state to a desired setpoint.
- High-level path planners generate desired state trajectories (missions, obstacle avoidance, etc) which use the controllers to execute high-level behavior related to the system's purpose

It is imperative to have good feedback regarding the robot's position and orientation in space. This feedback, otherwise known as *localization* or *pose estimation*, is a core element of control; after all, you need to know where you are in order to get where you want to go! As flying robotic systems, drones are no exception. Initial research was performed regarding setup and usage of such systems. For example, some typical autonomous mission applications include aerial photography, agricultural surveillance, and search-and-rescue operations. The standard localization toolkit for such applications is a combination of GPS, inertial, barometric, and magnetometer feedback. However, these applications are all performed outdoors where access to GPS is unhindered; this is not the case in our target environment as discussed below.

4.1 Localization Solution

The environment inside the NFAC has several barriers to localization. It is largely constructed of a metal skeleton, with panels affixed to the inside. These panels consist of a

concrete/asbestos mixture[2]; as with most indoor environments, this results in a workspace with no GPS reception and poor magnetometer performance (the skeleton behaves rather like a Faraday cage). Inconveniently, GPS and magnetometer are central components of typical UAV configurations as discussed above. Indoor localization systems do exist, such as ultrasonic beacons and motion capture (ie, *Vicon.*) However, given that the NFAC is a USAF-controlled facility with poisonous walls, it would be difficult to install such systems or otherwise modify the facility without a lengthy or heavily scrutinized approval process.

Following initial research and discussions/conceptual collaborations with other research groups at Ames Research Center, we discovered the Orb Slam 2 algorithm as a localization solution. Simultaneous Localization and Mapping (SLAM) is an important class of algorithms in which a robotic system builds a map of its surroundings while estimating its position within those surroundings. This is commonly done with spinning LiDAR sensors or visual fiducials. The nature of the NFAC, however, is that it is *huge and empty.* Maintaining a fix with sensors which can properly range at an empty space this large would require equipment which is either extremely heavy or extremely expensive.

The solution selected for this project was Orb Slam 2, a library which implements a Visual-SLAM algorithm using a camera stream and image recognition in order to calculate the pose estimate [3] [4]. It relies upon the well-known OpenCV library, using feature detection known as ORB in combination with SLAM techniques to generate and maintain a point cloud map of the environment (see figure 4 below.) The point cloud map can be saved and loaded for later use or processing if desired; however, the decision was made to operate the library with a fresh map each run. Below is an example screenshot of detected features and calculated output running in our office:

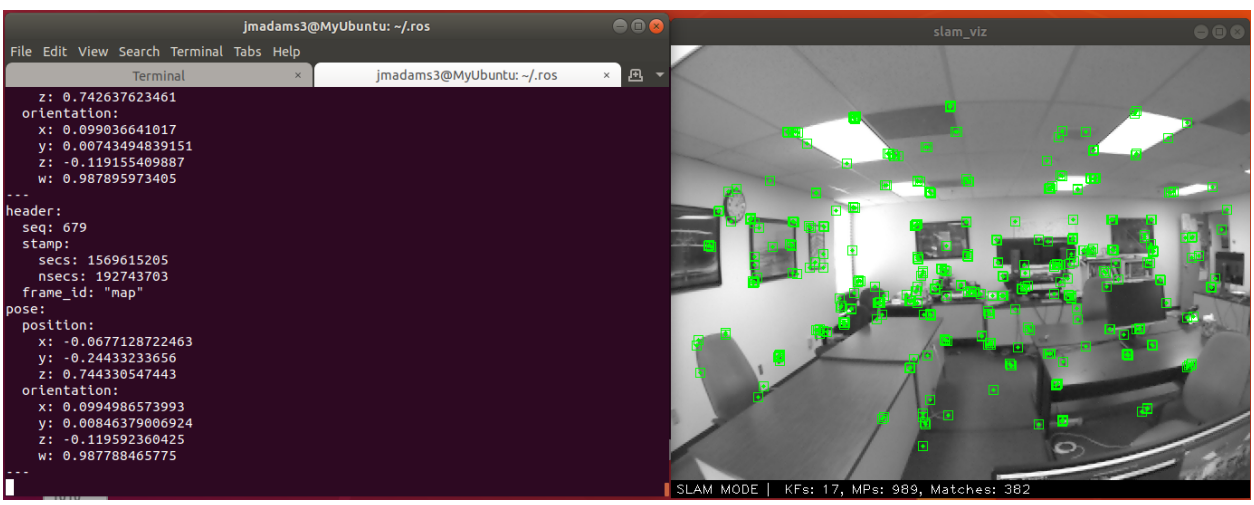


Figure 2: Orb Slam 2 output and detected features running within ROS

Intrinsic Camera Parameters

One critical aspect of Orb Slam's proper functioning is camera calibration. Classical computer vision utilizes mathematical models to map points in space to points on pixel sensors

[2]In addition to typical training courses, mandatory training on poison control and asbestos was completed

as a matrix transformation. These matrices, otherwise known as *intrinsic parameters,* are functions of the camera's physical configuration and describe such aspects as lens distortion and focal length. The intrinsic matrix is of particular use for 3D image reconstruction[5] (see below) and a related distortion vector is used for image rectification, ie 'undoing' distortions caused by lens shape [6]. Without a properly calibrated camera matrix, the image recognition will fail to detect features, as the same features will be differently distorted depending on camera angles. As discussed below, the software architecture chosen for this project provides a simple calibration node as well as information-bearing filetypes for easily loading calibration matrices to Orb Slam 2 or any other program which may need it.
The library works quite well if the camera is calibrated and the system properly tuned. However, some bug fixes were required since the system had difficulty initializing under normal conditions. More information regarding this and other setup are discussed in Appendix B.

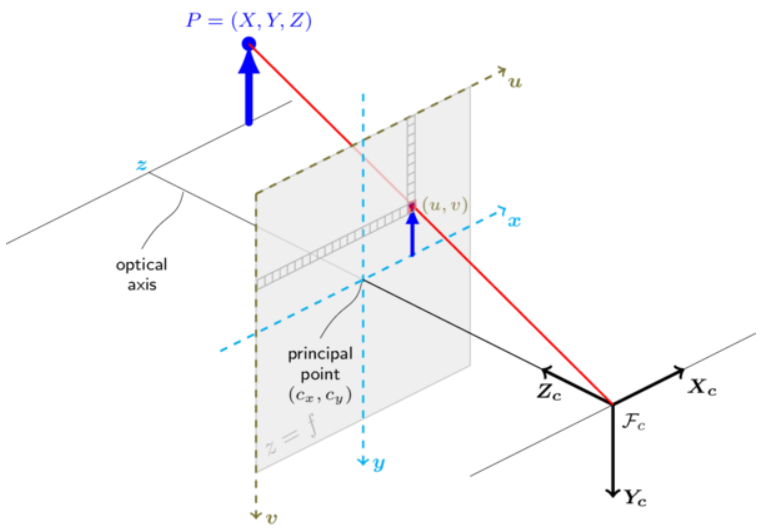


Figure 3: Camera model and 3D reconstruction diagram utilized by SLAM and other triangulating computer vision algorithms. Calibration matrices move between 2D and 3D, while Orb Slam 2 estimates pose at the camera lens[5]

4.2 Robot Operating System

Robot Operating System (hereafter referred to as *ROS*) is an open-source middleware layer facilitating communication between many programs running at once. These executables, known as as *nodes*, are connected within the ROS ecosystem using a standardized set of *messages* which are structural data types specified either by the programmer or ROS core libraries.

Nodes communicate on a publisher/subscriber model (rather like an online message board) whereby channels called *topics* are written to/read from by any node as long as they use that topic's message type. As a basic example, there could be one more more controller nodes publishing to a command topic, with a servomotor node listening and executing any commands written to the topic (thereby connecting software to hardware). This forms a graph structure of nodes and topics depicting information flow.

The publish/subscribe model allows for nodes written in many languages to communicate as long as they provide an implementation of the core ROS API and the required message types. It primarily uses C++ and Python, though many other implementations exist. Communication in this framework is extremely modular; nodes are interchangeable black-boxes and can be individually debugged/tested. The only requirement is that they are configured to speak the same message type.

With regards to this project, a distinct advantage to this modularity is the ability to 'drop in' a new localization node. For example, some testing facilities will include a Vicon system. Configuring the system to take positioning data from a Vicon node instead of an Orb Slam node is trivial; ROS's launcher system utilizes specialized XML to run nodes and describe/remap the connections between them.

In addition to XML, ROS typically uses the recursively-named YAML ("YAML Ain't Markup Language") document format to read and store node settings. YAML is a useful way to incorporate device settings and parameters as key-value pairs since there is much support provided for parsing the files. Generally, a node will parse its specified `.yaml` file upon initialization and register the values in the parameter server (a central repository of system parameters) for itself and other nodes to read.

Environment Setup

The ROS system is linux-based and run in Ubuntu 18.04 (ROS's officially supported distribution) on an Intel NUC. Further discussion of the NUC, which provides the necessary processing power, can be found in part 3.

Due to security restrictions in the AOX branch's protocol, the installation of Ubuntu was run within a virtual machine using Oracle's VirtualBox. Apart from being within a VM, the installation was otherwise typical as per the Ubuntu guides. Future work may entail transitioning the system to run in the relatively recent Windows Subsystem for Linux or researching NASA-approved linux distributions which can run a source-built ROS installation.

A full installation of ROS was performed according to the official documentation. Working in a linux environment for the duration of the project brought much more familiarity with the design mentalities, file structure, workflow, command line, build system (ie, CMAKE) and many other aspects of developing within linux.

Along the way, a list of steps for the drone's software setup was compiled, and is given in Appendix B. The guide includes the core ROS installation, installation of the required packages, dependencies within those packages, and various tweaks/bug fixes performed throughout the project. The official ROS documentation[7] is incredibly useful, including in-depth installation instructions, tutorials, and references.

ROS Message Types Relevant to Project

ROS messages are standardized structs which encapsulate simple and compound data types (including other message types) as named fields. The messages are language independent, and in fact include tools to compile them for any language. Nodes then simply need to include their language's compiled messages in conjunction with their ROS implementation

to read and publish messages. For the implementation of this project, the following standard message types were used:

- String: A string
- Header: Provides a timestamp and sequence number. Most message types include a header for synchronization purposes
- Quaternion: 4 float values giving the w,x,y,z components of a quaternion
- Point: 3 float values giving a 3D position in space
- PoseStamped: A compound message consisting of a Header, Quaternion, and Point message. Fully describes the 6-DOF position of the system
- Image: An image serialized for optimal transport across topics. Often accompanied by a camera_parameters message for rectification and other processing.

4.3 Nodes, Architecture, and Setup

The node architecture used for the UAV is a mixture of self-written and open-source nodes pulled from github. This eliminates the need to write many of the hardware drivers from scratch; often, these open source nodes are wrappers[3] for existing APIs. The custom-written nodes form the basis of the payload's functionality by localizing, controlling, and otherwise unifying the hardware. A full description of each node is given below, with setup/installation steps given in appendix B.

Open Source Packages

Basler Camera Drivers [8]

The Pylon camera suite is a software development kit (SDK) provided by Basler, including drivers and other library functions. These libraries allow software control over the cameras, exposing such commands as image capture and setting parameters. This node implements the camera as a ROS system by using the Pylon SDK to open connections to the hardware, request images at a set frame rate, and publish them out onto an image topic. A `.yaml` file is used to set parameters such as exposure time, resolution, and framerate.

Since both cameras in the payload are made by Basler, it is simple to run a driver node for each by supplying separate `.yaml` files and naming them descriptively; for example the high-resolution camera driver node was titled 'wallCamera' and the corresponding image topic in ROS is wallCamera/image_raw, where the node publishes Image messages.

Unfortunately, this node proved to be rather finicky, as there exist multiple versions of the node produced by different authors, as well as highly unclear framerate issues caused by the default settings. Furthermore, initial versions of the node did not fully implement the Pylon API, which required workarounds in order to manipulate lesser-used functions such

[3] shell code 'wrapped' around existing code to let it interface with another system, usually by translating function calls

as GPIO setup. The resulting debug time was rather steep, but it ultimately proved to be a reliable and extensible node.

Orb Slam 2 ROS [9]

This node is a ROS implementation wrapping the Orb Slam 2 library described in section 4.1. Though the original Orb Slam 2 repo contains a ROS node, this implementation is better integrated and trims some fat from the package. It brings localization into the ROS ecosystem by subscribing to the image topic (output from the camera driver above) and outputting the resultant PoseStamped once it obtains a fix on its position. Additionally, it outputs a debug image containing all recognized features (see grayscale output in figure 2)

This node can optionally publish the point cloud generated by the algorithm, though this was disabled in order to preserve bandwidth. Regardless, a sample of the point cloud output displayed within the ROS system is given below. Note the distinct room/hallway structure and calculated pose within the map:

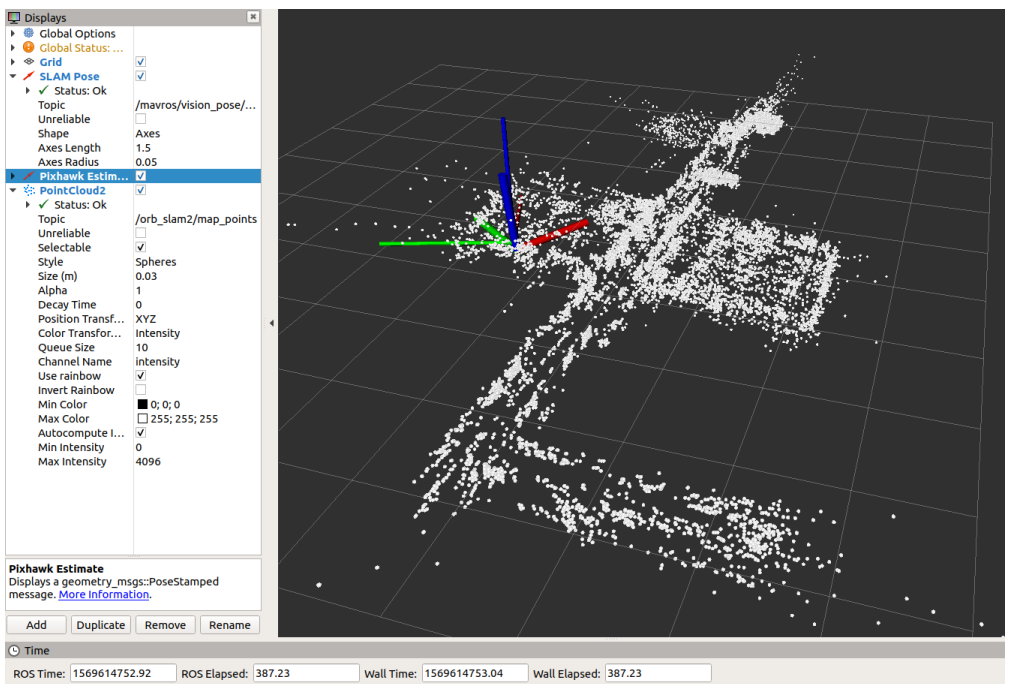


Figure 4: Point Cloud map generated while mapping the lab. Coffee can be found in the upper-right room.

ROS Image Pipeline [10]

ROS comes with many standard packages. One of these is the image pipeline, which provides several utilities for image processing and other perception-related tasks. They are closely tied to the OpenCV libraries and are often wrappers or type converters which give access to OpenCV within the ROS architecture. Two of these utility nodes were of particular use for the project:

- image_view node: subscribes to any image topic and displays it to the screen. Used for viewing raw camera output as well as viewing the debug image from Orb Slam 2. Note that it was necessary to use an older version of the image_view node as the newer version proved incompatible with the older output format used by the SLAM node.

- camera_calibration node: uses OpenCV tools to provide a quick and easy method for generating calibration matrices for the camera. By printing a checkerboard pattern and moving it around the full view of the camera, one can generate the intrinsic camera matrices required by Orb Slam 2 for image rectification.

mavros [11]

Mavros is a robust and functional node and designed to interface ROS with any vehicle speaking the mavlink protocol. Essentially, it translates ROS and Mavlink messages bidirectionally between the NUC and Pixhawk. Several custom ROS messages are provided to aid this interaction. Upon launch, the node opens a serial connection to the Pixhawk using the telem2 port via ftdi as discussed in section 3.4.

Once the serial line is open, several topics are exposed to the rest of the system allowing whatever interaction the application needs. In this setup, the NUC is referred to as the *companion computer* as it utilizes Mavros to assume high-level control and estimation which are impossible on Pixhawk hardware. Among the many topics available via mavros, the following pertain particularly to this setup:

- mavros/RC/in: Publishes the state of the RC transmitter allowing extended remote control. For example, other nodes can subscribe to this in order to respond to changes in knob states

- mavros/vision_position/pose: PoseStamped topic for *sending* localization estimates from SLAM or other system into the pixhawk

- mavros/local_position/pose: PoseStamped topic for *reading* localization estimates from Pixhawk's internal filters

- mavros/attitude: Quaternion topic broadcasting the calculated attitude of the UAV. Note this topic is generally equivalent to the Quaternion element of the local_position/Pose messages.

Custom Packages (Written using C++ and Python)

Quaternion Operations

Quaternions are mathematical constructs commonly used to represent vehicle attitude, and can perform the same functions as rotation matrices with much less computational cost. A quaternion takes the form $w + xi + yj + zk$ and can be thought of as an extension of complex numbers in a higher-dimensional space. They are highly computationally efficient representations of roll-pitch-yaw and serve as the de-facto standard for attitude representation.

A supplementary header file was written in order to perform the necessary quaternion operations without the need for casting or translating between message types. The header deals directly in ROS message structs and is intended to be a quick and easy set of functions for use in the ROS environment. The lightweight file can be used rather than including dedicated libraries such as Eigen, which function on their own data type definitions and require conversion to work with ROS. It implements common quaternion operations including but not limited to:

- Multiplication: Performs the operation $R = P * Q$ such that:
  - R.w = P.w*Q.w - P.x*Q.x - P.y*Q.y - P.z*Q.z;
  - R.x = P.w*Q.x + P.x*Q.w + P.y*Q.z - P.z*Q.y;
  - R.y = P.w*Q.y - P.x*Q.z + P.y*Q.w + P.z*Q.x;
  - R.z = P.w*Q.z + P.x*Q.y - P.y*Q.x + P.z*Q.w;

  Pre-multiplying a quaternion Q by P (the result being PQ) is equivalent to applying the rotation represented by P before any later attitude calculations. Conversely, post-multiplying applies the rotation after any previous calculation.

- Inversion: Generates the multiplicative inverse Q^{-1} of a quaternion Q such that $Q * Q^{-1} = \{1, 0, 0, 0\}$ which is the identity quaternion; it simply reverses the sign of the i, j, k components.

- Conjugation (Point Rotation): Performs the operation $Q * P * Q^{-1}$ ("conjugating P by Q"). This operation rotates P by Q. Its primary use is that when P is constructed as a point (p1,p2,p3) such that P $= 0 + p1 * i + p2 * j + p3 * k$ then conjugating P by Q is equivalent to applying the rotation Q to the point (p1,p2,p3) without needing a rotation matrix.

Frame Rotation

The camera system alone contains no inertial reference, and so Orb Slam 2 initializes itself in an arbitrary fashion. In other words, its output coordinates are given with respect to the first orientation in which it finds a fix, which may not be aligned with a reference frame in which gravity points straight down. This is problematic for a number of reasons. Since the px4 position filters are trying to fuse these position measurements, misaligned reference frames cause instability due to the impossiblity of the estimate converging. Even if this weren't the case, navigating with respect to a misaligned reference frame would spell disaster as it would not be working in the real world. Consider commanding an upward movement, but 'up' in this frame could mean straight into the wall!

The solution applied in this project is to rotate the misaligned frame into an inertial frame by pulling the attitude reading from the pixhawk (via mavros) and generating a correction factor, which is applied to all incoming measurements (Q_{SLAM}). This is done using quaternion pre-multiplication as discussed above:

$Q_{correction} = Q_{imu}^{-1}$; and

$Q_{final} = Q_{correction} * Q_{SLAM}$

Where Q_{imu} is calculated by the flight controller's complementary filter (discussed below in section 4.4.1), and is saved at the moment the SLAM system initializes itself. The result is a corrected attitude reading. The position measurements are also rotated by this correction factor using the conjugation operation, ensuring that readings are in the correct reference frame.

Though it is possible to modify the Orb Slam library to accept an initialized position, this would require some modifications (subscribing the node to orientation topics and converting internal reference frames) and it was instead decided to simply use this intermediary node to apply the rotations. Furthermore, writing a rotation node allows for future re-use if any other frame adjustments are needed.

Frame Scaling/Homing

Since we are only using one camera, output poses from the SLAM algorithm are not scaled to real-world coordinates. This is a limitation of monocular sensing, as the system has no reference for how far a meter is, for example. This can be solved using binocular (stereo) vision since one could stipulate how far apart each camera is, and use that for the triangulation (Orb Slam2 is already equipped to do this type of sensing); however, this requires double hardware and twice the processing power.

Instead, we solve this by using the onboard distance sensors, which measure in absolute units. The node was written to combine distance data from the sensors with distance data from the SLAM output. This is achieved by moving the payload a set distance, recording those distances, and calculating an appropriate scale factor with the formula:

$\Delta D_{real} = SF * \Delta D_{slam} \implies SF = \frac{\Delta D_{real}}{\Delta D_{slam}}$

The node operates in two modes, selected by the remote control. In scaling mode, the node checks every 3 seconds for a change in height (via distance sensor). If a sufficient height change is detected, the scale factor is calculated as above using the recorded change and incoming SLAM positions. After deactivating scaling mode, the node saves its home position and multiplies incoming positions by the scale factor, which converts them into absolute units (meters.) Lastly, the node subtracts off its home position and publishes them back out on a new corrected topic. It is also possible to save a 'home quaternion' which allows the user to designate any arbitrary position as the home orientation using similar techniques to the quaternion correction discussed above.

Between the frame rotation and scaling operations, the net result is a user-homed, local-frame attitude/position measurement given with respect to an absolutely scaled, gravity-down frame of reference. Thus, localization!

Setpoint Handler

This node implements path-planning by reading setpoints from a file which is generated by a supplementary python script. The script was written to create the desired 'lawnmower' pattern, but any type of trajectory planning can be used as long as it generates a list of setpoints for the node to read (again, the modularity of ROS shines here.) The current setpoint is published to the mavros node in order to command the system to that setpoint. The setpoint handler also subscribes to the drone position topic and performs comparisons in order to determine when the commanded and actual positions are within a defined threshold. When that condition is satisfied, a photo is taken and the next setpoint is read from the file and published.

Arduino Node

The onboard Arduino, discussed in section 3, serves to serialize the sensor data coming in from the payload package, which uses I2C sensors. It was written to to wrap the Arduino and allow that data into ROS system by pinging the Arduino (sending a char) at a fixed rate and publishing the response out to a topic. It does this by opening a serial connection using linux's termios mechanism and the Arduino's onboard FTDI chip.

The Arduino firmware is written with a server-like architecture to respond to pings from the ROS node by reading and sending sensor data. It runs a loop which reads incoming characters from the serial line and responds via a large switch statement. This node/firmware combination was written to be extensible and allows further interaction with the Arduino (via pinging with different chars) but the commands currently implemented are:

- Sensor Reading: Requests readings from both distance sensors on the I2C bus and sends them back to the NUC. A moving average filter is applied before sending the data. A 1-second watchdog timer was used to prevent freezing, as occasionally the serial line would glitch and fail to send commands[4].
- Status LED: An RGB LED was attached to the Arduino. Other nodes in the system may request a color change by posting to its topic and the Arduino will in turn activate that color
- GPIO Toggle: Any GPIO pin can be toggled if future aspects of the system require it.

Further discussion of the sensor hardware can be found in section 3.

Attitude Converter/Debugger

Although quaternions are computationally efficient and the standard depiction of attitude, they aren't at all human-readable. Infamously so! As such, a node was written which subscribes to each PoseStamped topic in the system, and consolidates/converts each quaternion to roll-pitch-yaw using the quaternion header written for earlier nodes. The node also extracts and displays the position data so that they can easily be debugged. The converted

[4]Setting up non-blocking serial behavior in linux is nontrivial, and these glitches were likely due to my inexperience as such

poses include raw SLAM, rotated/scaled SLAM, and pixhawk filtered. In addition to sanity checking the various attitude readings, this is also a useful tool for watching the pixhawk position filters for instability and providing system operators with quick feedback regarding attitude calculations within the system.

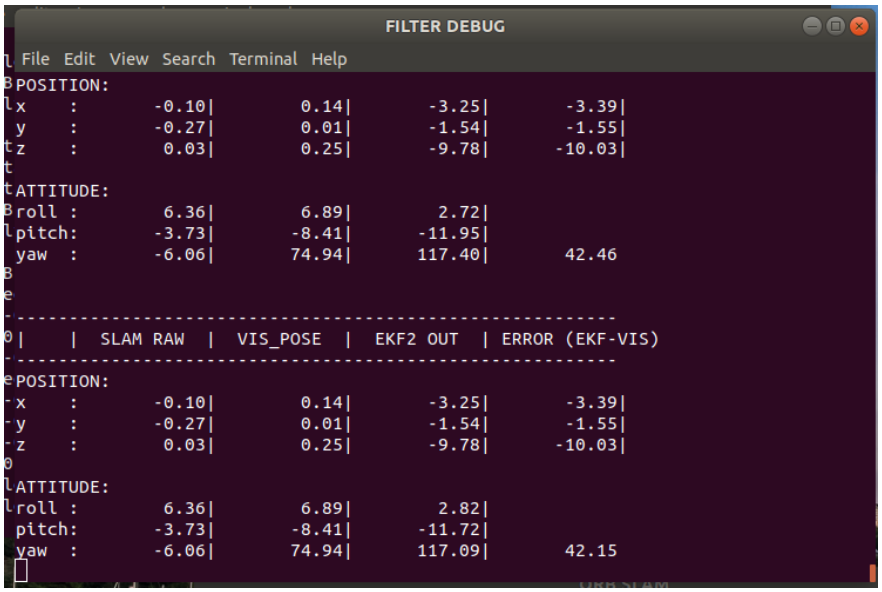


Figure 5: Example output of all quaternions after conversion to RPY

Big Picture: Node Architecture Graph

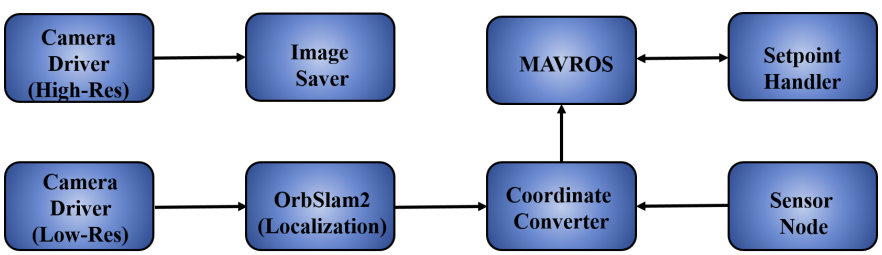


Figure 6: Graph structure demonstrating information flow between ROS nodes

4.4 Pixhawk Firmware (PX4) and Ground Control Station

The pixhawk has a few options for firmware, from which PX4 was selected. This firmware deals with all internal workings of the flight controller. Whereas early remote-controlled airplane setups tied the handheld joystick-based radio controllers (transmitters) directly to the servomotor outputs, modern flight controllers typically instead listen to the transmitter commands and algorithmically interpret them in relation to different control schemes.

One such example for drones is position control, wherein moving an analog stick simply moves the drone's position; the underlying control scheme which requires roll rate control, thrust mixing, and attitude estimation, is abstracted from the user (detail below.) This

functionality is exposed to us for programmatic interaction; in other words, we can write programs to call PX4's internal controllers to set attitude, attitude rates, raw thrust, position, and other types of control. We therefore utilize it for this project to command positions via setpoint stream. This interaction is handled largely by the mavros node and the wider ROS system described above, with the system placed into OFFBOARD mode so that the position control can be linked to the automated setpoint stream.

The PX4 codebase is an entire ecosystem by itself, containing internal messaging systems, modules, a shell, device drivers, and so on. This complexity can be treated mostly as a black box, especially with the mavros wrapper; however one must still understand certain aspects of its workings for dealing with system configuration, parameters, and hardware interaction. Thankfully, the PX4 dev team provides extremely helpful documentation for the entire system and all its capabilities in the form of a user guide[2] and developer guide[12]. Additionally, being an open-source project, the source code is always available for perusal or modification. Subsections of particular interest in the documentation are listed in appendix A.

4.4.1 Estimation and Control

PX4 provides a full featured Estimation and Control Library (ECL). As shown below in figure 7, it uses a cascaded Proportional-PID to implement position controllers for quadcopters. The diagram outlines the control scheme for one axis. Reading the diagram right-to-left, it can be seen that the system must first control roll rates, which is achieved using the PID loop. The output of this loop is the required motor mixing to generate the thrust necessary for controlling both the requested roll rate and altitude. The input is a proportional gain applied to the reference position, unless an external controller is directly controlling the roll rates.

The PX4 API exposes direct control of this roll-rate controller, indicated by 'from other module'; this may be a velocity-controlling flight mode or custom controller applications. However, when using position-control methods, the roll-rate setpoints are generated with the position controller, which is a straight proportional controller. In our system, this configuration is used, as we stream position setpoints to the vehicle via mavros.

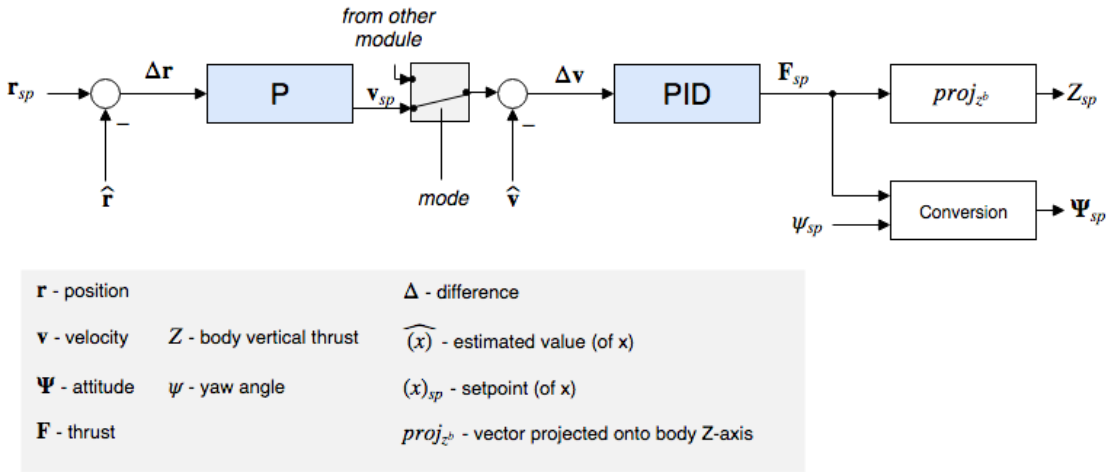


Figure 7: PX4 Position Control Diagram [12]

There are multiple options in ECL for generating estimations of position, attitude, and velocity. The newest option in the library is EKF2, which is an Extended Kalman Filter estimating extensive states in the system alongside position including wind state, vibration, and other parameters. It can use many sensor sources, primarily GPS, magnetometer, and IMU. EKFs in general are extremely robust tools for generating optimal state measurements from nonlinear sensors, as well as fusing multiple sources with each other and an internal model, in an optimal fashion.

However, since we are using visual localization techniques, older modules of ECL were used since they have a longer history and better compatibility with visual systems. In other words, we want a system which will directly track Orb Slam's output, but continued experimentation with EKF2 proved it to have stability issues with this type of input. So, the system was setup to instead use the following combination:

- AttitudeQ Estimator: A complementary filter which fuses IMU readings to output a quaternion describing the vehicle's attitude. Requires a yaw measurement which in our case is the SLAM output (since the magnetometer is disabled)
- LPE (Local Position Estimator): An extended kalman filter using position measurements from SLAM to generate an (x, y, z) position

4.4.2 Ground Control Station

Ground Control Station software is run on a 'base station' laptop and connects to the vehicle via radio link. It can be thought of as the graphical user interface for the flight controller, and is the primary method of interacting with the flight controller, alongside the transmitter. It allows the user to do things like change parameters, calibrate sensors, configure settings, update firmware, view current attitude/ground speed information, GPS information (if equipped) and much more.

QGroundControl [13] (sometimes shortened to QGC) was chosen at the recommendation of our colleagues; some samples of its functionality are given below. Figure 8 demonstrates

parameter tuning vehicle status (battery, signal strength, etc.) Figure 9 demonstrates a full readout of all mavlink packets on the radio link, including the vehicle's position and setpoint.

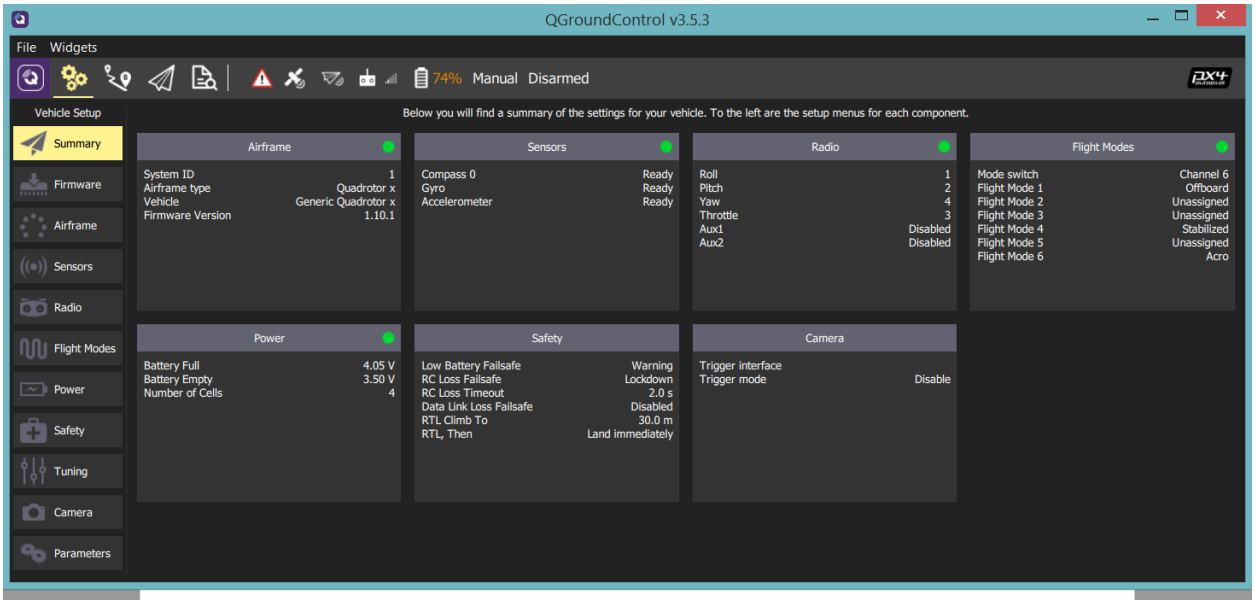


Figure 8: QGC: System at-a-glance

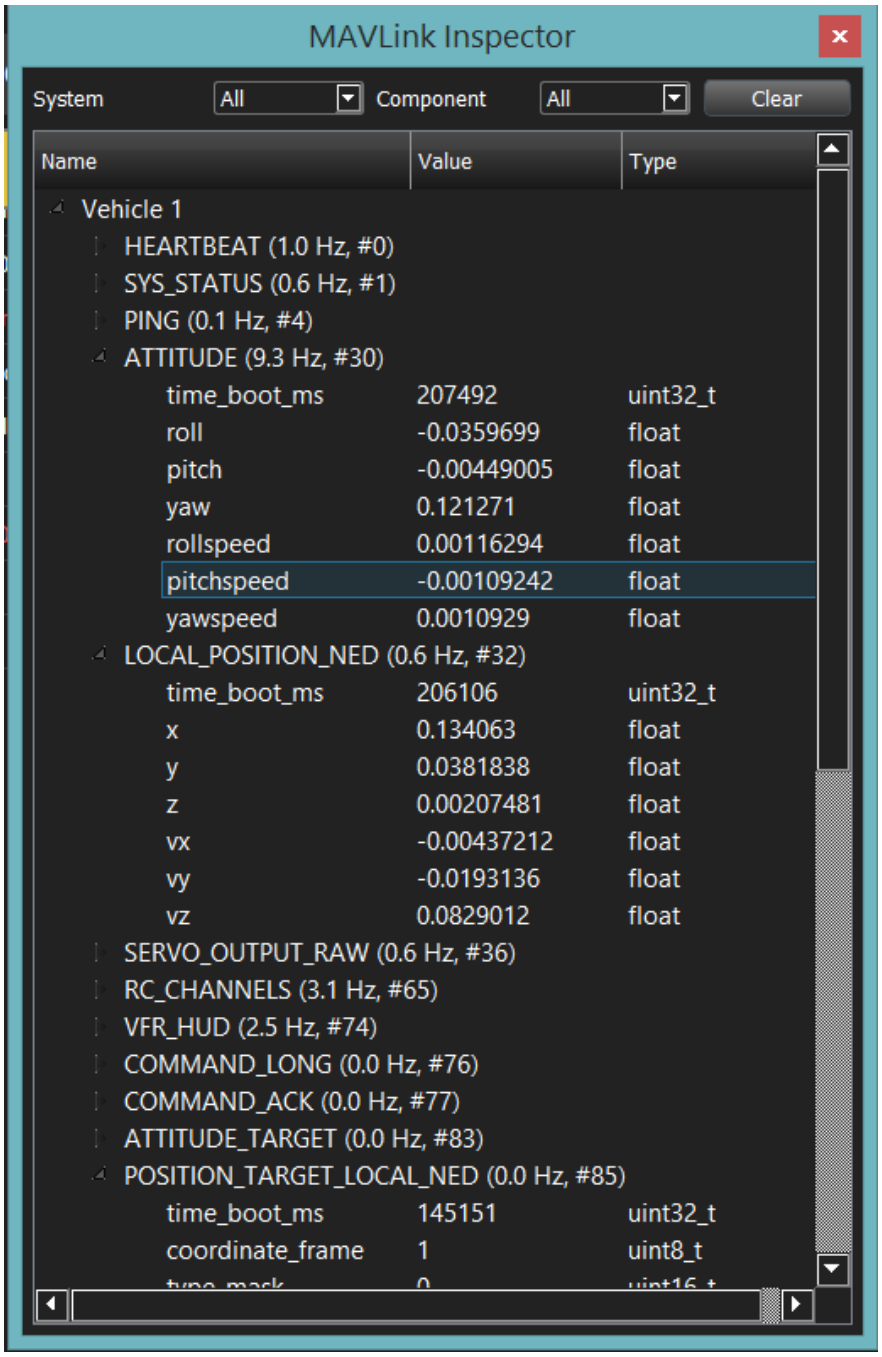


Figure 9: QGC: Mavlink telemetry including position, attitude, and current setpoint

5 Handheld Prototype Platform

In order to test the payload without flying, a non-flying prototype of the payload was first developed. Having such a platform enabled testing of all elements in an incremental fashion as each component was developed and integrated. For example, the ability to move the platform around in a carefully controlled manner allowed for testing and debugging of Orb

Slam 2. The eventual addition of a battery pack and small screen allowed for walkthrough mapping of the laboratory, for instance as seen in above in figure 4. From a system integration standpoint, this prototype served as the platform on which the software was developed (as they are inseparable portions of a robotic system) and facilitated testing for the interaction between new software nodes as well as using testing scripts to feed fake data to the hardware and analyze its performance.

Additionally, it was beneficial to create a handheld prototype due to various barriers to flight testing at NASA Ames. There are numerous approval processes required in order to obtain flight approval, including submission of Standard Operating Procedure document, acquisition of vehicle/equipment tags, safety training, and so on. Having such a prototype allowed development to continue unhindered while awaiting approval for flight testing.

5.1 Construction

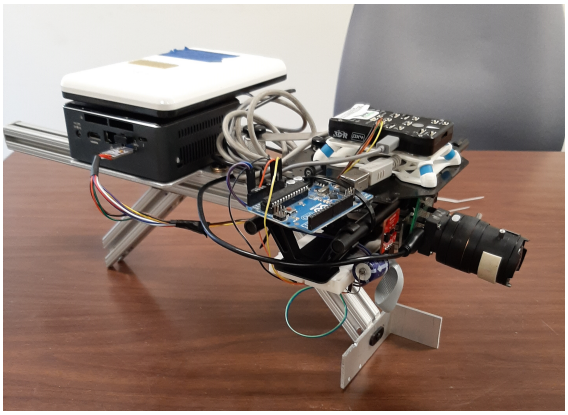
The handheld prototype was constructed with a frame consisting of 80/20 aluminum extrusion and the corresponding hardware T-nuts. It started as an early handheld rig for proof-of-concept work prior to my joining the project, constructed by Dr. Murakami. It contained only the camera assembly, lighting/circuitry, and some focusing beams. The prototype evolved from there as we reworked the shape, shifted placement, and added components piece by piece.

A shaped carbon-fiber sheet, which was taken from a drone kit lying around the lab, was used to house most of the components. Each of the components was attached with methods such as velcro, screws, and double-sided tape. A breadboard implementing the circuitry described in section 3 was attached to the underside of the sheet alongside the long-range distance unit. Finally, the sheet was bolted to the handheld frame. The NUC and battery pack were attached using a custom metal plate which was constructed in the AOX machine shop by the resident machinist, David Keil. A great deal of the frame construction was aided by consultations and fabrications by Keil. Having an expert machinist available for consult and fabrication, alongside a fully equipped machine shop[5], allowed for many excellent learning opportunities and carefully crafted equipment.

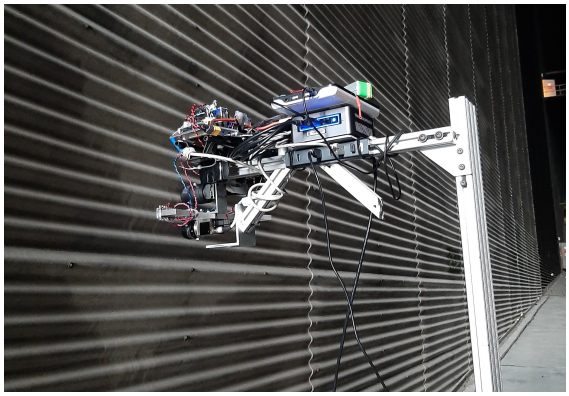
As a method of quick construction, the aluminum extrusion prototyping proved to be highly versatile, as sections can be easily added, removed, and shifted. Construction consists mostly of simple bolts, slid to any desired position. For example, when the weight at the front became unwieldy, it took very little time to bolt on a second handle and move the first one back to adjust the center of gravity (see figure 10a).

After several months of handheld testing and development, the drone stick was ultimately used for a non-flying data run in the NFAC. The aluminum frame construction made cart-mounting as simple as bolting it to a longer piece of extrusion, which was clamped to a rolling cart (figure 10b.) The results of this run are described further in section 7.

[5]including relevant safety trainings and PPE usage



(a) Aluminum frame with payload components



(b) Cart-mounted prototype inside NFAC

Figure 10

6 UAV Platform

Following the testing and development of the handheld prototype, we began converting the system to a proper UAV by migrating the system components to the actual flight vehicle. A good deal of drone hardware was available for our use which was taken from a previous summer internship group overseen in part by Dr. Murakami. This hardware was leveraged in addition to the custom payload to integrate the UAV system as follows:

6.1 Standard UAV Components

There are numerous components related to the operation of the drone itself (ie, not specific to our payload.) A good understanding of these 'vanilla' components is necessary for troubleshooting. The project uses following hardware, which forms a standard UAV setup:

Flight Frame

The flight frame chosen was a Byrd erector set which was used by a group of summer interns in the past and available for our use. It is a modular setup consisting of carbon fiber tubes and plates which are interlocked using metal clamps and associated bolts. The system is quite flexible, and can be set up with common frame styles such as quad-rotor, hex-rotor, and octo-rotor (in a 4-arm coaxial configuration.) Our UAV is configured as a 4-arm x-shaped quadrotor. The four arms attach via the clamps to a two-tiered main body segment which houses the power distribution board, pixhawk, and motor drivers.

The landing gear attaches to the main body and includes anchor points for strapping in the batteries. The frame is intended to be foldable via a rotating clamp system, though modifications were made to instead bolt the components directly together in order to mitigate vibration issues encountered during flight testing.

Power

The system is powered with lithium-polymer (LiPo) batteries. This style of battery requires many precautions during charging and use; failing to properly handle them can result in serious fires. The following are key elements of safe handling for LiPo batteries [14]:

- Nominal, Storage, Dead voltage: A single LiPo cell has a nominal voltage of 3.7 volts. When fully charged, a cell is 4.2 volts. Cells should never be allowed to discharge below 3 volts, else there is a risk of fires, swelling, and permanent cell damage. When out of use for long periods of time, it is best for the battery health to store it at 3.8V per cell.

- Cell configurations: To achieve desired voltages, cells are often stacked in a series configuration. For example, a battery pack designated 3S contains 3 series cells, resulting in a nominal voltage of 11.1V, and an absolute minimum charge of 9V.

- Discharge rating: Battery packs are rated for a safe amount of discharge, designated with a capacity- or C-rating. This rating governs the highest amperage permissable for a battery pack, and is calculated as $capacity * C$. For example, our 2.7 AH (2700 mAH) batteries rated at 35C can safely provide $2.7 * 35 =$ **94.5 Amps** to the motors. This significant output is part of why they are so useful for drone motors and yet so dangerous to mishandle.

- Charging safety: Since a pack contains several cells, a specialized charger must be used which can perform a balance charge. As cells age and wear, their internal resistance changes, and thus their charging rates; so the charger must actively work to ensure each cell is equally charged. A battery pack should be charged at no greater than 1C (ie, no faster than 2.7A for our 2.7AH battery). Batteries should be charged in a designated fireproof case (a 'bomb bag') in case something goes wrong, so that smoke can be vented without allowing a fire to start.

The drone is equipped with two 4s (4-cell) LiPo batteries connected in parallel, yielding a nominal system voltage of 14.8 volts, and a power distribution board is situated within the main body which routes power from the batteries to all other components. Bullet connectors ensure secure connection and allow for tapping into the power rails for custom purposes. These connectors were utilized to connect the NUC and LED arrays directly to the power rails through this board.

To ensure safe operation of the batteries, we used small digital volt meters attached to their charging leads to monitor the voltages. They are configured to sound an alarm upon low voltages, as well as display the charge level of each individual cell.

Actuators

The standard actuator in UAV systems is the outrunner brushless DC (BLDC) motor. This variety of motor has very high torque relative to its mass, making it ideal for aircraft propellers. Each motor is mounted at the end of the four arms (hence, it is a quadrotor setup) and affixed with a propeller. Two spin clockwise, and two spin counterclockwise. In this manner, the vehicle controls its yaw rate by adjusting motor speeds and exploiting conservation of angular momentum.

BLDC motors require Electronic Speed Controllers (ESCs) which convert control signals (ie, PWM) into 3-phase power signals which drive the motor. The ESCs are responsible for routing battery power from the power distribution board to the motors and thus use very thick wire. The 4 ESCs are mounted to the main body of the flight frame and can be seen in figure 12a (red rectangles)

Pixhawk and Related Radios

The Pixhawk was affixed to the main body with vibration-dampening foam tape. This helps isolate its internal sensors from the rest of the body, which is susceptible to vibration and other mechanical motor noise during flight. There are two radios required for pixhawk operation, both of which are affixed to the flight frame:

- Telemetry radio: standard SiK radio which communicates with its counterpart at the ground control station. Transmits mavlink packets from the Telem1 serial port across the radio link. These are the same set of mavlink packets which are broadcast through the Telem2 serial port to the NUC for use within the ROS system
- Receiver radio: paired to the remote control transmitter for relaying flight commands (eg, with the joysticks) and setting of other parameters through switches and knobs. These commands are also broadcast on Telem2 into the NUC

In addition to the radios, the TELEM2 port is connected to the NUC via ftdi, the battery is connected via a power module (providing logic-level power and battery voltage readings), and a safety button is attached which must be pressed before arming can occur.

6.2 Payload Packaging

The hardware discussed above encompasses a typical drone setup which would yield a system that could be flown manually. The following sections describe the work done in order to package the custom payload into the system. Along the way, both here and with the prototype, much guidance was provided by the branch machinist David Keil, who taught me valuable information regarding the safe usage of shop tools and provided professional fabrication work when needed.

Camera Bracket

The two cameras were mounted to a custom metal bracket, which started as a stock L-shaped bracket. Holes were drilled in order to accommodate the threaded holes in the chassis of each camera, which were used to attach the cameras. A large hole was drilled to allow a carbon-fiber tube through, followed by 4 smaller holes to attach a Byrd clamp. Finally, the clamp was used to affix the camera bracket to a carbon fiber tube placed in the front section of the body as seen in figure 12a. Additionally, the front-facing distance sensor was attached to this bracket with double-sided tape.

LED Driver/Mounts[6]

The LED drivers were configured to accept GPIO output from both cameras, such that the lights trigger anytime a camera shutter is open. That circuitry was wired to the LED arrays, for which custom ball-and-socket articulated mounts were 3D-printed and attached to the two front motor arms.

Sensor Interface Board

Initial work regarding the distance sensors was done with a breadboard, which was useful for getting everything right. However, it proved problematic and difficult to work with during testing as breadboards aren't permanent solutions. So, a sensor board was planned and soldered together in order to consolidate the circuitry with its related Arduino; this resulted in much more stable and clean operation.

The LED drivers and sensor interface board were mounted on a plastic plate by drilling in holes to accommodate standoffs. The downward-facing range sensor was also mounted here. Byrd system clamps were attached to the opposite side. This resulted in a tidy electronics package (shown in figure 11) which can be mounted to the carbon fiber tubes at any accessible point in the UAV. A few iterations were tested in order to achieve good balance in the center of gravity of the system as discussed below.

[6] planned/fabricated exclusively by Dr. Murakami

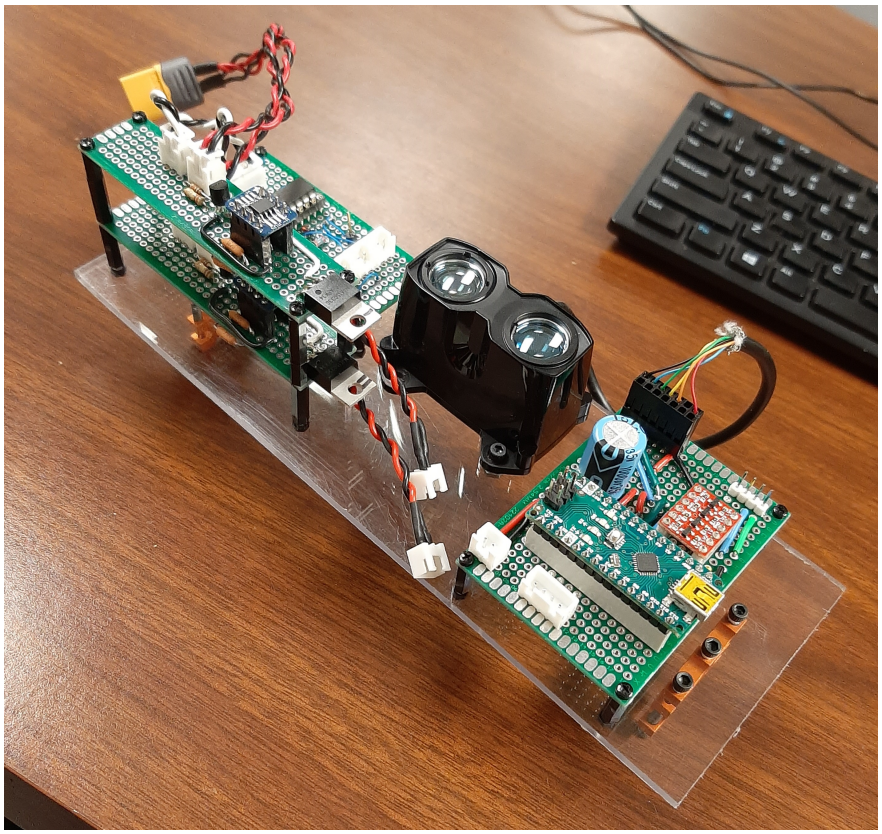


Figure 11: Circuitry plate with LED driver, rangefinder, and sensor board (left to right)

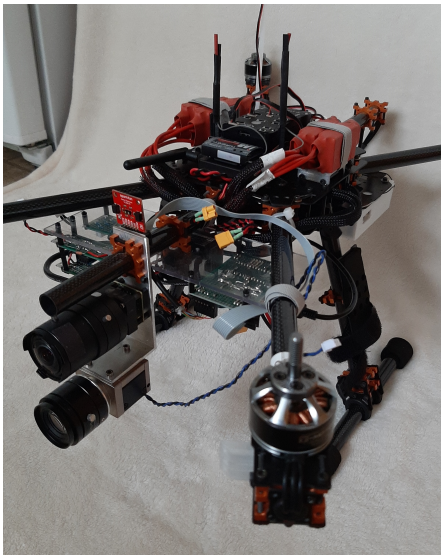
Weight Reduction

A few weight-reduction strategies were employed in order to minimize the vehicle weight. Cable runs were shortened, as the Byrd system came with quite lengthy power and ESC leads. USB cables were chosen to be as short as possible, and in the case of the ftdi board, eliminated entirely via the NUC onboard USB headers. Since the standard NUC case is metal, a custom case was designed in SolidWorks and 3D printed. The NUC was then mounted to a carbon fiber plate and bolted to the UAV using the clamp system. This allowed for an enclosure that was very lightweight, but still protected the computer from shorts and other hazards. Additional weight could likely be saved by replacing the stock landing gear with something lighter. The final goal for the payload is 1 kilogram.

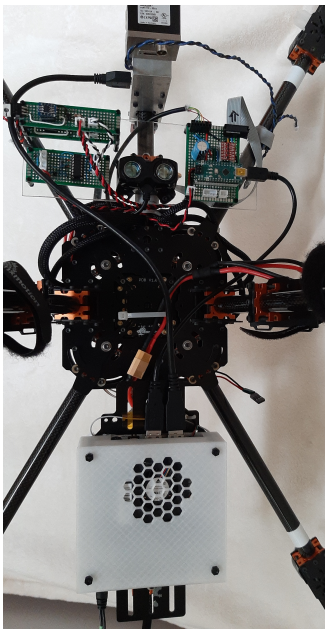
Big Picture: UAV Platform

Figure 12 shows the completed UAV with payload packaging. The front view shows the dual camera mount, flight controller, ESC's, and one of the motors. The underbelly view shows the 3D printed NUC case and sensor board mounted to the body. The placement of these components is done in such a way that the center of gravity remains relatively close to the center; the NUC counterbalances the cameras and, although the payload does add rotational inertia along the pitch axis, the vehicle feels good in the hands. Furthermore,

controller tuning should easily compensate for the new mass, especially given that effort was put into keeping an even center of gravity.



(a) Front View



(b) Underbelly View

Figure 12: UAV Platform

6.3 System Operation

Operation of the UAV system currently involves the following rough steps:

1. Ensure NUC is up and running the ROS architecture, and that the pixhawk is visible within the ground control station
2. Move/wave the system around laterally until a SLAM fix is found, indicated by the status LED turning blue
3. Hold the UAV flat or place it on the ground, and move transmitter control knob to begin scaling mode
4. To scale into meters, move system up and down for about 3 seconds in each position, ensuring the downwards-facing distance sensor is unobscured. Repeat a few times
5. Place UAV at the desired home position and move the control knob from 'scale' to 'home' position
6. In ground control, verify that the position and attitude have been homed to 0, and that the setpoints are visible.
7. Ensure proper distance and Personal Protective Equipment of all operators. Arm the vehicle, which spins up the propellers, and flip the control switch to 'Offboard mode' to initiate the mission.

7 Testing and Results

7.1 Prototype Tests

Initial tests of the localization nodes (Orb Slam 2, scaling, and rotating) were performed incrementally as new hardware components were added to the prototype. With each addition, the hardware element was tested individually via custom ROS launchers before adding them into the full system. This allowed for individual debugging via dummy scripts and other methods, with minimal interference upon integration due to ROS's modular nature. Tests such as this are demonstrated in figures 2 and 4.

More involved tests of the localization capabilities were performed in a wide-open space which allowed for marking the floor and wall with tape at set increments (ie, forming a .5 meter grid pattern) in order to pinpoint issues with Orb Slam 2. This highlighted some issues such as positional drift and the need for the quaternion debugging node described in section 4.

The camera drivers were tested extensively in conjunction with Orb Slam2, as the nodes were in some ways difficult to work with; however once dialed in, the camera drivers performed wonderfully. There were similar issues with mavros when adding the flight controller to the prototype, relating to parameter settings and issues changing modes within PX4; however persistent testing ultimately resulted in a well-integrated system. The full prototype was tested in both laboratory and the target environment as described below, in order to verify the proper collection of images.

7.1.1 Lab Setting

To test the full functioning of the system, the prototype was mounted to a cart in order to provide a mobile workstation. The cart was outfitted with a monitor, keyboard, and other equipment required for operation, and made it easy to move the system in a controlled manner. During this point in the project, Nick Califano traveled from Florida to work together during winter break. By his calculations, his machine learning system required a grid spacing of **11.89cm by 12.75cm** in order to capture the necessary detail (approximately 25 images per feature) and so the setpoints were generated with this spacing.

We ran the cart setup through several days of testing, debugging along the way. The run-throughs were performed in a few locations, including a large open-space portion of the lab, and in the conference room. They were also performed in the dark when possible, in order that the lighting rig could also be debugged.

There were some difficulties with this method of testing; cart-mounting the system made it cumbersome to perform the SLAM scaling operation (moving system up and down) and the centimeter-scale spacing proved to be very tight, especially with the cart's lack of free-spinning casters. A simple solution was to increase the threshold of the setpoint handler; however after further difficulties we decided to only measure in the horizontal (X) direction and manually increase the Z distance, while keeping Y (wall-facing) 1 meter from the wall. Reducing the setpoint dimensionality allowed

Difficulties aside, we were able to get good results; given in figure 13 is a sample of the dataset taken in the conference room. One can see the 'lawnmower' nature of the images as

the system was swept left to right along the bookshelf.

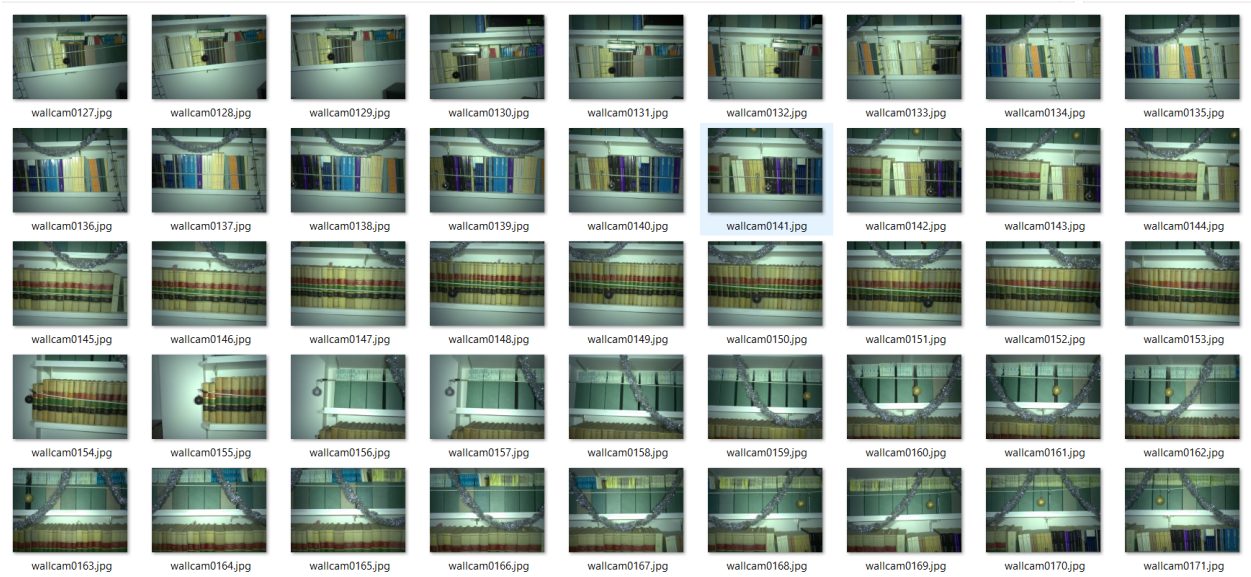
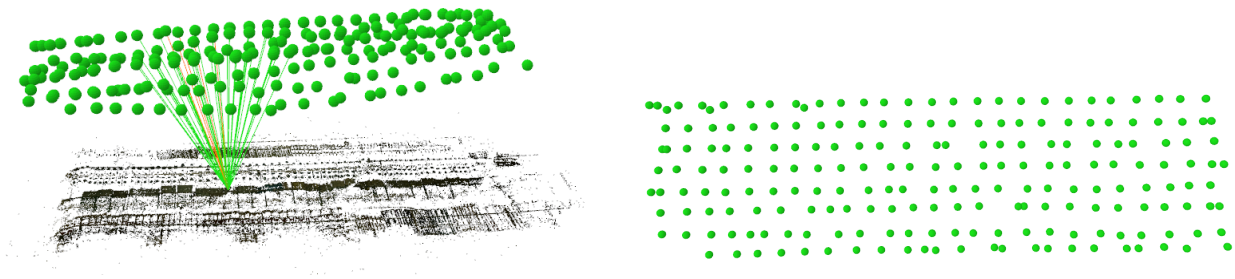


Figure 13: Subset of conference room dataset

The real test of the system is its performance with the setpoint handler node, which combines all relevant data in order to perform the system's intended purpose by capturing images at setpoints. In other words, exhibiting this functionality implies all other software is behaving correctly, as the dysfunction of any one node would make this impossible. Running full-system tests such as this serve as the ultimate test of the payload, as the localization, frame correction, sensors, cameras, and flight controller work together in concert to achieve the desired purpose.

Given below in figure 14 is a 3D reconstruction analysis of the conference room run, generated by Nick Califano using this dataset. It was reconstructed with the PIX4D program (an industrial post-processing photogrammetry and point-cloud software) and shows the camera positions (full 6-DOF poses) of each captured image maintaining a roughly 1M distance. The poses of each image in the dataset can be seen in relation to the reconstructed bookshelf. The grid pattern is shown at right, demonstrating that the system localized and took pictures roughly at the desired grid pattern. It can be seen that there are irregularities in grid spacing which may be attributed to inaccurate scaling, tolerances allowed in the code, uncertainty within PIX4D, and the occasional missed setpoint (requiring us to move the cart back, capturing it from the other direction)



(a) 3D camera positions (view from below) (b) Grid pattern (flat view)

Figure 14: Camera Position Results (conference room)

7.1.2 NFAC Full Run

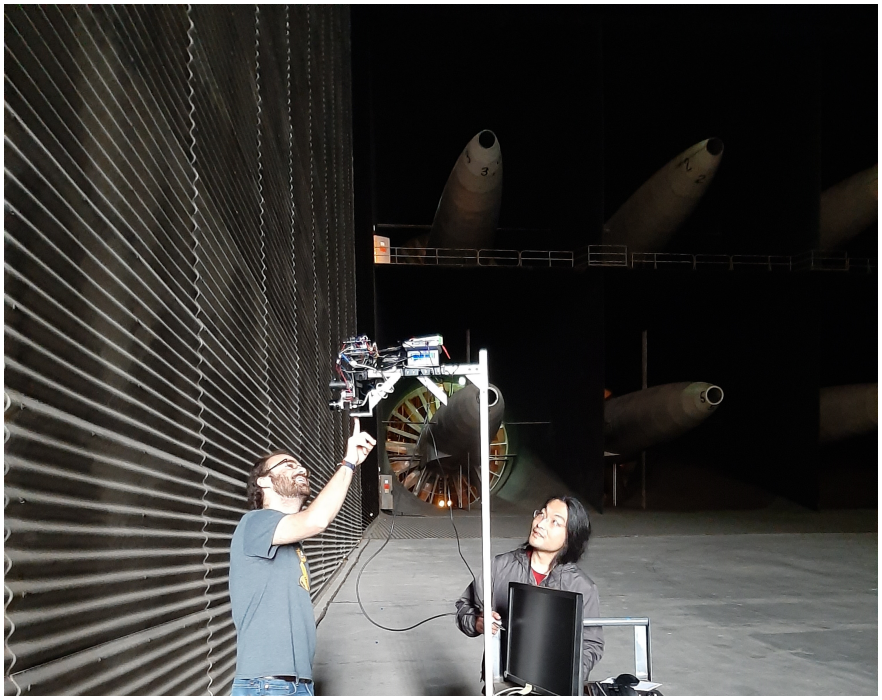


Figure 15: Cart in action!

In early December, we were able to complete a full system run in the NFAC over the course of a few days. The cart setup described above was transported to the wind tunnel when some time was secured for us to enter the facility. In order to make sure we were fully prepared, a pre-test checklist was devised since transporting the system to the facility was nontrivial. The list included such tasks as: charging all batteries, securing all hardware, gathering all supplies/tools, ensuring all cables were accounted for, and double checking connections. That last part was learned the hard way, as time was spent during the first day troubleshooting hardware issues related to the lighting circuitry, which was traced back to a faulty breadboard connection. In other words, **check power and ground!**

The ‘lawnmower’ pattern was set for a 10x1.5 meter grid, which was the highest our cart setup could accommodate. The grid spacing was set to the same as for the conference room runs, and again the y- and z- checking was disabled. We were able to do two full runs in adjacent sections; as a result, a 20x1.5 meter section of the wind tunnel was imaged, resulting in 1831 images of the tunnel walls. Following discussions with Nick Califano, it appears his classifier network, when used with this dataset, yielded a roughly 95% recognition rate for fasteners. Unfortunately, this patch of wall did not contain enough missing fasteners for further training of the neural network, but it was regardless an important milestone demonstrating the ability to localize and image in the target environment.

The SLAM system can be seen successfully detecting features from the NFAC walls during this run in figure 16 alongside the setpoint handler functionality:

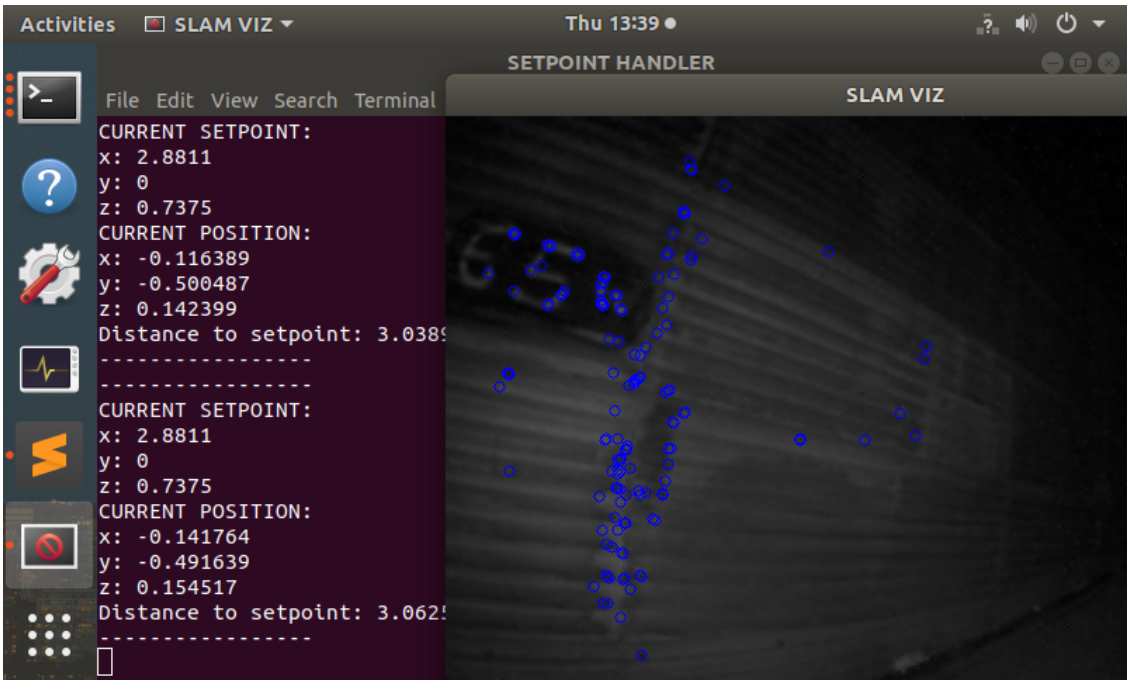


Figure 16: Setpoint node running alongside SLAM visualization inside NFAC

As in figure 14 above, analysis of the NFAC dataset was performed using PIX4D. Since the program can output the camera positions in 3D space, Using these results as a ground truth, it was found that the mean X-direction spacing of the images was 14.04 centimeters, with a standard deviation of 4.9 cm. The commanded horizontal spacing was a constant 11.89 cm, indicating that the localization for this run yielded an RMS error of 5 cm. Possible sources of this error include improper scaling, inaccuracies within Orb Slam, and calibration errors. Furthermore, PIX4D is not a perfect ground truth; it contains its own estimation uncertainties and homes using a 2D barcode which may have been measured innacurately and not placed perfectly level.

Earlier runs demonstrated lower error, but were not conducted on this scale, indicating that the system may have trouble on longer-range missions. Provided below is a graph of the X-Y positions, after translating the data such that the first points is (0,0). It can be seen that we captured the slope of the floor and maintained decent Y-spacing.

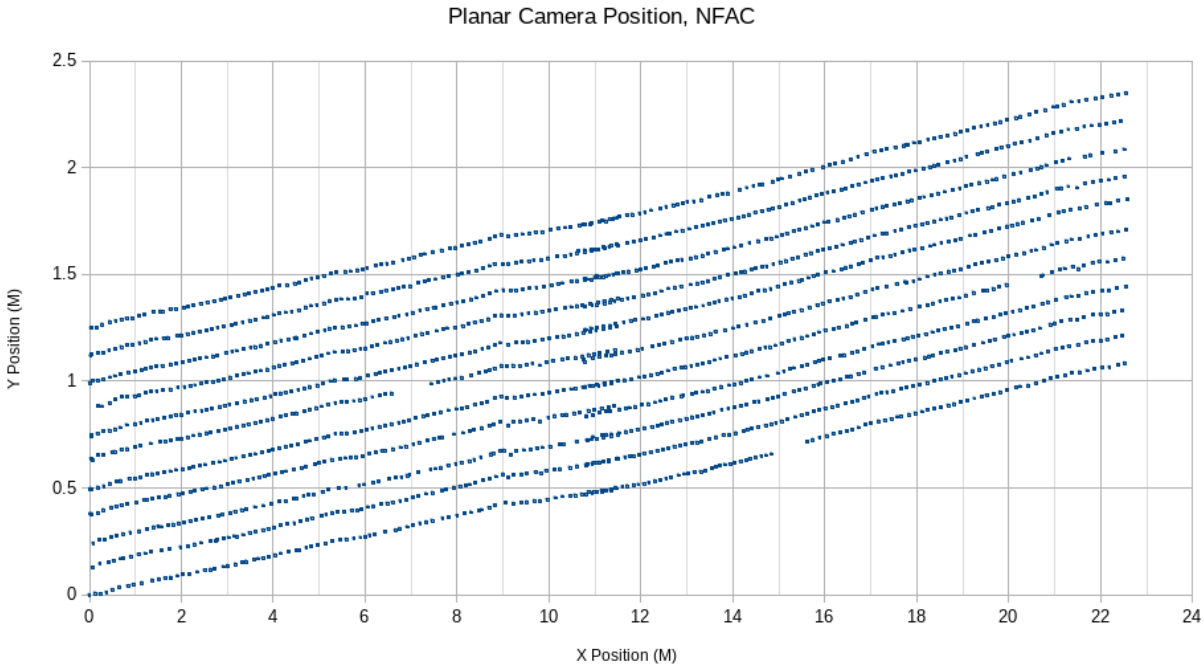


Figure 17: X-Y plot of wall positions

7.2 Flight Tests

Though NASA Ames is home to several flight test facilities, it is a challenge to gain access to them. As the area is restricted airspace, flights are limited to indoor areas, and even then require paperwork and other approval processes. This is for good reason; the procedures for safe flight require, among other things, a Standard Operating Procedure, multiple pilots including safety pilot, and safety trainings for the testing area. Early on, the NUARC[7] facility was available for our use; the building included multiple safety nettings, proper PPE (hard hats, safety glasses, etc) a large fan to simulate disturbances, battery charging and ground control stations, fire extinguishers, and so on. Training was completed for use of the NUARC, but the system was not yet ready for flight. Currently, the building is under renovation and unavailable for use; regardless, safety training and building approval was an important step in learning protocol for such facilities.

By the time the UAV was constructed and ready to fly, the NUARC was under renovation. However, we instead secured permission to fly in an unused hangar elsewhere at Ames Research Center with the goal of setting the space up as a future drone room. The approval process required a vehicle tag, inspection by a flight official, and approval for specific flight times.

After gaining access to the hangar, we began manual flight testing. For each flight, a log was taken including flight time, battery voltage at start, voltage at end, and notes. The purpose of early manual flights was to tune the controllers and ensure our vehicle could fly without issue. Tests were performed without a payload attached, given that untuned

[7]NASA Unmanned Aerial Research Center

controllers mean instability and possible crashes.

Throughout these tests, we noted a yaw drift. In other words, commanding the yaw to stay still resulted in motion. This could be due to a number of factors, including misalignment in the motor mountings or excessive vibration entering the pixhawk. Indeed, we learned after the first test that the flight frame was introducing vibration problems despite the isolating foam.

The graphs below shows the IMU readings from a test flight, demonstrating excessive vibration during a simply hover maneuver. They were generated using the Flight Log online tool provided by the PX4 development team, and analyzed with the aid of [2]. It can be seen that vibration is present along each accelerometer axis in excess of $\pm$ 10 m/s^2 with even higher magnitudes present along the y-axis.

In addition to causing mechanical issues (for example, screws vibrating loose) this type of noise makes it difficult for the attitude filters, which are performing integrations of this, to do their job. This was likely a contributing factor to the vehicle's inability to maintain yaw, especially without the magnetometer to help stabilize the estimation.

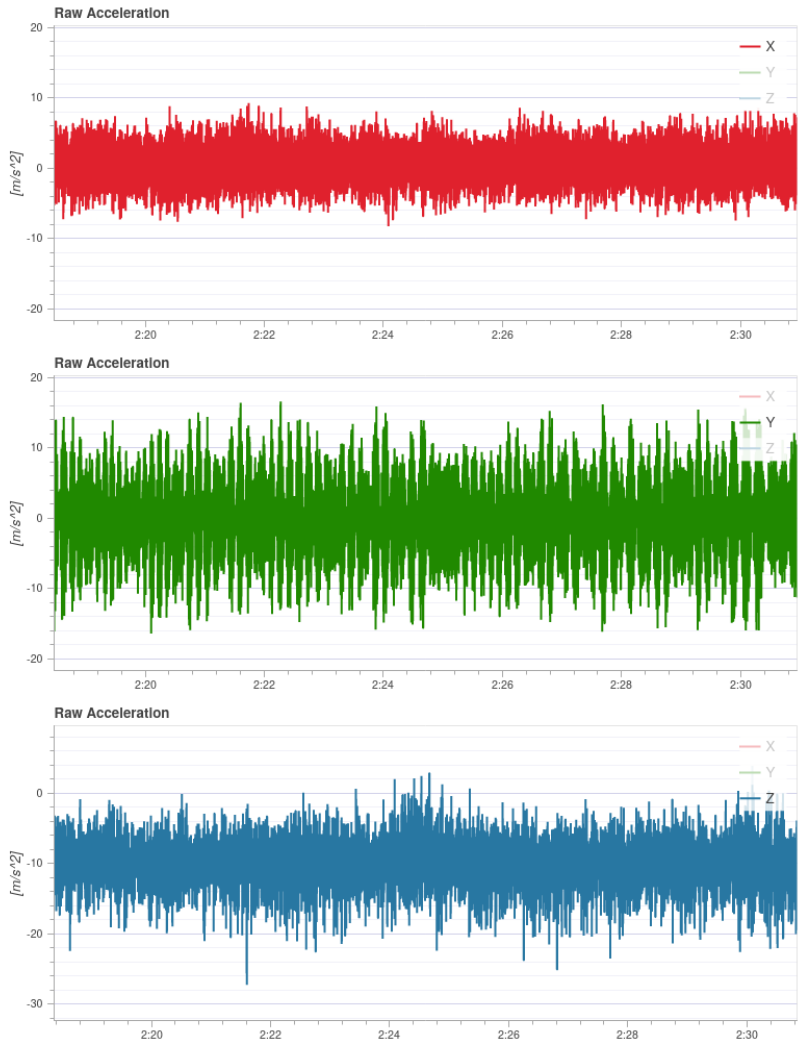


Figure 18: Accelerometer plots demonstrating significant vibration along each axis. Note z is centered around a gravitational offset roughly equivalent to 9.8

Another view of this is given below in a normalized Power Spectral Density graph, demonstrating that the vibrational energy is present across a wide frequency band. It can be seen that the acceleration signal to noise ratio is very poor; in a well-performing setup, the accelerations should reflect only the vehicle's movement and gravitational forces (which should manifest only at low and DC frequencies.)

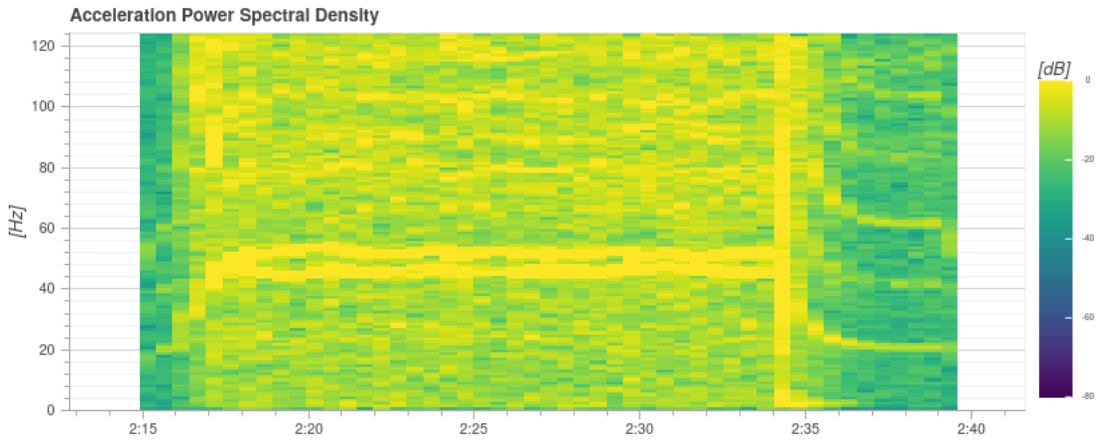


Figure 19: Power Spectral Density plot of accelerometer readings with poor SNR

In light of the above findings, adjustments were made to the flight frame involving further securing components and bolting the arms in place, rather than the default folding operation. The landing gear received the same treatment. This sacrifices the drone's ability to be folded, though foldability was never a concern to begin with. Future tests will be performed to determine if our modifications to the flight frame have helped mitigate the vibration issues and lessened the observed yaw drift.

8 Future Work

The high degree of interest in this project combined with the growing ubiquity of UAVs makes for a project with very promising future prospects. Additionally, my participation in the project has been extended through September 30th at the time of this writing.

Near-future work on this project revolves primarily around further flight testing. Currently, flight tests have been performed manually for the purposes of controller tuning, and without payload. These tests will be expanded to include manual flight with payload (to test localization under pilot control and tune controllers for payload) and semi-autonomous flight. Determining the performance of setpoint following will be important to analyzing the performance of the system when operating fully autonomously. For example, a simple flight of moving from point A to B followed by more advanced missions within our flight testing hangar. Flight testing can resume once Ames Research Center has been cleared for reentry following the Covid-19 pandemic measures in place.

With more advanced flight testing yielding more enlightening results on the system performance, we can then move towards flying in the NFAC itself. However, since this is a very high-stakes endeavor, we will naturally need to make sure all issues are ironed out before taking that final step.

Future methods of control and calibration may be employed as the project evolves. The method of moving the system up and down is often inaccurate and tedious, and could be better solved by a method of operation in which the user moves the system between two measured points and presses a button to indicate a set real-world distance has been moved. Additional computer vision methods for scaling may be applied.

Further localization methods may be explored, such as portable ultrasonic beacons, vicon systems, or all-in-one visual system such as the Intel T265 stereo camera.

Further weight reduction/packaging strategies will be employed, such as designing and populating a custom printed circuit board.

Since operation must be done without an attached monitor, additional telemetry may be required. For example, a small OLED screen may be added to better convey system state, and modifications to the mavlink protocol could be made in order to collect more verbose status. Additionally, some RC transmitters natively support mavlink telemetry, which could be a further source of information.

9 Conclusion

Following the procedures documented in this report, we were able to create a localizing and imaging payload to facilitate aerial inspection within the National Full-Scale Aerodynamics Complex at NASA Ames Research Center. Though the payload has not yet been flown, this work helps pave the way for future flight testing encompassing indoor flight and autonomous inspection missions aiding the inspection of this mission-critical test facility.

Tests within the NFAC demonstrated the viability of visual localization techniques in this target environment even with walls containing minimal texture. As such, this work shows promise as an option for indoor flight, which is a notably difficult endeavor. Furthermore, we demonstrated viability of camera-based diagnostics, as we generated a new dataset consisting of diagnostic images during cart-mounted testing (about 1800 images) for processing via machine learning, which was used to detect features within the NFAC walls with high accuracy.

Testing of the payload highlighted issues with inaccuracy and drift in the measurement, and it was necessary to alter the thresholds to work mostly in one dimension. As such, it will be beneficial to spend more time analyzing the scaling accuracy of Orb Slam 2, implementing computer-vision based scaling, using a stereo camera, or trying other localization techniques.

The software developed during the course of this work uses ROS, which is highly modular, and therefore makes an excellent platform, educational or otherwise, for testing new localization and path-planning techniques.

The operational procedures described in section 6.3 were developed and utilized, and highlighted both challenges and benefits of the approach. An important takeaway during project testing was that more interaction capabilities with the system are necessary, as a monitor will not always be available, and stock Ground Control Station telemetry is relatively slow and difficult to parse as QGC natively displays it. As such, future work (discussed further in section 8) should include enhanced telemetry and control methods which aren't dependent on a monitor, especially when flying autonomously.

In preparation for using the payload during actual flight, space for flight testing was secured, and early manual flights were performed with the intention of controller tuning and mechanical analysis. Flight testing was approved by the administration and adhered to NASA-mandated safety procedures. Vibration issues were identified with the frame, which were a likely culprit for an observed drift in yaw. Mechanical fixes were implemented and await further flight testing at such a time as it is viable to return to the flight space.

Summary of Topics Learned and Applied

I started this project with decent experience in embedded coding and mechatronics systems, but with little to no knowledge regarding typical UAV setups or the wider field of aerospace engineering. The work performed during the course of this project yielded a highly bolstered understanding of all such related topics, old and new. In some cases it was a crash course and in others a supplement to all my academic training. Even the skills I thought I knew well were improved during the course of this work. Additionally, it was an excellent chance to learn best practices, professional engineering communication, and other core aspects of the transistion between academic and professional engineering. Critical topics which were learned and applied include:

- Standard UAV components including actuators, radios, flight controller, and airframe construction
- Electrical workmanship and practical electronics crafting. Design using crimped and locking connectors, strain relief, and other best practices
- Rapid prototyping workmanship: interaction with machinist; CAD/3D printing; machine shop tool and protective equipment familiarity
- Software development in linux; package installation flow; virtual machines; CMAKE and related build systems
- Safe handling and usage of LiPo batteries including best charging practices, discharge ratings and curves, and proper voltages (storage, nominal, dead)
- All aspects of ROS including modular/distributed programming, publisher/subscriber models, and visualization
- Visual localization and SLAM techniques; core tenets of computer vision, including camera calibration, operational parameters, projection matrices/triangulation, and OpenCV
- Aerial vehicle attitude conventions and techniques, including quaternions, roll-pitch-yaw, and estimation thereof. Writing the quaternion operations header and debug node proved to be invaluable, as the attitude adjustment process took a fair amount of trial and error to get right.

References

[1] NASA, 2005. [Online; accessed June 18, 2020].

[2] PX4 Dev Team. Px4 autopilot user guide (license: cc by 4.0). https://docs.px4.io/master/en/index.html.

[3] Montiel J. M. M. Mur-Artal, Raúl and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[4] https://github.com/raulmur/ORB_SLAM2.

[5] https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[6] https://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration.

[7] http://wiki.ros.org/.

[8] https://github.com/basler/pylon-ros-camera.

[9] https://github.com/appliedAI-Initiative/orb_slam_2_ros.

[10] https://github.com/ros-perception/image_pipeline.

[11] https://github.com/mavlink/mavros.

[12] PX4 Dev Team. Px4 development guide (license: Cc by 4.0). https://dev.px4.io/v1.9.0/en/.

[13] https://github.com/basler/pylon-ros-camera.

[14] Hyperion-world.com. Lipo handling guide, 2016. https://hyperion-world.com/media/attachment/file/LiPo-Handling-Guide.pdf.

Appendices

A PX4 Guides

The following links are provided to point out specific useful subsections of the PX4 documentation[2][12]

- Full wiring diagrams:

  `https://docs.px4.io/master/en/assembly/quick_start_pixhawk.html`

- Hardware pertaining to full UAV setup (menu at left):

  `https://docs.px4.io/master/en/peripherals/`

- Setting up PX4 to connect to companion computer (NUC):

  `https://dev.px4.io/v1.9.0/en/companion_computer/pixhawk_companion.html`

- Setting up LPE for external vision estimation (Orb Slam 2):

  `https://dev.px4.io/v1.9.0/en/ros/external_position_estimation.html` and `https://dev.px4.io/v1.9.0/en/advanced/switching_state_estimators.html`

- EKF Tuning and Overview:

  `https://docs.px4.io/v1.9.0/en/advanced_config/tuning_the_ecl_ekf.html`

- Controller Diagrams:

  `https://dev.px4.io/v1.9.0/en/flight_stack/controller_diagrams.html`

- Offboard mode:

  `https://docs.px4.io/v1.9.0/en/flight_modes/offboard.html`

- Other Flight Modes:

  `https://docs.px4.io/master/en/flight_modes/`

- Other computer vision options supported by PX4:

  `https://docs.px4.io/master/en/computer_vision/`

- Parameter Reference:

  `https://dev.px4.io/master/en/advanced/parameter_reference.html`

- Flight Log Analysis:

  `https://docs.px4.io/v1.9.0/en/log/flight_review.html`

B ROS Installation Steps

1. clean install of ubuntu 18 + software updates

   -Install sublime (or editor of preference):

   ```
   http://tipsonubuntu.com/2017/05/30/install-sublime-text-3-ubuntu-16-04-official-way
   ```

   -When working in a vm, using a shared folder within virtualbox is very helpful. By default, it needs root so run:

   ```
   sudo adduser <username> vboxsf
   ```

   to allow access without entering credentials each time

2. Full install of ROS Melodic using the tutorials:

   ```
   http://wiki.ros.org/melodic/Installation/Ubuntu
   ```

   -dont forget to do this to source the catkin ws for terminals in your bashrc:

   ```
   echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
   ```

   -also consider this command to let rosedit use sublime:

   ```
   echo "export EDITOR='subl '" >> ~/.bashrc
   ```

   -moving the full catkin_ws from machine to machine via flash drive or git seems to work ok, with the exception of the symbolic links typically generated by previous builds. So far it seems that we can just run 'catkin_make' at the root of the workspace to regenerate those links.

3. Installation of needed libraries for the project; unless otherwise noted, follow installation instructions from each site:

   (a) PylonSDK: (Camera driver SDK)

   ```
   https://www.baslerweb.com/en/products/software/basler-pylon-camera-software-sui
   ```

   Instructions found in the sdk folder, file titled INSTALL. Part of the process will be to run 'sudo ./setup-usb.sh' from the source folder to install the udev rules.

   (b) OpenCV: (image processing) Installed manually at first, but it actually gets installed with a full ros install

   (c) Eigen: (Linear algebra/matrix support) same as opencv; originally installed manually but learned later it comes with full ros install.

4. USB Symbolic links Given that there are currently multiple devices which will populate as USB devices (uno32 or arduino, and pixhawk ftdi) we should rename the devices based upon their vendor IDs. This was done using the guide steps at:

   ```
   https://dev.px4.io/v1.9.0/en/companion_computer/pixhawk_companion.html
   ```

   and each usb-based node in the system uses the following USB symbolic names. These can be changed in the main launch file:

   pixhawk: /dev/ttyPixhawk

   uno32: /dev/ttyUno32 (not used)

   arduino: /dev/ttyArduino

   TO-DO: WRITE INSTALLATION SCRIPT TO AUTOMATICALLY DO THIS similar to basler setup.sh which renames the usb cameras

5. Installation of important ROS nodes; unless noted, follow installation instructions on each page.

   Generally, to install as a catkin package, extract sources to ~/catkin_ws/src and run catkin_make at the top level. Package.xml files will let the system know that a package exists.

   If the nodes have been previously built on another machine, it's necessary to delete catkin_ws/build/cmakecache.txt before building on the new one. This cache can cause the build to fail!! (for example, moving the entire catkin_ws folder to a new machine.) You can also just delete the workspace's /devel and /build top level dirs if needing nuclear option

   Note too that you use this command at the catkin_ws root to install every single dependency: (ie, it's ROS's package manager)

   ```
   rosdep install --from-paths src --ignore-src -r -y
   ```

   Pylon Driver Node: Interface driver for talking to Basler cameras and getting their images published onto ROS. There are TWO VERSIONS; we are using the newer one:

   ```
   http://wiki.ros.org/pylon_camera
   http://github.com/basler/pylon-ros-camera
   ```

   (this is the newer one)

   Note the step for telling rosdep that we need to have the pylon sdk installed, otherwise it can't find the location that pylon installs the sdk (/opt/pylon5)

   This one comes with some funkiness with camera names since we want to use two pylon cameras. To distinguish, the user sets a "device user ID" for the camera, and then the pylon driver node will look for that name. I found it easiest to just set these with the pylon viewer gui (in windows) and refer to those in the config files for each camera.

Currently named 'slamCam' and 'wallCam' for the navigation and diagnostic cams, respectively

Mavros: Interface to the pixhawk mavlink port. Translates all messages and includes plugins for specifying position, setpoints, and other useful things

```
https://github.com/mavlink/mavros/blob/master/mavros/README.md
```

Most of the setup can be done with rosdep and the install instructions. Note the need to manually run mavros/scripts/install_geographiclib_datasets.sh to provide the correct datasets to one of the dependencies

Orb Slam2 ROS: http://wiki.ros.org/orb_slam2_ros This is a ros-oriented implementation of the library and cuts some of the fat from the original repo

INITIALIZATION IS DIFFICULT! The two bug fixes mentioned here do help:

```
https://github.com/raulmur/ORB_SLAM2/issues/59
```

In files Initializer.cc and ORBMatcher.cc

Also: CALIBRATE THE CAMERA!!

6. uno32 Package: Written by us, c++.

   NOTE! You need to add user to the dialout group to allow usb access without root. Running as root doesn't let python resolve some of the ros includes anyways. This should also let you talk to the pixhawk on telem2. run:

   ```
   sudo usermod -a -G dialout [username]
   ```

   and reboot