2022

TRIUMF Photogrammetry Summer Co-op Term Documentation

MAY 2022 – AUG 2022

JASON YUAN

Table of Contents

Project Overview

This is my “documentation” of the tasks that I worked on from May 2022 – August 2022. I’ll create new sections for projects that I spent a few weeks on and classify the projects that took about a week or less together in a miscellaneous section.

Most photos, presentations and some related files are backed up to my OneDrive. I’m not sure where that will be or how you can access if my TRIUMF account becomes deactivated. I also have a copy of the codes, notebooks, and CAD files that I used during the summer found on my GitHub. Hopefully, no code or documents are missing but feel free to reach me at jason.yuan@mail.utoronto.ca if you ever find anything missing or confusing.

Camera Calibration

Overview

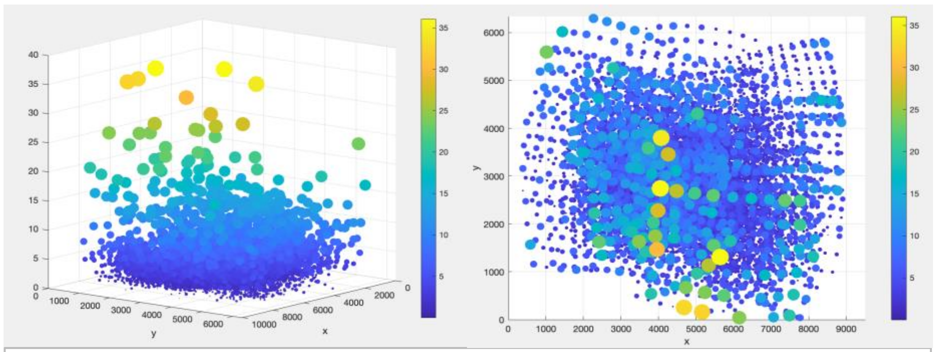
Geometric camera calibration is an algorithmic process that estimates the parameters of an optical system through various images, or a video taken by the target optical system. The estimated parameters from the calibration process can be used for lens distortion correction, depth estimation, size measurements and 3D scene reconstruction. In the case of this project, 3D scene reconstruction from 2D images is most pertinent in photogrammetry.

During my term over the summer, we were able to do a complete run on the calibration of the optical system — dome port plus camera in metal housing — underwater with the acrylic and glass dome ports. Additionally, another set of calibration photos were collected with the dome ports and camera in-air. Moreover, after calibration it became clear that the size of the image greatly impacted the ability of OpenCV and MATLAB's corner detection algorithm (See attached references for link to some of the reasoning). One workaround to this problem was to then downsize the image, detect the corners, and then upsize. However, much like signal processing, the process of downsampling and then upsampling means that information will be lost — this leads to further inaccuracies with the detected corners. Running Xiaoyue's corner error script will help to quantify the amount of error on the detected corners with this downsizing approach.

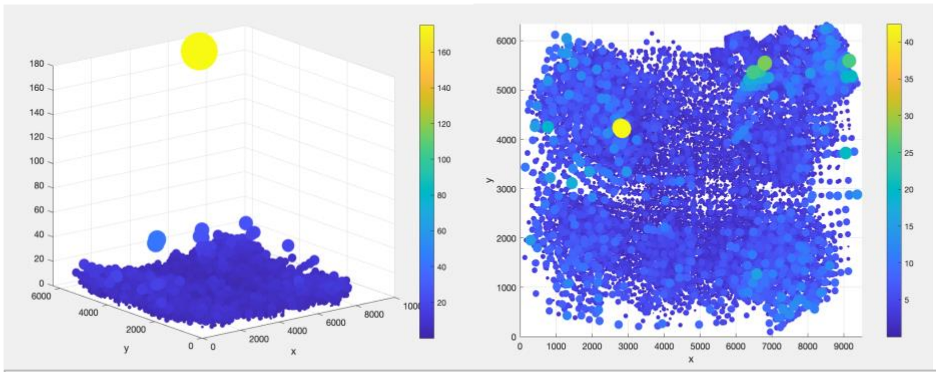
One thing that definitely still needs to be completed would be to run all the detected corners that are used in the camera calibration process through the corner error script to identify how much error there is in each set of corners. Note that MATLAB and OpenCV may use a different corner ordering convention which will need to be adjusted in order for usage in Python. Moreover, it seems that the OpenCV corner detection algorithm cannot handle partial detections, or at least mis ordered blanks, so this will also present a slight annoyance.

Completed Calibrations

Within the folder called **Metal Housing Pool**, there are two additional folders called **Current Analysis** and **Old Analysis**. The files in the current folder were all completed with MATLAB's Fisheye calibration model in the 2022 release version — the high distortion and partial detection parameters were enabled. The folder with the old calibration results consisted of some different parameters enabled as well as calibration completed in OpenCV — the one of interest would be the OpenCV non-planar calibration results.



Rayotek Air



Rayotek Water (Removed the single outlier (left) from the right image)

Fig 1. Reprojection errors of detected corner points for the Rayotek in-air and underwater calibration photo sets

In most cases, it was found that MATLAB's fisheye model outperformed the pinhole model and OpenCV's fisheye model. However, the mathematical equation being used to represent the fisheye model in MATLAB does not perfectly coincide with the standard intrinsic and extrinsic matrices of the pinhole calibration model.

Code and Corner Finding

The code used in my calibration work can be found in my GitHub. My calibration scripts were divided between MATLAB and Python, but mostly focused on MATLAB. The copies of my Python and MATLAB scripts essentially do the same things and follow the basic procedure as outlined by the OpenCV and MATLAB camera calibration tutorials.

In the MATLAB script, **cameraCalib.m** is the main calibration function. The variables from lines 17-24 are the main ones that will need to be changed.

- input_dir: The directory/folder where the calibration photos are located

- data_save_dir: Save directory for written files (i.e. not images or figures)
- save_pth: Save directory for calibration parameters and figures — can make the same as data_save_dir
- read_input: Input file that has all the paths to images that can be used for calibration
- filter_input: Remove select files from the read_input files list
- corner_dir: Folder that contains each used image's detected corner points saved as .xlsx files

The remaining variables should be fairly straightforward to use. If there are any confusions regarding what the corner files look like, take a look at some of the current and old analysis folders. Last note, remember to either pre-create the required save folders or add code to check and then make the folders.

The Python script **calibrateCamera.py** is only the camera parameter estimation portion. The path variables are all intended to be paths to expected directories, or vectors except for the save path. There is a notebook version (**cameracalibration.ipynb**) of the camera parameter estimation that may be a bit more up-to-date and straightforward than the .py file.

The main distinction from the tutorials is the addition of the downsizing corner finder and a selective timeout when performing corner finding. In both MATLAB and OpenCV, I have included functions that will perform the downsized corner finding in the files **map_corners.m** and **downsize_corner_finder.py**. The basic procedure for how both functions work is as follows:

1. First pass to detect the chessboard grid of a particular size
    a. In Python, this process may take a long time so a 10 second timeout feature is implemented
2. Downsize the image by a set factor and maintain the aspect ratio
3. Repeat step 1
4. If the grid is found, crop out the chessboard ROI using the 4 corners identified
5. Perform a chessboard corner detection on the cropped chessboard ROI using the original sized image
6. If step 5 yields no results, then take the results found in step 4 and scale up by the same factor that they were downsized and add the appropriate offset
7. Repeat step 2 if step 4 is unsuccessful

With the downsized corner approach, the process of corner finding becomes noticeably shorter. Whereas a couple hundred of images with different board orientations could take more than a day, this process allows them to be completed in under 4 hours. Below, is an example of the corner finding error using this technique.

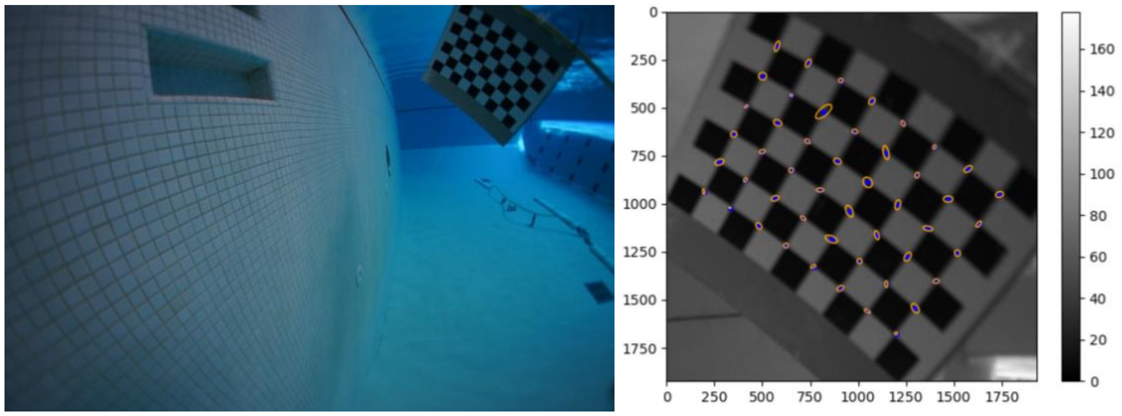


Fig 2. Image with the Rayotek dome (left) and the corner finding error associated with a downsizing factor of 3

From examination of the corner finding, there appear to be a mix of poor and good performing corners. In order to assess how comparable downsizing could be, it would be best to take an image that can be detected with and without the proposed method and see how the results compare. Alternatively, one possible improvement to the corner finding would be to take the detected Harris features from the corner error finding code and assign those to be the corners. The drawback is, again, that it is unclear how accurately Harris feature detection will be on the larger sized images.

Potential things to try in the calibration procedure

One potential solution that could be considered is to change the calibration target. Several papers have proposed that the chessboard calibration target is not the best choice because each corner has only two edges to evaluate gradients on. With only two gradients, it subjects the feature extraction accuracy to much greater errors if distortions are involved. Hence, using a different calibration target could assist in remedying the corner detection error.

Another approach would be to change the camera calibration model. Linked under the resources section are some papers that propose some different calibration toolboxes. Of interest is the generic camera model related calibration. I did not have a chance to test this calibration model due to missing dependencies and incompatible library versions, but I think it would be interesting to see what type of calibration results could be obtained from this toolbox. From the toolbox, I was working on replicating how the paper's authors visualized biases of camera calibration — with this feature, it would enable us to evaluate how much bias was present in the MATLAB and OpenCV calibration and whether a different model would be necessary. My current approach was to plot reprojection errors as a vector and then color code the arrows based on the magnitude of the reprojection error vector.

As mentioned previously, I was not able to completely run all the detected corners through the corner error finding code. It would be interesting to put all the detected corners through the corner error finding to see how large of an error may be introduced through downsizing. Alternatively, through the corner error finding you can modify the script to update detected corners that are above a certain threshold to be the closest detected Harris feature.

Pressure Testing

Overview

The purpose of pressure testing the hemispherical dome ports is to verify that they can withstand the pressure once submerged in the filled detectors. The main concern is that the EzTops acrylic dome will not be able to withstand the increased pressure and fail catastrophically. The group in Winnipeg performed a finite element analysis on the acrylic dome and the objective of pressure testing is to validate their simulations.

My work on the side of pressure testing was to build and then calibrate the test circuit. The circuit design is based on a Wheatstone bridge that Mohit, a postdoc on mPMT, modified. The test circuit that we are using consists of two of the Wheatstone bridge designs connected to one SD card reader. The entire circuit is operated by one Arduino Uno and powered by a Thor Labs power bank.

Circuit Design

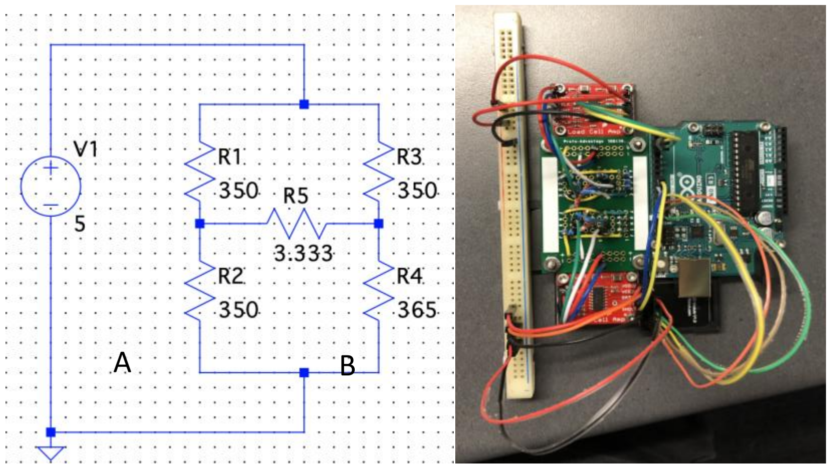


Fig 3. Modified Wheatstone circuit to measure strain gauge resistance (R4) change (Left) and completed circuit design (Right)

The design of the circuit is a modified Wheatstone bridge setup. The reason 350 Ω resistors are chosen is to match the nominal resistance of the strain gauge and create a balanced Wheatstone bridge circuit when no strain is applied. The difference of the bridge circuit in Figure 3 and a normal Wheatstone bridge is that the output terminal AB is shorted when it normally is not. As such, it affects the equation that will be derived for determining output voltage and the strain equation to use.

The complete circuit is shown in Figure 3 on the right. The two red components are the HX711 load cells — an amplifier and ADC converter. The green board between the two load cells is the modified Wheatstone bridge circuit. There are two bridge circuits on the board, one for each of the strain gauges to be connected to the circuit. The 350 Ω resistors used in the Wheatstone circuit are made by a combination of 340 and 10 Ω resistors. The small black part underneath the Arduino Uno is the SD card reader where we log the HX711 readings. Lastly, the breadboard power line on the left is intended to be a common source for power to keep the circuit compact.

While it should not be too great of an issue, make sure that connections are made correctly to ensure the appropriate polarity. The strain-voltage relationship that I derived is based on the output terminal labelling in Figure 3. Make sure that when connecting the strain gauge to the bridge circuit, the red wires are connected to each other, and the blue wire of the bridge is connected to the black of the strain gauge. If this is reversed, I think all that will happen will be a polarity change in the ADC readings which can be corrected during analysis — remember to change the sign of the output voltage and not the ADC reading.

Lastly, the circuit is mounted on one of two variations of a 3D printed board. The screws to be used are M3 sized. The first board variation has through holes where the M3 screws will go through the electronics and boards and are fixed in place with a nut — one of the nuts is not threaded through completely because there were none remaining. The underside of the Wheatstone circuit (small green board in Figure 3) has 3 washers underneath each hole in order to elevate the component for a better fit. This board is the one currently in use. The second variation has holes that are slightly smaller than the required size. For these holes, the same M3 screws can be used but you should tap these holes to avoid using additional nuts; these holes are currently untapped. When placing the electronics components onto the board, place the washers between the green electronics board and the through hole to avoid the fit being too tight.

Calibration Equations

The basic principle behind how the strain gauge works is by measuring changes in resistance. When the strain gauge is stretched or compressed, the change from its nominal length results in a change in the resistance that can be measured with a sensitive enough circuit. The output from the circuit that we use is a discretized level reading from an ADC converter. We first need a conversion from ADC reading to output voltage which is covered in Eq. 2.

$$\frac{Resolution\ of\ the\ ADC}{System\ Voltage} = \frac{ADC\ Reading}{Analog\ Voltage\ Measured} \tag{1}$$

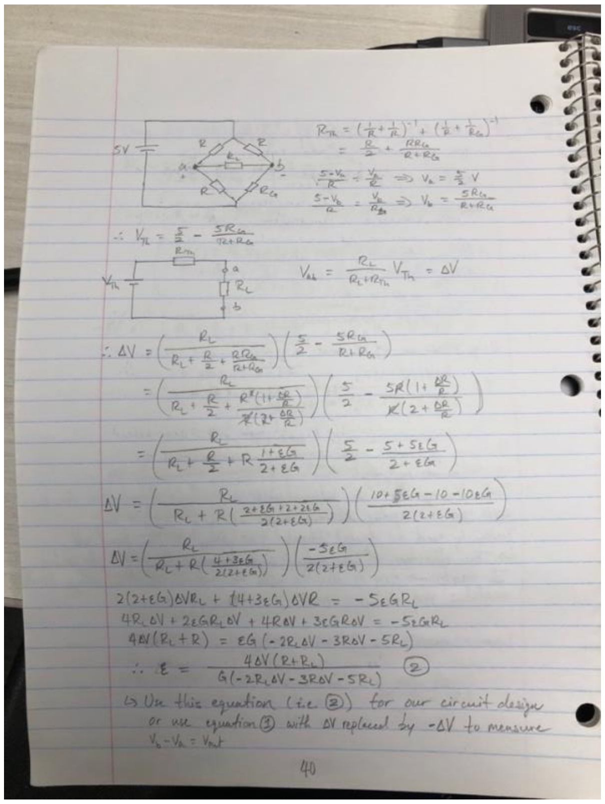
$$\Delta V = \frac{ADC\ Reading}{2^{24}} \times \frac{5}{G_F} \tag{2}$$

The value of G_F will be determined by the gain factor that is used in the circuit. The resolution of the ADC is 24-bits — in 2's complement format — and in our circuit, we have set the AVDD to be 5 V which will map the input differential voltage to between ±2.5 V.

$$\frac{\Delta R}{R_o} = G\epsilon \tag{3}$$

$$\epsilon = \frac{4\Delta V(R+R_L)}{G(-2R_L\Delta V - 3R\Delta V - 5R_L)} \tag{4}$$

Eq. 3 relates resistance change to strain gauge strain. Eq. 4 shows the relationship between the strain and the output voltage ($V_A - V_B$). The derivation can be found below and a link to some related material is at the end.


5V
R
R
R_L
a
b
+
-
R
R_G
R_Th
V_Th
a
R_L
b

$$R_{Th} = \left(\frac{1}{R}+\frac{1}{R}\right)^{-1} + \left(\frac{1}{R}+\frac{1}{R_G}\right)^{-1} = \frac{R}{2} + \frac{RR_G}{R+R_G}$$

$$\frac{5-V_a}{R} = \frac{V_a}{R} \Rightarrow V_a = \frac{5}{2}V$$

$$\frac{5-V_b}{R} = \frac{V_b}{R_G} \Rightarrow V_b = \frac{5R_G}{R+R_G}$$

$$\therefore V_{Th} = \frac{5}{2} - \frac{5R_G}{R+R_G}$$

$$V_{ab} = \frac{R_L}{R_L+R_{Th}} V_{Th} = \Delta V$$

$$\therefore \Delta V = \left(\frac{R_L}{R_L + \frac{R}{2} + \frac{RR_G}{R+R_G}}\right)\left(\frac{5}{2} - \frac{5R_G}{R+R_G}\right)$$

$$= \left(\frac{R_L}{R_L + \frac{R}{2} + \frac{R^{\cancel{2}}(1+\frac{\Delta R}{R})}{\cancel{R}(2+\frac{\Delta R}{R})}}\right)\left(\frac{5}{2} - \frac{5\cancel{R}(1+\frac{\Delta R}{R})}{\cancel{R}(2+\frac{\Delta R}{R})}\right)$$

$$= \left(\frac{R_L}{R_L + \frac{R}{2} + R\frac{1+\varepsilon G}{2+\varepsilon G}}\right)\left(\frac{5}{2} - \frac{5+5\varepsilon G}{2+\varepsilon G}\right)$$

$$\Delta V = \left(\frac{R_L}{R_L + R\left(\frac{2+\varepsilon G+2+2\varepsilon G}{2(2+\varepsilon G)}\right)}\right)\left(\frac{10+5\varepsilon G-10-10\varepsilon G}{2(2+\varepsilon G)}\right)$$

$$\Delta V = \left(\frac{R_L}{R_L + R\left(\frac{4+3\varepsilon G}{2(2+\varepsilon G)}\right)}\right)\left(\frac{-5\varepsilon G}{2(2+\varepsilon G)}\right)$$

$$2(2+\varepsilon G)\Delta V R_L + (4+3\varepsilon G)\Delta V R = -5\varepsilon G R_L$$

$$4R_L\Delta V + 2\varepsilon G R_L \Delta V + 4R\Delta V + 3\varepsilon G R \Delta V = -5\varepsilon G R_L$$

$$4\Delta V(R_L+R) = \varepsilon G(-2R_L\Delta V - 3R\Delta V - 5R_L)$$

$$\therefore \varepsilon = \frac{4\Delta V(R+R_L)}{G(-2R_L\Delta V - 3R\Delta V - 5R_L)} \quad \textcircled{2}$$

↳ Use this equation (i.e. ②) for our circuit design or use equation ① with ΔV replaced by $-\Delta V$ to measure $V_b - V_a = V_{out}$

40

Fig 4. Derivation of the strain-voltage relationship of the Wheatstone bridge circuit

In our pressure tests, we expect that there will be a very small amount of strain and so our acrylic is likely to remain within its elastic regime. Hence, through calibration, we can experimentally determine a linear equation that relates output voltage to the strain — the strain changes in the elastic regime are effectively linear and so mapping the output voltage to strain should remain linear as well since resistance is proportional to the strain.

Strain Circuit Code

There are two test code functions that can be used for the strain circuit: **HX711_Test.ino** and **Strain_Read.ino**. The HX711 test script is mainly to test that the load cell is functional and to verify that the ADC readings make sense and will not saturate. Currently, the main script (**Strain_Read.ino**) is the one that's loaded on the Arduino Uno — this will be the code that you should be making changes to and using during the test as it reads out ADC values to the microSD card. I have added comments to most of the relevant lines, but the basic structure off the code flow is as follows:

1. Initialize microSD card and check that the microSD card is writeable
    a. At the moment, there is no way to visually see this but if an external LED was wired into the circuit an indicator could be given about whether the card can be written
    b. Alternatively, it might be worthwhile to try using the built-in LED (instantiate with **pinMode(LED_BUILTIN, OUTPUT)** in the setup function. This is likely to fail though because the SPI communication utilizes digital pin 13 as the SCLK
2. Initialize the load cells and set gain
3. Get a baseline reading from both load cells (takes the average of 100 readings)
4. Create the data text file. If a file exists, iterate through some number of numbers until an unused name is found
5. Write header, reading time, and zero readings of the load cells to the text file
6. In main loop, continuously read the inputs from the load cells and write to the text file
    a. Currently, the text file writing process is to open, write, and then close the file. However, I'm not sure if it is necessary to close the file each time it's written

Between readings, a 1 s delay is included but the actual delay time between readings will likely be a bit longer due to the time it takes to compute the time of reading and obtain the actual readings.

Calibration Methodology

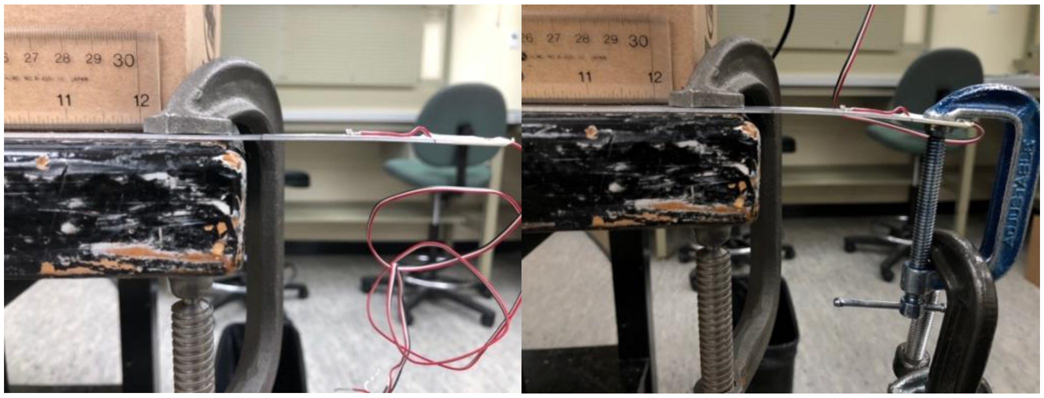


Fig 5. Strained (right) and unstrained (left) images of an aluminium bar

If the circuit needs to be re-calibrated, you can proceed to use the following procedure based on what Mohit and Jacob did:

1. Draw an area of interest on the bar that sandwiches the strain gauge
2. Take a picture of the aluminum bar without any weight applied to it, making sure the area of interest is visible
3. Connect circuit to the bar with no weight and collect readings for about 2 minutes
4. Take a picture of the aluminum bar with enough weight applied to observe a slight deformation with the area of interest visible
5. Connect circuit to the bar with the same weight as step 3 and collect readings for about 2 minutes

After the data has been collected, use an image analysis software like ImageJ or Fiji to fit circles to the section of interest in the bar to compute the strain. Then, using the collected readings and calibration equations compute the strain. If the computed value and measured values are within 500 µm of each other, the circuit should be good to use. Calibration data I collected for gain of 64 and 128 can be found in sheet 2 of the Excel file under the OneDrive's pressure testing folder. The current calibration yielded the following equations for our calibration cells with gain set to 128:

Cell 1 (Without the brown 1/8 W resistors):

$$\Delta V = 1.9052 \times 10^{-9} \times ADC\ Reading + 2.7808 \times 10^{-6}$$
$$\epsilon = -4.1658 \times 10^{7} \times \Delta V + 90.5556$$

Cell 2 (With the brown 1/8 W resistors):

$$\Delta V = 3.4216 \times 10^{-10} \times ADC\ Reading - 3.8288 \times 10^{-5}$$
$$\epsilon = -4.1592 \times 10^{7} \times \Delta V + 102.2675$$

Debugging methods and possible failure modes

In the very unfortunate event that something is wrong, there might be a couple of easy things to check first.

First, check that all the wires connected to the power bar are connected and that there are no loose wires. This will be the easiest to see as the microSD card reader and load cells should light up when they are receiving the correct amount of power.

Next, check the formatting of the microSD card. The microSD is expected to be formatted as — or at least have a partition that is — FAT16 or FAT32. If not, reformat the chip and try again to see if the microSD card can be initialized.

If there is still a problem, remove the bridge circuit and check for faulty connections. It's possible, though unlikely, that the soldering on the series resistors has broken off or failed. In that case, make sure to resolder together the resistors. If any of the connected lines are not actually connected, remove the old solder, and apply new and then check connection again.

If none of the above is working, it might be that one of the components on the circuit has somehow become faulty. The best bet would be to remove that part and re-attach. However, I do not believe that that is likely to happen since previous testing seems to indicate that the circuit itself is functioning properly. Prior to using in the actual pressure testing vessel, set aside the circuit to record values for an entire day to see if everything can be logged without error.

Possible improvements to circuit setup

One potential addition to the circuit is a LED to signal proper initialization of the circuit components. This is implementable with either the analog pins or any leftover digital pins. If the LED is implemented with the analog pins, use analogWrite and choose a value between 0 and 255 to control the LED brightness. The connection is simple and can be completed in the diagram below:

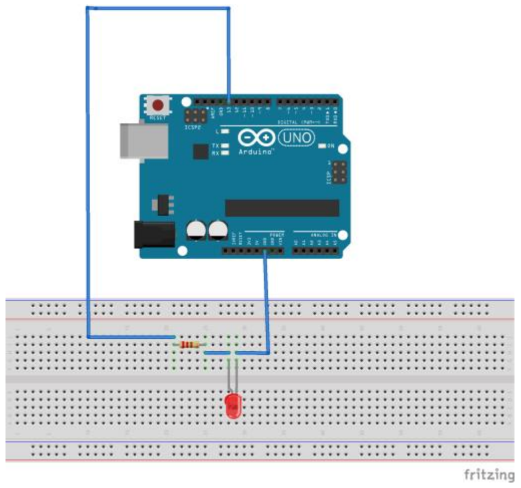


Fig 6. Simple LED Arduino circuit

In the case of the circuit we are using, one could solder the negative end of the LED to the resistor and directly stick the resistor's other terminal into GND and the LED's positive end into any of the analog pins. With the indicator LED, early warning of the circuit's malfunctions can be encoded.

SubC Rayfin Camera

Overview

The SubC Rayfin camera is an underwater deep-sea camera. The intent of the camera is to be used in the Water Cherenkov Test Experiment where it will be mounted on a moveable arm to ensure complete coverage of the inner detector. The Rayfin camera has a wide-angle lens which has less distortion than the fisheye lens making it much easier to calibrate.

The work that was completed on the Rayfin camera during my summer term consisted of testing the hardware and setting up initial interfacing functionality. In the hardware tests, we checked to make sure that initial sensor data (depth, temperature, tilt, roll, yaw, etc.) was accessible and that the camera itself would function correctly. While testing the camera, we discovered that the camera had a faulty depth sensor that needs to be replaced. With regards to interfacing, I started by looking into writing scripts to operate the camera autonomously through the scripting API. The scripting of the camera however was limited in its functionality and would not necessarily fulfill our needs. After a discussion with SubC, an alternate solution to interface through the camera with its serial port was proposed. An initial test script that connects to the Rayfin via TCP/IP protocol with Python was successful — commands such as have the camera take a picture, zero the IMU and start/stop data logging were all executed correctly. Moreover, the script also prints out the immediate one-line response from the Rayfin after a command is executed.

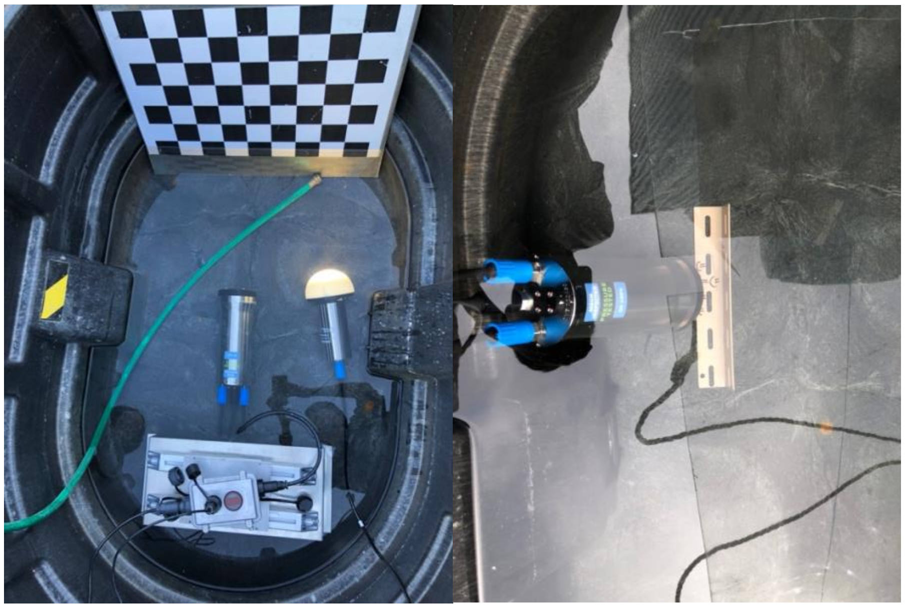


Fig 7. Rayfin camera testing setups

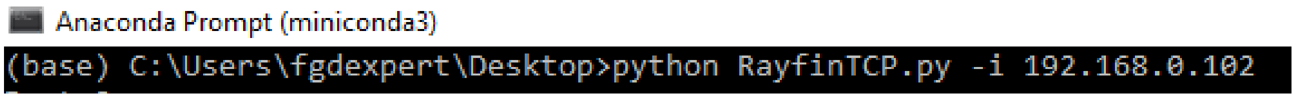
Rayfin Python Test Code

According to the Rayfin manual, the camera can be operated via serial communication. Specifically, the Rayfin features networking support via TCP, UDP and serial — there may be some other types, but we can ignore those for now. For the work that I completed, I chose to stick with only the TCP protocol. The TCP protocol is slower than UDP but is better at arbitrating which data packets were received first — feel free to search up the differences between TCP and UDP. However, it should be relatively easy to switch from TCP to UDP; it may only require a change of the port number from 8888 to 8887.

The code that was used to test the basic interfacing is called **RayfinTCP.py**. When running the function, pass in the IP address of the Rayfin camera as an argument in the command line — see Figure 8 for how to run the file from the command line. It is optional to provide the port number and if no value is provided then the port defaults to 8888. The main function is separated into commented sections, but the basic procedure is as follows:

1. Check input arguments from command line
2. If no IP address is provided, return error, and repeat step 1
3. If an IP address was given, assign the IP address to the variable CAM_IP and assign the port number to TCP_PORT if one was given (otherwise, defaults to 8888)
4. Check if the port is open and can be connected
    a. If it cannot be connected or is closed, return error and repeat step 1
5. Read user input command and find the equivalent Rayfin API command
6. Encode API command in a hexadecimal byte format

7. Connect to the port
8. Send encoded API command to the Rayfin camera
9. Output data read from the camera after step 8
10. Repeat step 5 until ‘q’ or ‘Q’ is inputted



```
Anaconda Prompt (miniconda3)
(base) C:\Users\fgdexpert\Desktop>python RayfinTCP.py -i 192.168.0.102
```

Fig 8. Executing the Rayfin TCP test file from command line

The TCP communication script is very similar to most tutorials that can be found online. However, the main difference is in how the Rayfin API commands are encoded. I have included the link to SubC’s manual for all external communications which is not part of the original manual. In their additional documentation, the encodings of some commands are provided but a few of those appear to be incorrectly written so be careful when inputting their values.

The message encoding needs to be of a hexadecimal byte format. Using the example command ‘TakePicture’, the conversion that is performed in the function *get_TCP_message(message)* is as follows:

1. Encode the input in ASCII format and then convert to a hexadecimal string
    a. ‘TakePicture’ → ‘54616b6550696374757265’
2. Get the length of the input message (not the length of the hex string from step 1)
    a. This is counting the number of characters in the string
    b. ‘TakePicture’ has 11 characters
3. Convert the length of the message to a hexadecimal value and remove the 0x part
    a. 11 in hexadecimal becomes ‘0xb’ and the part of interest to us is ‘b’
4. Pad the string from step 3 if it is only a single value (i.e. the value is between 1-15)
    a. Because the length in hexadecimal is ‘b’, make it into ‘0b’
5. Create the hexadecimal string data packet by combining the length, padding of six 0’s, and the encoded message
    a. The result of this combination is ‘0b00000054616b6550696374757265’
6. Convert the string result in step 5 to byte format
    a. ‘0b00000054616b6550696374757265’ → \x0b\x00\x00\x00TakePicture in byte format

One thing that I noticed while testing this communication method was that the script sometimes would pause until the Rayfin outputted something — it would only wait when the script was reading from the Rayfin. This might be related to how I have setup the test listener and can be avoided by not waiting for the Rayfin to send an output message.

Rayfin API Scripting

The scripting in the Rayfin API is mostly easy to use. Each line of the script is a command that sets some specific camera parameter or takes pictures. A list of all available commands can be found on page 46 of the manual. The structure of the script is then just a sequential order of actions that the user would like the camera to perform. Within the Rayfin scripting language, there are two control key words: 'When' and 'Repeat'. The 'When' key word mimics an 'if' statement in Python and is automatically checked after some interval of time. 'Repeat' is a key word that is similar to a 'while' loop and can be used to write an infinitely looping code. Within each Rayfin script, it is possible to execute other scripts with 'LoadScript', 'ExecuteScript' and then 'StopScript' commands. Below is a sample of a script to take a single picture with the Rayfin.

```
// Take a single photo if the roll, tilt or heading of the camera has changed

AutoFocus
AutoExposure
WaitFor:3000
ManualFocus

When:Roll != 0|TakePicture
When:Tilt != 0|TakePicture
When:Heading != 0|TakePicture

SetIMUZero     // Set current position to be the new starting orientation
```

If a picture taking script was written with the Rayfin API, it would ease the handling when interfacing with Python. There are some advantages to this method. The first is that the parsing time of the Rayfin API commands is known to be 1000 ms which allows for a fixed delay to be set in a Python interface. Secondly, it reduces the number of commands that need to be made available which helps simplify the controls — this is fairly trivial as long as the command encoding format in the Python interface is correct.

FIFISH Drone and Accessories

Overview

The FIFISH Pro v6 Plus drone is the new version of an underwater ROV to be used in the Hyper-K detector for increased coverage of the PMTs lined along the inner wall. Using the

ROV, photos that are collected will also have a known orientation of the drone/camera that will assist with feature matching and the 3D reconstruction process.

The work that was done on the FIFISH drone during the summer was to test its features, test the new locating accessories, and examining the positioning accuracy. The FIFISH Pro v6 Plus has additional built in-features such as an altitude and distance sensor that was not included in the old version of the drone. New built-in features were tested for functionality and as well as compatibility with new public releases of the FIFISH control app. Additional hardware that was received and tested include the Doppler Velocity Logger (DVL) and Underwater GPS (U-GPS) systems.

U-GPS and DVL

The U-GPS and DVL are two acoustic devices used for positioning and navigation of the ROV. Figure 9 below shows the equipment for the U-GPS system and Figure 10 is the DVL.

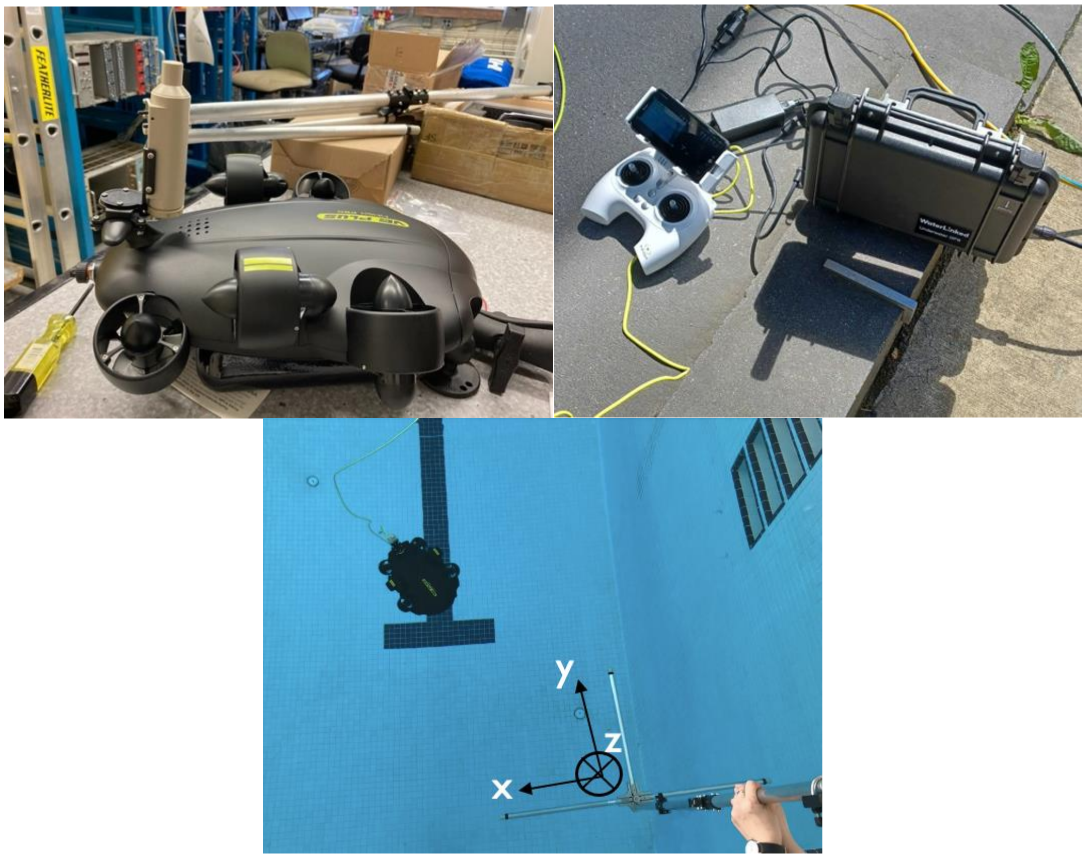


Fig 9. Components of the U-GPS system featuring the locator (Left), topside control unit box (Right) and deployable antenna (Middle)

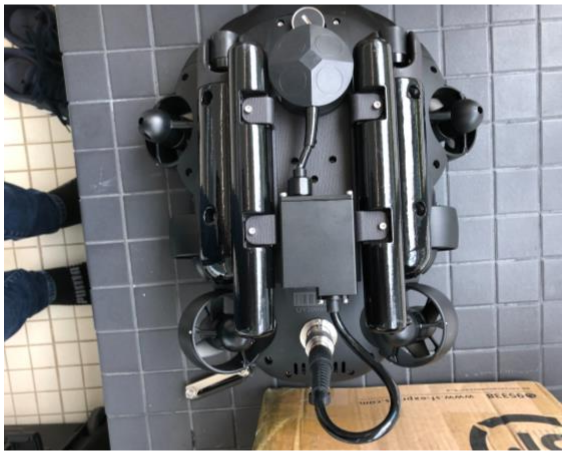
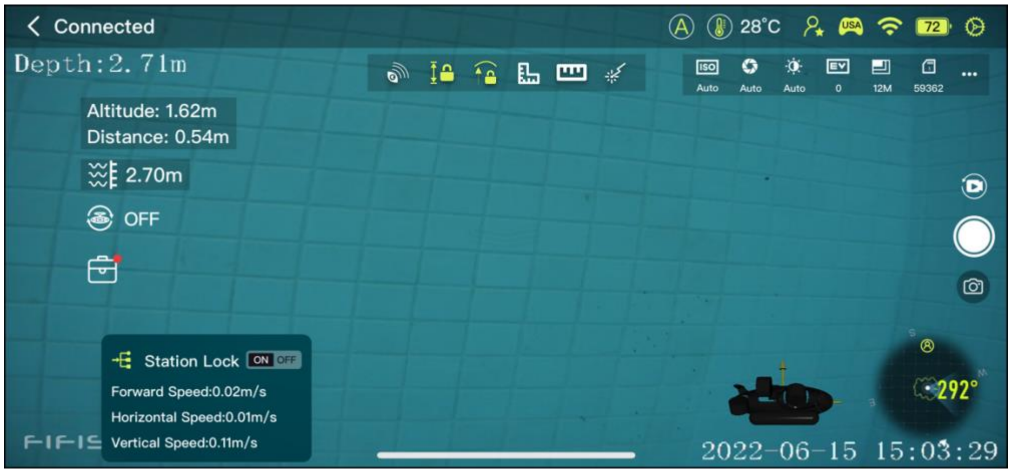


Fig 10. DVL connected to the underside of the ROV

Both devices operate with acoustic signals. A sound wave in the kilohertz region is emitted from the U-GPS locator that is then detected by the receivers on the antenna. The DVL emits a sound wave at 1 MHz that is then received by its own receiver. The U-GPS system is able to find relative positions of the drone with respect to the antenna by triangulating the position according to the time it takes for the acoustic signal to reach each of the antenna's receivers. The locator and topside box units are initially synced but time-drift occurs after some number of hours of usage. To maintain station-locking and physical drifting of the ROV, the DVL should be used. The DVL has 4 transducers which enable it to measure the velocity of the ROV through the Doppler shift of transmitted and received acoustic signal and data from an IMU. The combination of these two devices is the basis for development of an accurate auto-piloting system.

Usage of the DVL and U-GPS are directly integrated with the FIFISH app. When the DVL is connected, the ROV will be able to use the "Station-locking" feature as seen in Figure 11. The U-GPS can be used with the app or through Waterlinked's GUI interface — Waterlinked is the company that designed the U-GPS system and the developers of the FIFISH drone simply integrated the features with their app. If connecting to the Waterlinked GUI, follow the connection instructions on their site (linked in the resources section) and use the data extraction script to save the relative position data to a CSV file. Alternatively, the FIFISH app can enable the U-GPS by selecting the "UQPS" feature. The setup of the connections can be found by following the manual that is linked in the OneDrive. Most of the setup is identical to the instructions on the Waterlinked page and the only additional step is to connect the topside control box's ethernet port to the ROV remote controller's USB mini-B port.


Connected

Depth:2.71m

Altitude: 1.62m

Distance: 0.54m

2.70m

OFF

Station Lock ON OFF

Forward Speed:0.02m/s

Horizontal Speed:0.01m/s

Vertical Speed:0.11m/s

2022-06-15 15:03:29

Fig 11. Station Locking enabled by opening the feature from the toolbox on the left

When testing the U-GPS system's positioning accuracy, we found that the depth readings from the locator were typically inconsistent with the drone's own depth measurements. The setup that was used to test for accuracy in each coordinate was to fix the drone's position in two coordinates (for example, y and z) and then move the drone along at a constant rate in the third direction. The plots should show one coordinate trending linearly and two other coordinates relatively constant. Figure 12 shows some collected data readings. The uncertainty attached with the measurement of the position does account for the fluctuations in the locator's depth readings, but in practice it might be better to rely on the drone's data for depth instead. While the locator values (blue X's) in the y-coordinate of Figure 12 are not as close to the expected as desired, a likely reason could have been that the station locking with the DVL was not enabled. Without station-locking, slight disturbances may have shifted the y-coordinate positioning.

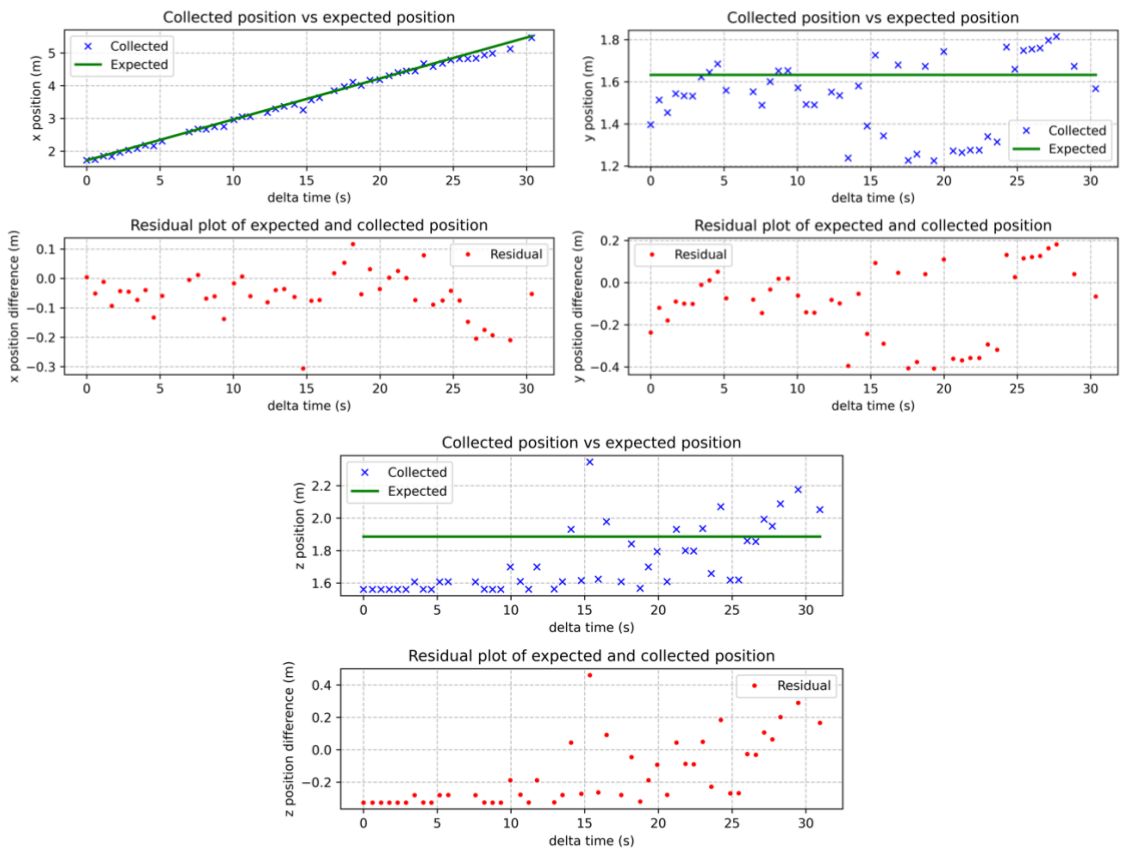
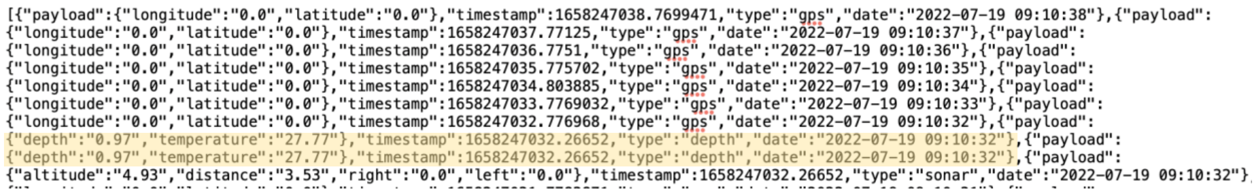


Fig 12. Plots of the drone's x, y and z coordinates and their respective expected values

FIFISH data logging file

In the FIFISH app, the drone's metadata logging file can be downloaded. In the new version of the app, the company has included options for what data to include in the exported log. A new version of the log parsing script was written and is included in the GitHub repository. Some discussion with the company is needed to address potential bugs in their current metadata logs. Figure 13 shows an instance of a repeated entry in the log file which may be a possible issue.



```
[{"payload":{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247038.7699471,"type":"gps","date":"2022-07-19 09:10:38"},{"payload":
{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247037.77125,"type":"gps","date":"2022-07-19 09:10:37"},{"payload":
{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247036.7751,"type":"gps","date":"2022-07-19 09:10:36"},{"payload":
{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247035.775702,"type":"gps","date":"2022-07-19 09:10:35"},{"payload":
{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247034.803885,"type":"gps","date":"2022-07-19 09:10:34"},{"payload":
{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247033.7769032,"type":"gps","date":"2022-07-19 09:10:33"},{"payload":
{"longitude":"0.0","latitude":"0.0"},"timestamp":1658247032.776968,"type":"gps","date":"2022-07-19 09:10:32"},{"payload":
{"depth":"0.97","temperature":"27.77"},"timestamp":1658247032.26652,"type":"depth","date":"2022-07-19 09:10:32"},{"payload":
{"depth":"0.97","temperature":"27.77"},"timestamp":1658247032.26652,"type":"depth","date":"2022-07-19 09:10:32"},{"payload":
{"altitude":"4.93","distance":"3.53","right":"0.0","left":"0.0"},"timestamp":1658247032.26652,"type":"sonar","date":"2022-07-19 09:10:32"}
```

Fig 13. Repeated data entry in metadata log from July 19

The repeated entry could pose problems if the two entries differed in their depth value readings. A workaround would be to average the payload values at the same timestamps, however that might not be the most consistent solution. Another point to clarify would be why the photo type entry in the log is “unknown” when the saved photos are JPEG type.

```
{"payload":{"photo_type":"unknown"},"timestamp":1658246951.3127398,"type":"photo","date":
"2022-07-19 09:09:11"}
```

Miscellaneous Projects

Overview

This section covers some of the other tasks that were completed or started over the summer.

Diffuser Light Clamp

For the LED lamp seen in Figure 7, the diffuser bulb’s clamp was not the right size to fit around the Rayfin camera. An updated clamp model was completed and printed to replace the old one. M2 screws were used for connecting the clamp to the bulb and a M3 screw with nut is used to tighten the clamp. To tighten the clamp, use a wrench to hold the nut in place and then turn the M3 screw. There are two STL file versions of the model: one that has all the holes and one that has only the tightening holes. If the latter model is printed, then a template with all the hole positions can be used to drill in the holes after the print is completed.

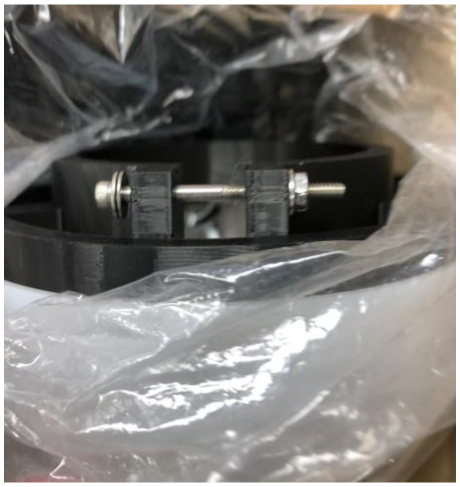


Fig 14. M3 Screw for tightening the clamp

Acrylic Dome Measurements

Dome measurements were made on the EzTops acrylic dome to measure how much of a slant was on the dome flange. The measurements were made with a scanner that sampled some number of points at fixed locations and the output text files have a list of points and their coordinates. The coordinates are defined in some reference frame as determined by the robotic arm gantry that was used to make the measurements. A script was written to process the text files and plot the points, but no further analysis was completed. The scan of the dome was not on a well-balanced surface that would prevent the dome from wobbling and so some of the measurements feel a bit off and inconsistent.

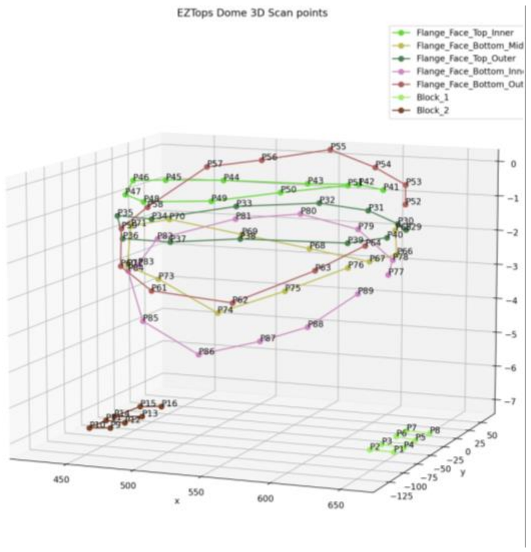


Fig 15. Plotting the points of different scans completed on the EzTops dome

Useful Resources

OneDrive Folder

- Summer 2022 OneDrive Folder

Strain Gauge

- Strain Doc

Camera Calibration

- Checkerboard detection on large images

- Gaps between corners in checkerboard
- Accurate checkerboard detection
- List of calibration related papers with code
    - BabelCalib MATLAB Camera Calibration Toolbox
    - Generic Camera Model calibration
- Deltille Grid Detection
    - Most up-to-date GitHub version
- OpenSFM quality measuring metrics
    - This resource is for structure from motion reconstruction and has several quality measures that should still be applicable to our calibration
- Calibration Quality Metrics
    - Similar to the previous resource
- Inferring Bias and Uncertainty
- Biasing Aspects of planar calibration methods

FIFISH Drone

- FIFISH Manual
- Waterlinked U-GPS Setup instructions
    - Waterlinked U-GPS GitHub
- Waterlinked DVL
    - Waterlinked DVL GitHub
- Waterlinked GUI documentation

Rayfin Camera

- Rayfin Manual
- Rayfin Command API
    - Connecting with TCP/UDP/Serial

Image Sharpness

- Evaluate sharpness of fisheye lens
- Lens sharpness
- Underwater Photography