Hardware Synchronization System

Final Report

Sensory

Ritwik Sathe

Wyatt Hansen

Matt Boyett

Colton Mulkey

Michael Mengistu

Department of Computer Science

Texas A&M University

Date: 12/14/21

Table of Contents

1 Executive summary

The overall goal was to implement an arduino-based system that triggers the collection and synchronization of camera and lidar sensor data. The sensor data collection is currently not triggered and synchronized by a single system, so the autonomous vehicle is not able to detect potential hazards on it's charted path. The objectives are to have the lidar and radar trigger synchronously when a PPS signal is received and to additionally have a data validation program created that detects objects in the camera and lidar and detects the time difference between two frames of data.

The overall project design features several hardware and software components that will work together to timestamp and synchronize sensor data. For the software component, a launch file initiates the camera driver and Rviz, which will allow one to see the image captures in real time. When the PC has received all of these signals as inputs, it will then send off the data, timestamps, and error analysis to memory to be stored. Following this, the data captured will either be manually evaluated to determine the error rate utilizing a ROS-based program we developed.

For the Hardware component, an Arduino program was developed that takes a PPS signal output from the GPS as a serialized input. The GPS is plugged into the input pin so when the PPS signal is detected, a 10 Hz and 1 Hz signal synchronously output from the two output pins (with a 126 ms delay with regards to the PPS input). The 10 Hz signal triggers the camera, which begins running at 10 fps while capturing data. The 1 Hz signal triggers the lidar, which begins capturing data as well. The outputs are also sent to a Ubuntu PC so they can be used as benchmarks for error analysis in comparison to the signals sent to the sensors.

The results of this project yielded an efficient way to synchronize different sensors together. This allows for the synchronized data to be effectively used in a car or robot system which makes every action it takes more accurate and safe. This solution is able to be reused for future projects with different systems, saving both money and time on future robot or car projects. Overall, this system design is necessary for precise and accurate information from multiple sensors and allows the rest of the system to operate significantly better.

2 Project background

This project dealt with making object detection for an autonomous vehicle safer and more responsive. In order to be routed to a specific location, the autonomous vehicle uses GPS technology and proceeds to follow the route. Though the vehicle has various sensors such as a camera and lidar that detect objects in the road and along the road in order to prevent potential collisions, dynamic detection is not always foolproof. This is due to the fact that the lidar and cameras are not communicating with the GPS, so they are essentially not informed about the intended route the vehicle is on. If the sensors communicate with the GPS, they will be able to more accurately detect objects because their focus will be more tunneled in as opposed to wide and spontaneous.

With that in mind, the general scope was to develop an Arduino-based system that uses the Pulse-Per-Second (PPS) signal from a Global Positioning System (GPS) to synchronously trigger camera and lidar sensors in an autonomous vehicle. A PPS signal is what the name suggests: a very short signal with a sharp rising or falling edge that repeats every second; it is then obvious that the frequency is 1 Hertz (1/second). GPS-based time has long been trusted to provide extremely accurate time for device synchronization. Such accuracy is crucial if the devices are to work together to navigate the vehicle. Finally, Robotic Operating System (ROS) development software will synthesize time-stamped data collected from the sensors to minimize inconsistencies between the data. This will ensure proper decision-making by the autopilot system. Several teams are developing the sensors' capabilities, and the success of their work depends on how well the sensors are integrated and synchronized with minimal error. Thus, this division serves as a crucial intermediary between the external signal and the proper data collection.

2.1 Needs statement

The various sensor data collection is currently not triggered and synchronized by a single system, so the autonomous vehicle is not able to detect potential hazards ahead on its path routed by the GPS navigation software.

2.2 Goal and objectives

The goal was to implement a system that synchronously triggers the collection of camera and lidar sensor data. The precision of the synchronized sensor triggering and data collection helps the autonomous vehicle accurately detect the presence of hazards and traffic controls in order to keep its passengers safe. A stretch goal was to implement alerts whenever a sensor stops working or starts malfunctioning so passengers can be notified of the issues and get an alert so that they do not endanger themselves in a car with bad sensors. This stretch goal was not able to be met, but it would be a worthwhile endeavor in future development of the sensor system. Our progress was measured against the following metrics: physical development, validating proper connections between sensors (the lidar being triggered correctly and the camera running at 10 fps) running time, mean and standard deviation of synchronization delay, and synchronization error.

2.3 Design constraints and feasibility

Within the system, all hardware is connected through native serial connections, so data transfer is instantaneous. The only wireless data link is between the GPS and the Arduino framework. The transfer pulses occur on the rising edge of each clock cycle. While the camera and radar sensors can accept 10 Hz pulses, the Velodyne lidar sensors can accept only 1 Hz pulses, so the lidar sensors will have less data to collect. All sensors will still transmit data at 10 Hz to the computer. The main challenge was to program the computer to match the time-stamped data amongst all devices and overcome the data collection time window discrepancy. While PPS time signal outputs may be extended to synchronize at smaller time scales (see 3.4), the ability to achieve this depends on the circuit's susceptibility to clock skews, which is a phenomenon in synchronous digital circuit systems where the same sourced clock signal arrives at different components at different times. The skew is also referred to as the instantaneous difference between the readings of any two clocks. Because of the potential synchronization issues, the vehicle must have a maximum distance between itself and the obstacles to begin data collection and simultaneous data validation while traveling at a certain speed. Basic data validation was necessary just to demonstrate that the hardware is synchronizing properly, while the detailed validations were performed by the other teams that were developing the sensors' capabilities. The circuit had to have minimal power consumption. Finally, since the upgraded sensors were not available in time, the system was developed with the existing hardware in the Engineering department.

2.4 Literature and technical survey

The prior research and development of systems that trigger the collection and synchronization of sensor data is immense. These systems have been needed in a vast number of industries such as Unmanned Aerial Vehicles, Hobbyist projects, and high-speed camera photography. Below are some examples of these systems.

1.1 "Using PPS to Synchronize with External GPS"

1.1.1 This paper details a current product, FLIR's Ladybug5+ Camera, and its ability to directly accept PPS signals and NMEA data streams over the GPIO pins. By accepting the PPS signals, the internal

precision time is continuously synchronized with GPS time and the timestamps of each image frame are within the microseconds of GPS time. The Ladybug API that FLIR developed has two functions that a synchronization system would use to set and get GPS time sync. [6]

1.2 "Time Synchronization for Wireless Sensors Using Low-Cost GPS Module and Arduino"

1.2.1 "The paper presents a new time synchronization method working independently on each node without exchanging time-sync packets among nodes." The authors show that through this method it is faster and more time-efficient. This can be done without building the wireless sensor network with each device. The system goes through a process of time stamping data, and re-samples this data to get time-synchronized data. They found that the time relative errors were found to be within +-400ns, while the time-stamp standard deviation was 40.8ns. [7]

1.3 "Open-Source LiDAR Time Synchronization System by Mimicking GPS-clock"

1.3.1 This paper evaluates the time synchronization of multiple sensors when building a sensor network. In the research paper they use open-source LiDAR to demonstrate how to synchronize multiple other sensors such as cameras, radars, and other types of LiDARS. They also mimic a GPS clock interface to provide a 1 µs synchronization precision for their time synchronization system. This research can be used to help design our project since we need to be able to trigger a Pulse-Per-Second (PPS) signal from a GPS to synchronize multiple cameras, lidar sensors, and radar sensors. [5]

1.4 "Generation of GPS Based Time Signal Outputs for Time Synchronization Application"

1.4.1 This research paper details an experimentation of time signal output from binary frames received from the GPS. It compares PPS output and serial communication output. The hardware was configured to receive both PPS and Serial interrupts and generate NGTS/T and NMEA-format packets according to the signal received. The research concluded that synchronization intervals can be stretched out or shrunk between 1 second and 1 minute. Particularly regarding PPS, the same design can be extended to achieve time precision in milliseconds using NTP and other algorithms. Regarding the proposed project, the research gave a better idea about the PPS time packet output that the system is expected to handle when synchronizing. [8]

1.5 "Brief industry paper: The matter of time — a general and efficient system for precise sensor synchronization in robotic computing"

1.5.1 "The Matter of Time" outlines synchronization systems in robots and the challenges and methods behind them. The article mentions an Intersection over Union (IoU) method which can be "used to determine the threshold of tolerable synchronization error." This would be one useful way to determine how effective our synchronization is. To achieve synchronization, the paper outlines the requirements which are, triggering sensors simultaneously and giving an accurate timestamp to each sensor sample. The difficulties with this being the differences between the sensors triggering mechanisms and time stamping mechanisms. The paper then proposes their solution which is to design specific hardware to trigger the sensors at the same time. Then Systems-on-chip is used to get an accurate timestamp on the data as soon as it exits the sensor. The paper explains that making corrections to data through the robots OS will introduce more variance so their design focuses on using hardware. Overall, this paper describes a problem close to ours and offers up a good solution that could be implemented into our project. [9]

2.5 Evaluation of alternative solutions

- Arduino vs Raspberry Pi

Arduino	Raspberry Pi
● Microcontroller board with just CPU, RAM, ROM, and IO connectivity ● Only need a firmware instructing the hardware what to do ● 16 MHz clock speed ● Traditionally used for for interfacing Sensors and controlling LEDs and Motors ● Digital IO (for digital Input and Output) and Analog IN (analog input) pins ● Micro-USB connection but no power adapter needed (just a USB port) ● Power cuts simply cause a restart with no damage to hardware or software ● Uses Arduino IDE ● Models range from $4-$23	● Microprocessor-based mini computer with a CPU, RAM, storage, graphics driver, on board connectors, and an Operating System ● Need to work with Raspberry Pi OS ● 1.2 GHz clock speed ● Traditionally used for developing software applications w/ Python ● 40-pin GPIO for connecting LEDs, Buttons, Sensors, and Motors ● Micro-USB connection but power adapter needed ● Power cuts can damage hardware and software ● Can use any IDE ● Models range from $35-$75

While Raspberry Pi's are without a doubt more versatile and powerful, using a lightweight, low maintenance hardware such as microcontroller seemed to be the smarter move since it can provide all the functionality needed with high fault tolerance and little need for maintenance.

- Data validation alternatives (manual vs automatic and Pendulum vs other dynamic systems that can be modeled)

  There were two main ways to do data validation either manually or automatically. Doing things manually comes with many drawbacks like the limited amount of data could be analyzed. It would also be subject to human error but it would not take much setup time to start validating data.

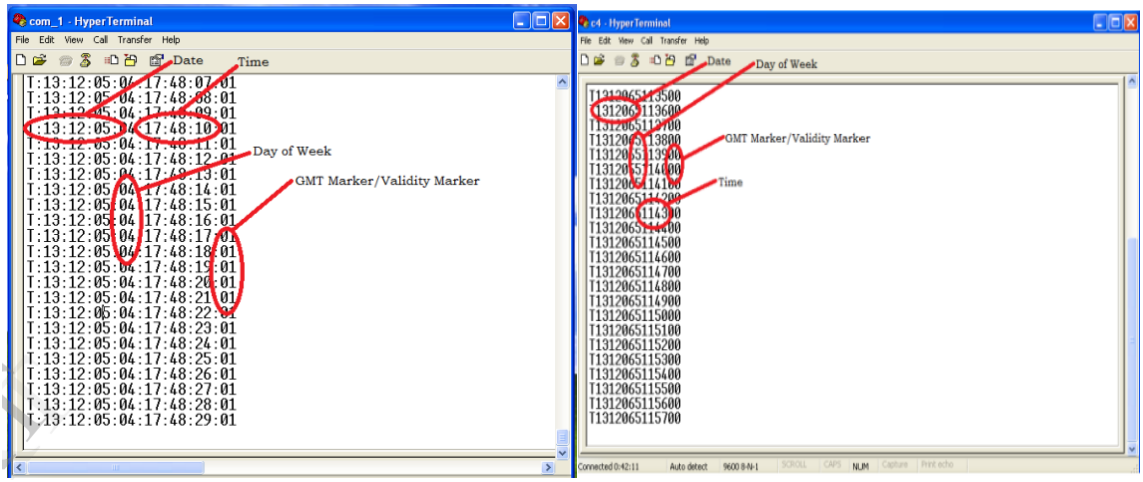
  On the other side of the coin, there was automation. Automating this validation process would have allowed us to analyze a large amount of data and be very precise. This comes at the cost of a large setup time to code an automatic solution. The main way to do this would be to grab an object which moves constantly and then measure the distance between where the object is at the same timestamp between different sensors. A great object for this would be the pendulum, which has consistent movement despite slowing down over time. Overall, automating the data validation process would have been much better than manual but it would have taken a longer time to set up and implement.

- Timestamp Validation

  1. Since the external signal is a GPS PPS signal, the timestamps may be output in one of two formats: National Grid Technical Standard (NGTS) frame or T frame. The following

pictures comparing the two kinds of outputs are from the research paper for which the timestamp technical survey was based on:

T format [8] NGTS format [8]

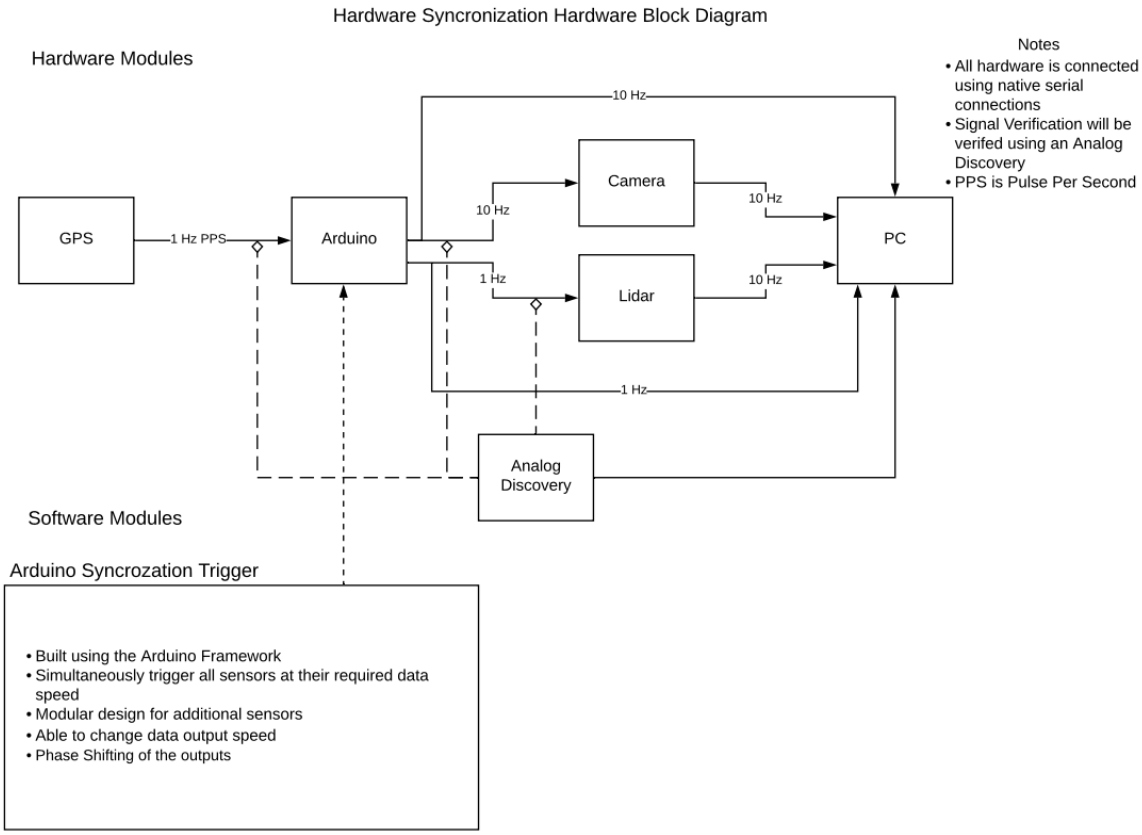


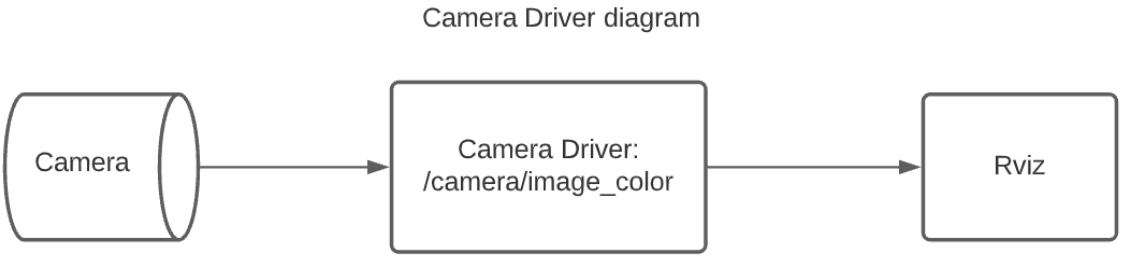
The T format is more human-readable and therefore more easily parsable, than the NGTS format. The NGTS format, however, is far more compact, so it transmits more efficiently.

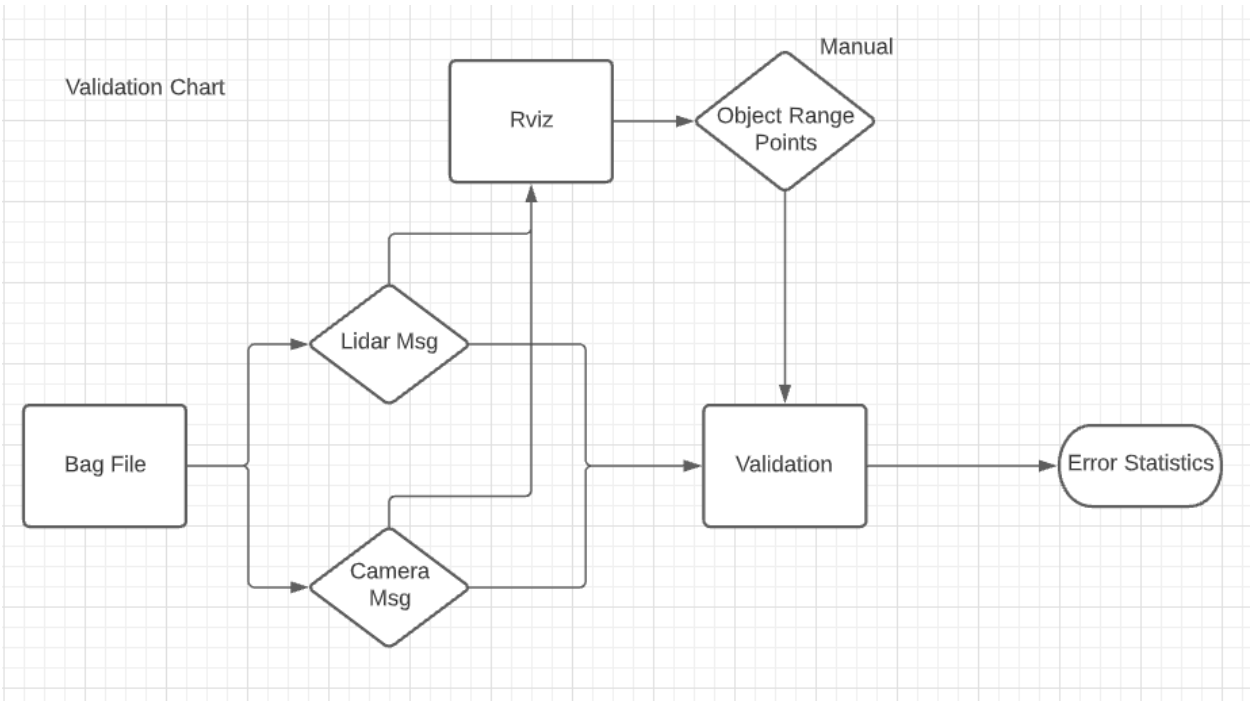
3 Final design

3.1 System description

Our design for the Hardware Synchronization is illustrated in the block diagram below. The diagram is split into two sections, the Hardware and Software. Each of the Hardware and Software modules will be described in detail, explaining what the purpose for each is and why we need it in our design.

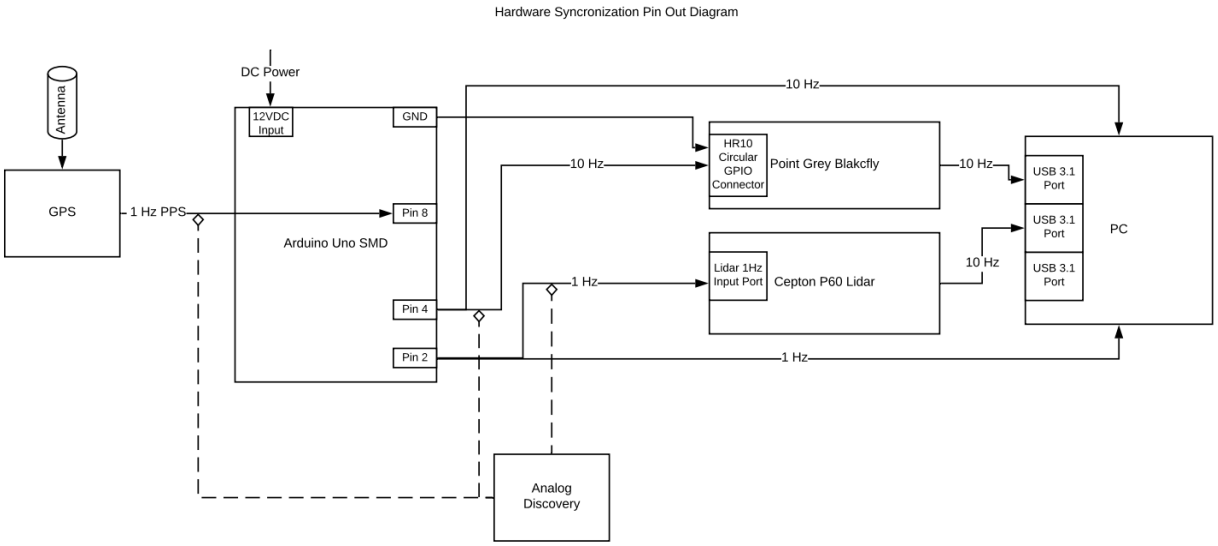

Hardware Syncronization Hardware Block Diagram
Hardware Modules
Notes
• All hardware is connected using native serial connections
• Signal Verification will be verifed using an Analog Discovery
• PPS is Pulse Per Second
10 Hz
GPS
1 Hz PPS
Arduino
10 Hz
Camera
10 Hz
PC
1 Hz
Lidar
10 Hz
1 Hz
Analog Discovery
Software Modules
Arduino Syncrozation Trigger
• Built using the Arduino Framework
• Simultaneously trigger all sensors at their required data speed
• Modular design for additional sensors
• Able to change data output speed
• Phase Shifting of the outputs


Camera Driver diagram
Camera
Camera Driver:
/camera/image_color
Rviz



3.2 Complete module-wise specifications

3.2.1 Pin-Out Diagram



2. 3.2.2 Arduino Synchronization Trigger

This program lives on an Arduino in between the GPS and the sensors. The input of the Arduino is the PPS 1 Hz Rising Edge Signal from the GPS. The output is a 10 Hz and 1 Hz signal to the appropriate sensor and also to the PC. The Arduino is also programmed to be modular to allow more sensors to connect to the system that accept either a 1 Hz or 10 Hz signal.

3. 3.2.3 PC Data Error Analysis

The PC receives the 10 Hz and 1hz signal so it can accurately use it to timestamp the data. The program is coded in C and is parallelized to reduce the amount of time it takes until the data is stamped. The program will also be optimized to run as quickly as possible so that the data is time stamped as soon as possible. The PC will receive the sensor data and using the 10 and 1hz signal it will time stamp them and then it will store it in memory to be used later.

To validate the data, a program is used to calculate the error between the sensors. This program will receive the output data with the timestamps. The program will parse the timestamps then compare all the sensors at the same timestamp. This program will first align the sensor data using the background as reference. The program will then search each image for the spherical object used and then compare its height compared to the other sensors. From the difference in height the time difference between images can be calculated. This is run through all the data and then the average time error is calculated to determine how accurate our hardware synchronization is.

4. 3.2.4 GPS

The GPS is a Reach M2. The GPS has a highly reliable Atomic Clock inside of it. This Atomic Clock is what we will use to generate the Pulse Per Second (PPS) over a serial connection to the Arduino. The PPS coming from the GPS is a Rising Edge Signal that is generated at 1 Hz.

5. 3.2.5 Arduino

The Arduino will take an input of the PPS from the GPS. We will have a program nested in the Arduino to generate the Outputs of a 10 Hz and 1 Hz Rising Edge Signal. These outputs are serial connections to the sensors. The Arduino is wired to each sensor by using the pin connectors to connect to each of the sensors' proprietary ports.

6. 3.2.6 Camera

The Camera is a FLIR PointGrey Camera. There might be multiple cameras but they will all operate the same way. The camera will take in a 10 Hz Rising Edge Signal and output the data over a serial connection at 10 Hz as well.

7. 3.2.7 Lidar

The Lidar is a Velodyne VLP-16 Lidar. The Lidar has a serial connection input of a 1 Hz Rising Edge Signal and an output of its data at 10 Hz. The Lidar is the only sensor that has an input of 1 Hz.

8. 3.2.8 PC

The PC runs Ubuntu 18.04. Each of the sensors will then transmit their data to the PC over the serial connections. Each of the inputs to the PC are at a 10 Hz rate.

3.2.9 ROS framework

ROS melodic is the software framework used for the development of the software component.

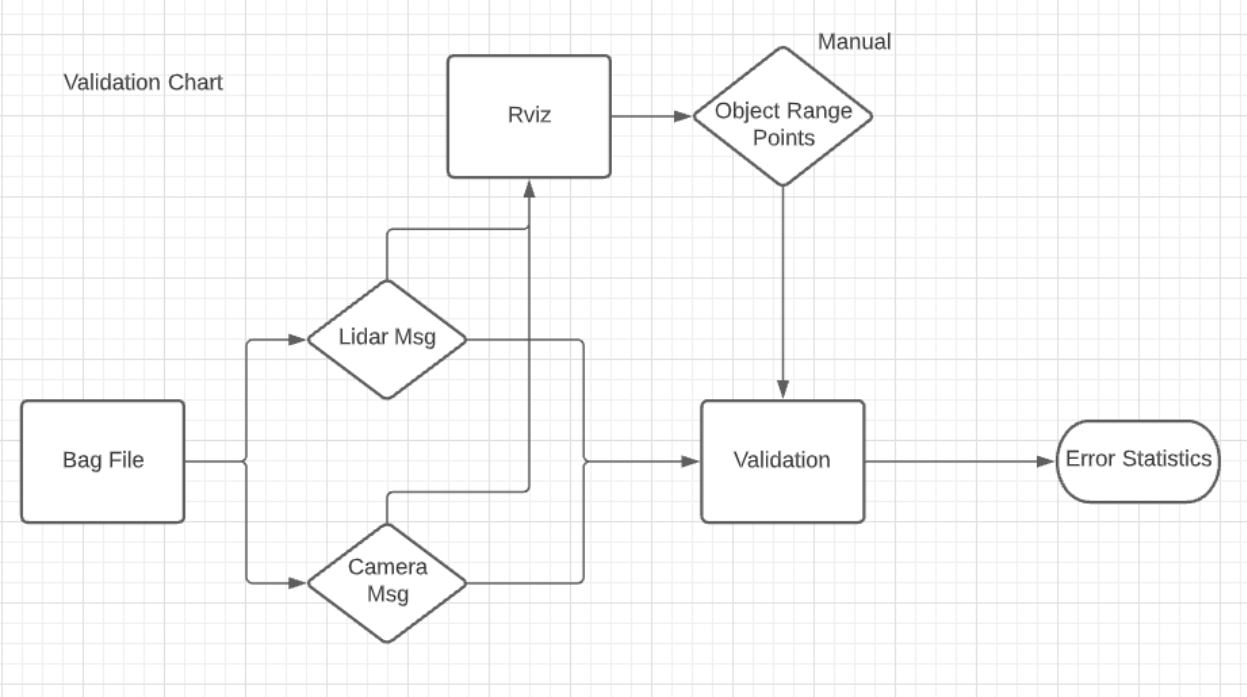
3.2.10 FlyCapture SDK

Software development kit that initiates the camera's image captures and transmission of data to the PC.

3.2.10 Camera Driver/Rviz launch file

The launch file initiates the PointGrey camera driver and Rviz. The camera driver receives raw image data from the camera, converts the data into ROS image messages, and publishes them to a ROS topic. On Rviz, the user then subscribes to the ROS topic containing the image messages.

3.3 Approach for design validation



The stated need for this project is to create a system which synchronises the data received from the sensors, so that the error of time between sensors is not above a certain value. To measure this error, a program was written to determine the difference between sensors at the same timestamp.

To set up this validation the first step is to collect specific data. Data is collected by dropping a large identifiable object from a measured height and distance from the sensors. For this validation a basketball is used since it is large enough so that it will not be missed by the lidar. In addition, a basketball is a sphere and is easily identified with image recognition software. For the setup, the environment must have different colors than the object used and the object should be far from the wall. This makes sure to distinguish the object in the lidar and also prevents shadows from showing up in the camera detection. Once the object is set up it is dropped from a height and the sensors will record the data. This data will then be time stamped by our system and then sent to the validation program.

This program is responsible for measuring how synced the system is and will compare the sensor data. To determine how accurately the sensors are synced the basketball is used as reference. The program runs through each frame in a certain time range for both the lidar and camera and then it compares the two. It first gets the y position in the camera and then the y position in lidar. The lidar gets this by looking through a bounded range for points and taking the average y. The camera achieves this by looking for the right hue value and taking the average y position. It then converts from pixel coordinates to meters using the height dropped. After this, the time difference is calculated between the two using physics equations. Then the error is calculated by accounting for the time difference labeled. This process is run through a specified time range and then the mean and standard deviation are calculated and displayed.

The advantages of this method is the ease of setup and relatively simple calculations to determine the time difference between sensors. The disadvantage of this method though is that it is unable to measure the error over long periods of time since the basketball will not fall forever. There are also some inputs that must be manually found like the lidar range so it is not fully automatic. Overall, this method should be sufficient for validating the system design for the syncing of multiple sensors.

4 Implementation notes

In this section we will talk about the project and each functional aspect of it to such a detail that it would allow a skilled engineer to understand, reproduce, and modify our project. We will include any explanations of previous terminology, links that we used to reference any of the hardware and explain how they work together.

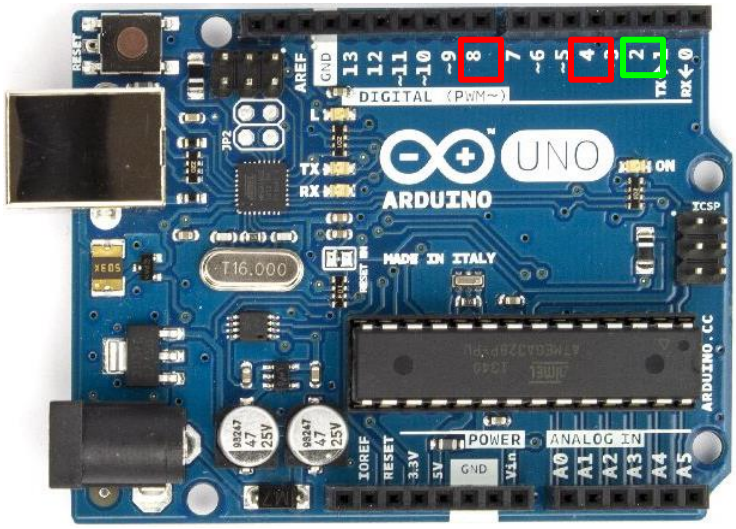
4. 1 Hardware:

The hardware was split into three sections of work: the Hardware Connections, the Arduino Code, and the Analog Discovery for Hardware Validation. Each of these sections are a key component of the project working but are separate implementations that need to be able to work together to get the desired output. These sections are talked about in detail below.

4. 2 Hardware Connections:

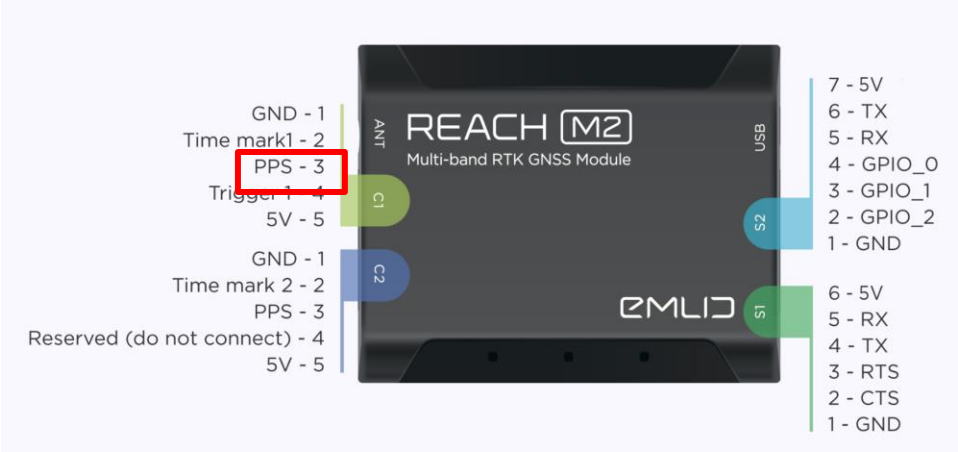
To set up and wire the entire system that receives the PPS input and outputs a 1 Hz and 10 Hz signal to the lidar and camera respectively, you first need to take inventory of the required components. Though we were not able to show the lidar component of the system in our demo, I will still detail what to connect if a lidar is present.

- Arduino UNO SMD and its USB Type-B power cable



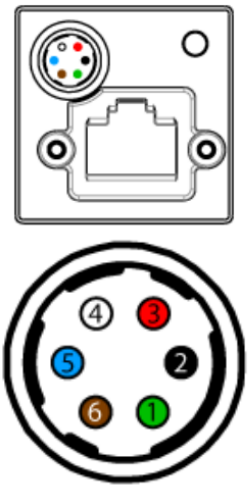
Green for input pin and Red for output pins

- A breadboard
- 8 jumper wires
- The Reach M2 GPS, its Micro-USB cable (for power), its C1 5-cable connector, and its antenna



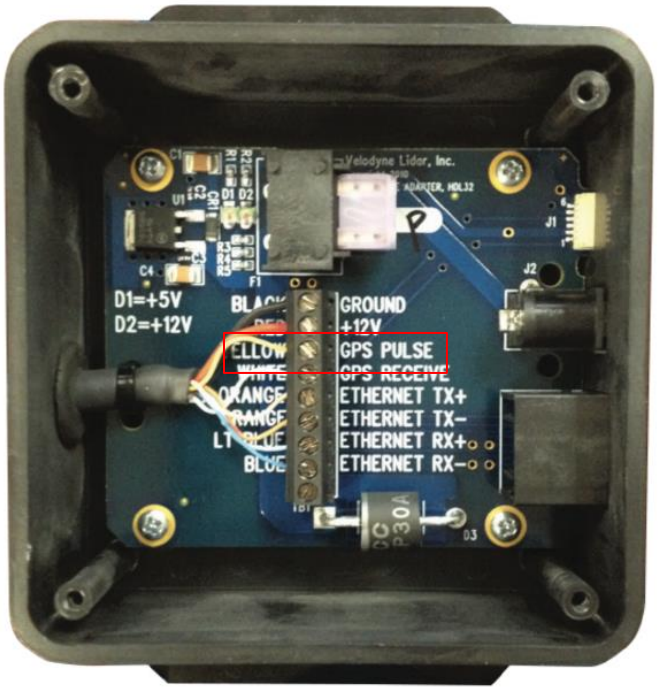
Pin-out diagram for C1 port

- A smartphone with the ReachView 3 Smartphone app installed
  - Apple App Store Link
  - Google Play Store Link
- The Flir PointGrey Blackfly camera, its USB 3.0 connector, and it's GPIO connector

Diagram	Color	Pin	Function	Description
	Green	1	Power	+12 V DC Camera Power
	Black	2	Opto Input 1	Opto-isolated input
	Red	3	NC / +3.3 V	+3.3 V output. Current 120 mA (nominal). Firmware enabled
	White	4	Opto Output 1	Opto-isolated output
	Blue	5	Opto GND	Ground for opto-isolated I/O, not connected to camera ground
	Brown	6	GND	DC camera power ground
To configure the GPIO pins, consult the General Purpose Input/Output section of your camera's Technical Reference Manual.				

GPIO connector Pin-Out Diagram for reference

- The Velodyne VLP P16 Lidar



Velodyne Connection Box

- The Ubuntu PC (which will be running the software).
- Surge Protector (optional but recommended)

NOTE: In order for the system to function correctly, you need to ensure that you are OUTSIDE for the GPS antenna to connect to a satellite and output a 1 Hz PPS signal. The antenna needs to be held upright and pointed directly to the sky.

1. With these parts in hand, you need to first plug the USB type-B power cable into the arduino and connect the cable to a power brick that can be plugged into a wall outlet or surge protector connected to a wall outlet (preferably). Wait 1 minute for the Arduino to power on (indicated by the green and orange LEDs) before moving to the next step.
2. Next, if the orange LED is displaying (which means the program is running), connect a jumper wire to the end of two separate rows of pinholes on the breadboard and connect another jumper wire from the GND pinhole on the Arduino toto the blue "-", or ground, column on the breadboard.
3. Following this, plug a wire into pin 2 (1 Hz output) and pin 4 (10 Hz output).
    a. Proceed to plug two jumper wires into the 1 Hz row and one jumper wire into the 10 Hz row.
4. Connect the open wire on the pin 2 row to the lidar input port and then plug the lidar into a power source to power it on.
5. Before hooking up the camera, be sure to correctly wire the camera's GPIO connector by connecting the BLUE wire to the ground column and the BLACK wire to the 10 Hz row on the breadboard.
    a. Now, plug the camera's USB 3.0 cable into a power source of your choosing.
    b. Next, connect the GPIO cable to the corresponding port on the camera.
6. To set up the GPS you need to do several things:
    a. First, connect the GPS's micro-USB cable to a power supply of your choosing.
    b. Connect the antenna to the GPS and connect the C1 cable to the corresponding port.

c. Now, connect a jumper wire to pin 8 and connect the other end to the BLUE wire from C1 cable.
d. Install the ReachView 3 app onto your smartphone and follow the setup instructions.
    i. In the App settings, be sure to change the coordinate input mode to “average single”
    ii. Also change the position streaming setting to “serial”
    iii. Link to setup guide for these particular settings

7. Lastly, connect the remaining 1 Hz and 10 Hz jumper wires to the Ubuntu PC for error validation.
    a. This can be done by splicing two USB cables and connecting the jumper wires.

Final product (breadboard and lidar not including since this was setup was for testing the PPS triggering functionality)

4. 3 Arduino code and waveform generations

The Arduino IDE is what we used to be able to accomplish detecting the incoming PPS signal from the GPS, and sending a synchronized 10 Hertz and 1 Hertz signal output. The entire development process of the Arduino code was documented on the class Github page under the name of “Arduino10HZ1HZ.” The development of the code started with achieving the output of a 10 Hertz signal. This was done by utilizing the inputted frequency and duty cycle for 10 Hertz, which was set to 10 (10 Hertz) and 50 (50% Duty Cycle). The “void loop()” function was utilized to create a never ending loop that would take in the current time from the internal clock and calculate the length of the 10 Hertz period of being On and Off. It would then sequentially Digital write the Output pin to be high and then delay the program for the allotted

time and then Digital write the Output pin to be Low and then delay the program for the allotted time. What this achieves is a programmable output based on the desired frequency and duty cycle.

The second version of the "Arduino10HZ1HZ" was used to implement the 1 Hertz synchronous output. To do this we added a global variable called "Count." The idea behind this was that we can just count the number of rising edges from the 10 Hertz output and know when to turn on and off the 1 Hertz. This works because by definition of the 10 Hertz and 1 Hertz we know that there should be 10 rising edges in the 10 Hertz for every 1 rising edge in the 1 Hertz outputs. Doing this we were able to get a 1 Hertz output.

The third version of the "Arduino10HZ1HZ" was used to implement the phase shift of the 1 Hertz compared to the 10 Hertz. The point of a phase shift is to delay the 1 Hertz output to achieve better data synchronization from the sensor's internal data processing. To achieve this we made new variables that gave a better description of the desired 1 Hertz signal such as: onehzPeriod, onehzDutyCycle, and onehzPhaseShift. "onehzPeriod" is the full count cycle of the 10 Hertz, so by default its 10. "onehzDutyCycle" is the desired count for how long to keep the 1 Hertz signal High, so by default we have a 50% duty cycle so the input is 5. "onehzPhaseShift" is the desired delay from the start of the 10 Hertz output, so by default it should be 0 but this can be changed as desired.

The last version of the "Arduino10HZ1HZ" arduino script was used to implement the input from the GPS pulse-per-second. This was done by using another input pin on the arduino, by default we used pin 8. We realized that the start of the file changes to looping to check if the input pin is reading High and then we trigger the method for outputting the 10 and 1 Hertz outputs. This allows us to stay in sync with the input GPS PPS and still have the desired 10 and Hertz be outputted at a synchronous speed.

4. 4 Analog Discovery for Hardware Validation

To verify that the outputs that we were getting are synchronized and as expected we used an Analog Discovery for an Oscilloscope. The first attempts for checking the outputs were to use serial prints inside the Arduino code. This raised issues with the timing due to how slow the serial output is with Arduino code. Using the Analog Discovery allows us to analyze the output without affecting the timing and process of the Arduino program. To use the Analog Discovery we used a program called Waveforms to achieve the outputs that are seen in the next section.

4. 5 Camera driver implementation:

The launch file consists of two nodes. Each node is one line of shell code that runs forever loops, which is why each line is in a separate file.

One node launches the PointGrey camera driver. The other node launches Rviz.

The camera first needs to be started up on flycap so that it can capture images. Then the USB core memory needs to be expanded to at least 1 MB for the camera to transmit the raw data to the camera driver, which can then convert the data into ROS image messages and publish them onto /camera topics.

One then configures Rviz to subscribe to the /camera/image_color topic and can display the image captures in real time.

See "ROS Camera launch file" in Section 6 on how to configure the USB core memory, turn on the camera, start up the launch file, and configure Rviz.

4. 6 Data Validation:

Gravity Physics -

The calculations for Gravity Physics are contained in GravityPhysics.py and it consists of three methods, getTime, timeDiff and nextFallPoint. **getTime returns** the time it takes to fall a distance d assuming that the object started at rest. **timeDiff** takes in two such distances and then **returns** the difference of time between them. Finally, **nextFallPoint** takes a distance fallen and then a time interval as input. It **returns** the expected distance that the object should have fallen after the time interval timeI. So if an object would have fallen an additional 1 meter in a time interval of 2 seconds then this function would return the dist + 1 where dist is how far the object had already fallen.

```
from math import *
def getTime(d):
    # gets time it takes to fall distance d
    return sqrt(d * 2 / 9.8)

def timeDiff(h1, h2):
    # takes in two distances(in meters) an object has fallen and returns the time difference between the two
    #print h1,h2
    t1 = getTime(h1)
    t2 = getTime(h2)
    return abs(t2 - t1)

def nextFallPoint(dist, timeI):
    # gets the next point an object should be back based on distance fallen dist and the time interval timeI
    t = getTime(dist)
    t += timeI
    return 0.5 * 9.8 * t ** 2
```

Distance Fallen

Height Fallen

Height Fallen

Time Interval

Camera and Lidar Detection -

Lidar Detection

The Lidar detection is done in lidarFindObj which takes in an array pc (same array pypcd library gives you from a lidar message), the ranges to look for the object and the dropH (drop height in lidar cords). This program will loop through all the lidar cords and take the average z value within the range specified. The z value is calculated by subtracting the drop Height from the z value (this gets the height dropped). This function will also return -1 if no values were found.

Example usage: lidarFindObj(pc, 0, 5, 1, 2, 0, 5, 4)

This will **return** the average z value with a drop height of 4 in the x range from 0-5, y range 1-2 and z range 0-5

```
from GravityPhysics import *
def lidarFindObj(pc, x0, x1, y0, y1, z0, z1, dropH):
        #returns average z value of an object (where z is the height dropped in point cloud cordinates)
        zSum = 0
        count = 0
        for x, y, z in zip(pc.pc_data['x'], pc.pc_data['y'], pc.pc_data['z']):
                if (x >= x0 and x <= x1) and (y >= y0 and y <= y1) and (z >= z0 and z <= z1):
                    zSum += (dropH-z)
                    count += 1
        if (count == 0):
                return -1 #returns -1 if no data was found in specified range
        return (zSum/count)
```


Camera Detection

Usage: cameraAvgY(image,hue0,hue1, sat0, sat1, lum0, lum1)

This function **returns** the average y coordinate pixel based on hue, sat and luminosity range. This function will also take in an image in HSV mode (in opencv cv2.COLOR_BGR2HSV). **Returns** -1 if nothing is found in that range.

Usage: cameraFindObj(image, groundH, dropH, bHeight, hue0 = 0, hue1 = 10, sat0 = 20, sat1 = 220, lum0 = 20, lum1 = 220)

Takes in the pixel height for the ground, the pixel height for the drop of the object and the Total Height dropped in meters. It also optionally takes in the hue, sat and luminosity ranges to find the object. It also takes in an image file and resizes it to 1008,756 to reduce the run time of the function. It **returns** -1 if nothing is found and **returns** the height dropped in meters if an object is found.

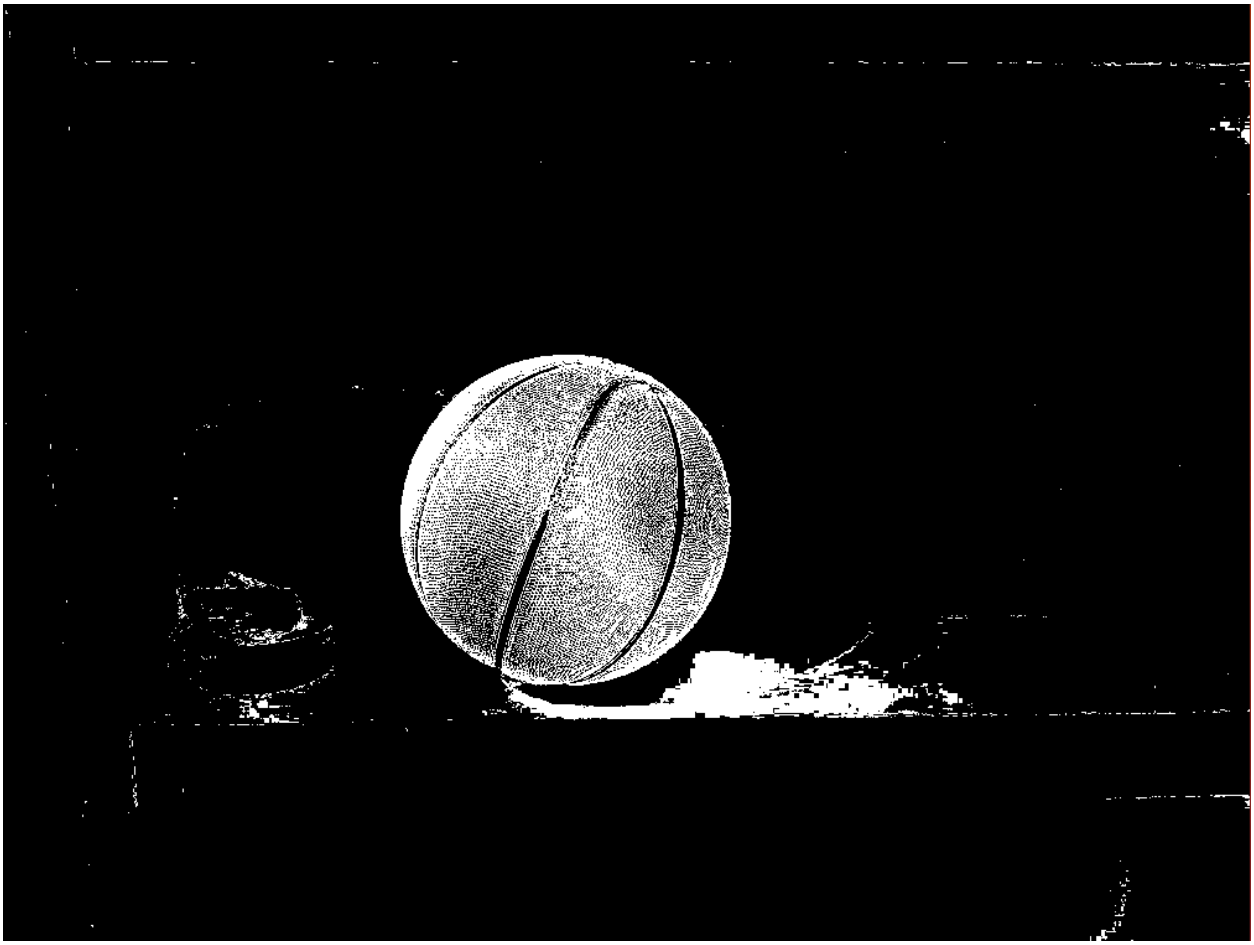
```
def cameraAvgY(image, hue0, hue1, sat0, sat1, lum0, lum1):
        #returns the average y coordinate of pixels in the image which are in the hue range [hue0,hue1]
        avg = 0
        count = 0
        for y in range(image.shape[0]):
                for x in range(image.shape[1]):
                    (h, s, v) = image[y, x]
                    # check for basketball color
                    if (h >= hue0 and h <= hue1): #and s > 10 and s < 220 and v > 20 and v < 200):
                        avg += y
                        count += 1
        if(count == 0):
                return -1 #returns -1 if nothing is found
        return avg / float(count)

def cameraFindObj(image, groundH, dropH, bHeight, hue0 = 0, hue1 = 10, sat0 = 20, sat1 = 220, lum0 = 20, lum1 = 220):
        width = 1008
        height = 756
        image = cv2.resize(image, (width, height))
        image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        #get average y coord of object based off hue color
        avg = cameraAvgY(image,hue0,hue1, sat0, sat1, lum0, lum1)
        if(avg == -1):
                return -1 #return -1 when nothing was found

        diff = dropH - avg
        pixelHeight = dropH - groundH
        percentage = diff / float(pixelHeight)
        realHeight = bHeight * percentage

        return realHeight
```

Showing Camera Detection -

ex. Usage: python showCamDetect.py imageFile hueLow hueHigh

This script uses CameraDetection.py's cameraHueView function which is the same as cameraAvgY except cameraHueView will show the processed image and show it to the user. Takes the image file as an input as well as the hue range to search for.

Ex output-



Decodebag -

Ex Usage: python decodebag.py bagFile timeLow timeHigh

Outputs mean and standard deviation of error within specified time range.

https://pypi.org/project/pypcd/ -pypcd library used

decodebag.py takes in the bagfile and the time range in the messages to use for error calculation.

This program makes use of several libraries including pypcd for point clouds, openCV for images, rosbag for reading messages and a few others. The program loops through all the messages for the topics front camera and lidar_left and then checks if they're in the specified time range. Then it calls the respective find object function and then stores the y position and time stamp in the corresponding array. Finally all the data is run through the timeDiff function (ignoring data which found nothing) and the error is calculated. Finally the mean and standard deviation are calculated based on the error found earlier.

```
from GravityPhysics import *

def validate(bagFile, timeA, timeB):
        #takes in bag file and the time range at which the error will be calculated
        t0 = rospy.Time(timeA).to_sec()
        t1 = rospy.Time(timeB).to_sec()
        bag = rosbag.Bag(bagFile)

        img_msgs =  bag.read_messages(topics=['/front_camera/image_raw', '/lidar_left/velodyne_points'])

        bridge = CvBridge()

        lidarH = []
        cameraH = []
        for topic, msg, t in img_msgs:
                if t.to_sec() >= t0 and t.to_sec() <= t1 and topic == '/front_camera/image_raw':
                        cv_image = bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
                        cv_image = cv2.resize(cv_image, (800, 600))
                        #gets camera y pos (using ground pixel position and pixel drop height and height droped in meters) then hue values
                        var = (cd.cameraFindObj(cv_image, 250, 450, 2, 150, 170), t.to_sec())
                        cameraH.append(var)
                if topic == '/lidar_left/velodyne_points' and t.to_sec() >= t0 and t.to_sec() <= t1:
                        pc = pypcd.PointCloud.from_msg(msg)
                        #gets lidar with bouding cords x, y, z (z should be height) and then the height droped from (in z cords)
                        var = (cd.lidarFindObj(pc, 6.85, 6.95,3.70, 3.95,-2.3,0,0),t.to_sec())
                        lidarH.append(var)

        bag.close()

        timeD = np.zeros(min(len(lidarH), len(cameraH)))
        i = 0
        #print lidarH
        for x,y in zip(lidarH,cameraH):
                if(x[0] == -1 or y[0] == -1):
                        continue #skips any data where an object was not found
                diff = timeDiff(x[0],y[0])

                #accounts for the difference in the labeled timestamps
                labeledDiff = abs(x[1] - y[1])
                diff = abs(diff - labeledDiff)

                timeD[i] = diff
                i+=1
        print("mean: ", np.mean(timeD))
        print("standard deviation: ", np.std(timeD))
```

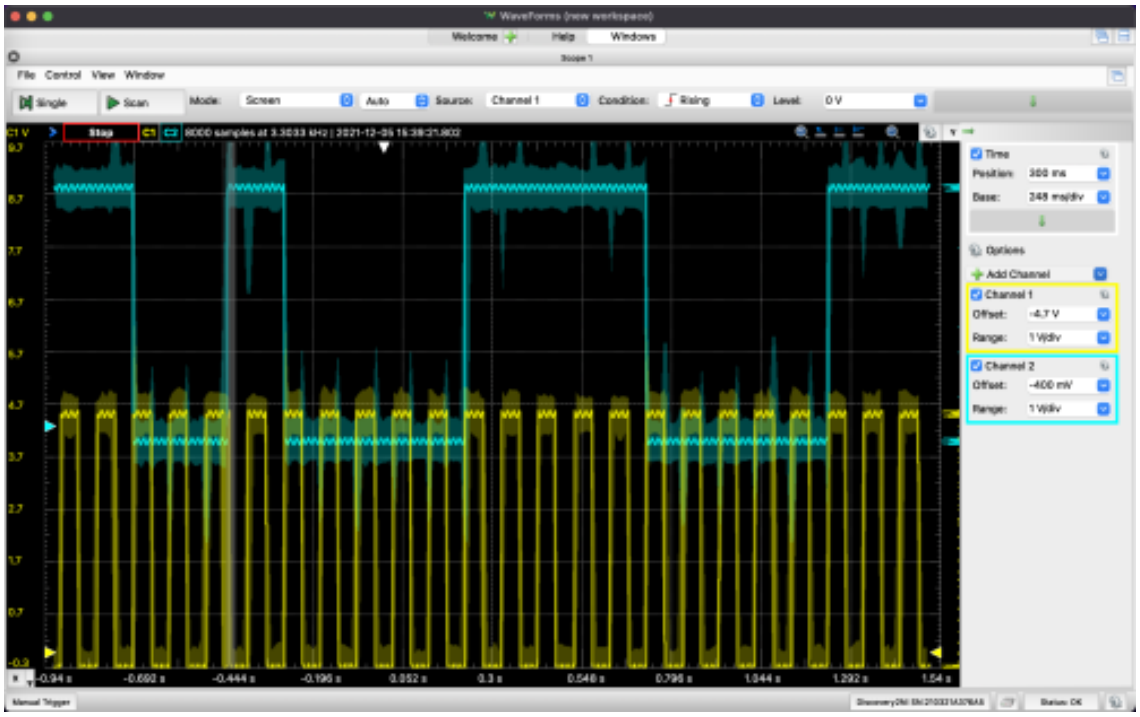
Time Range
Bag File
Bag Topics
Find Obj Parameters

5 Experimental results

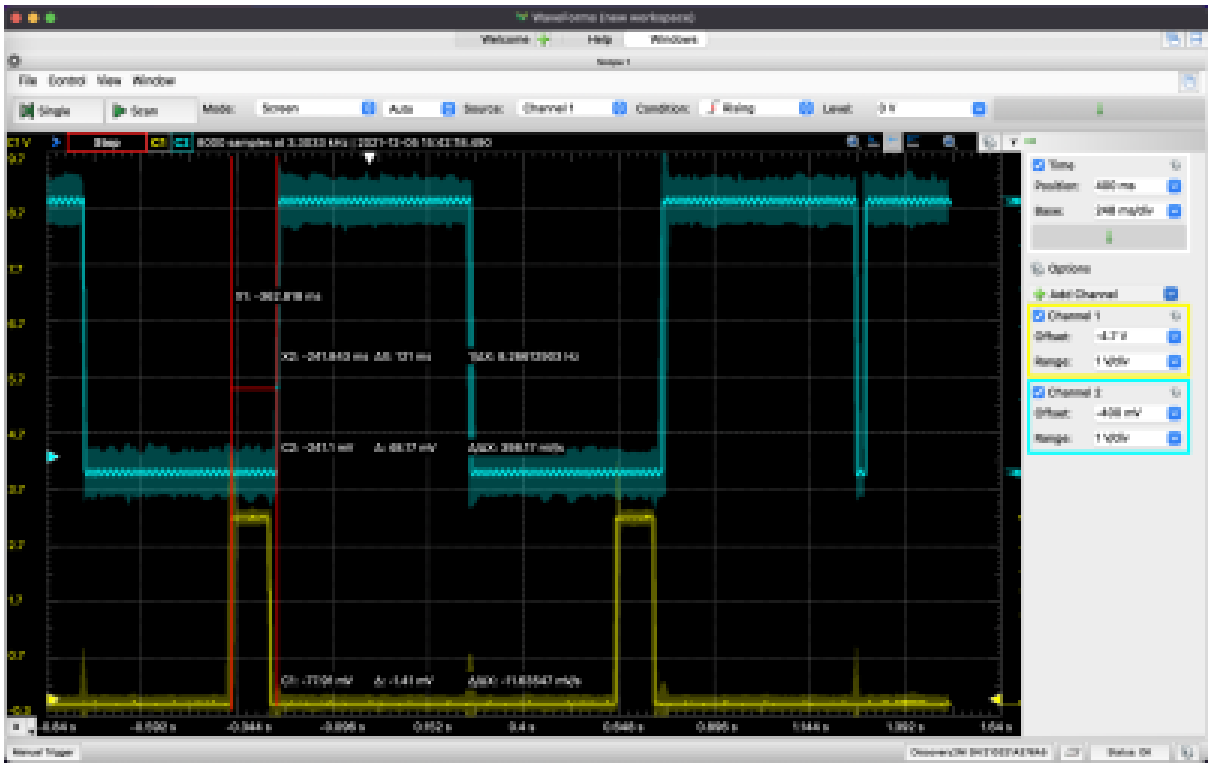
Analog Discovery - used the analog discovery to test the following:

- To view if the signals are outputting correctly using the scope.
- To measure if the signals are outputting the right frequency.
- To measure the delay of measuring the delay between the 1hz and 10hz rising edges.
- To measure the period of signals.

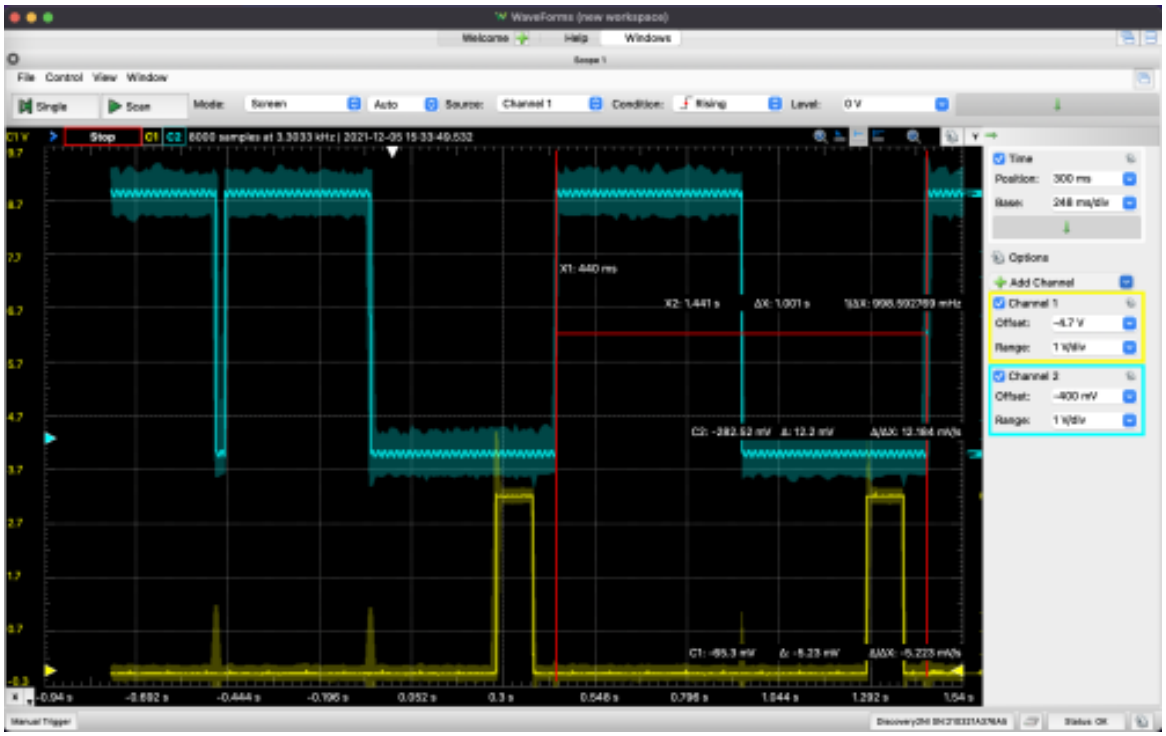
Test results-



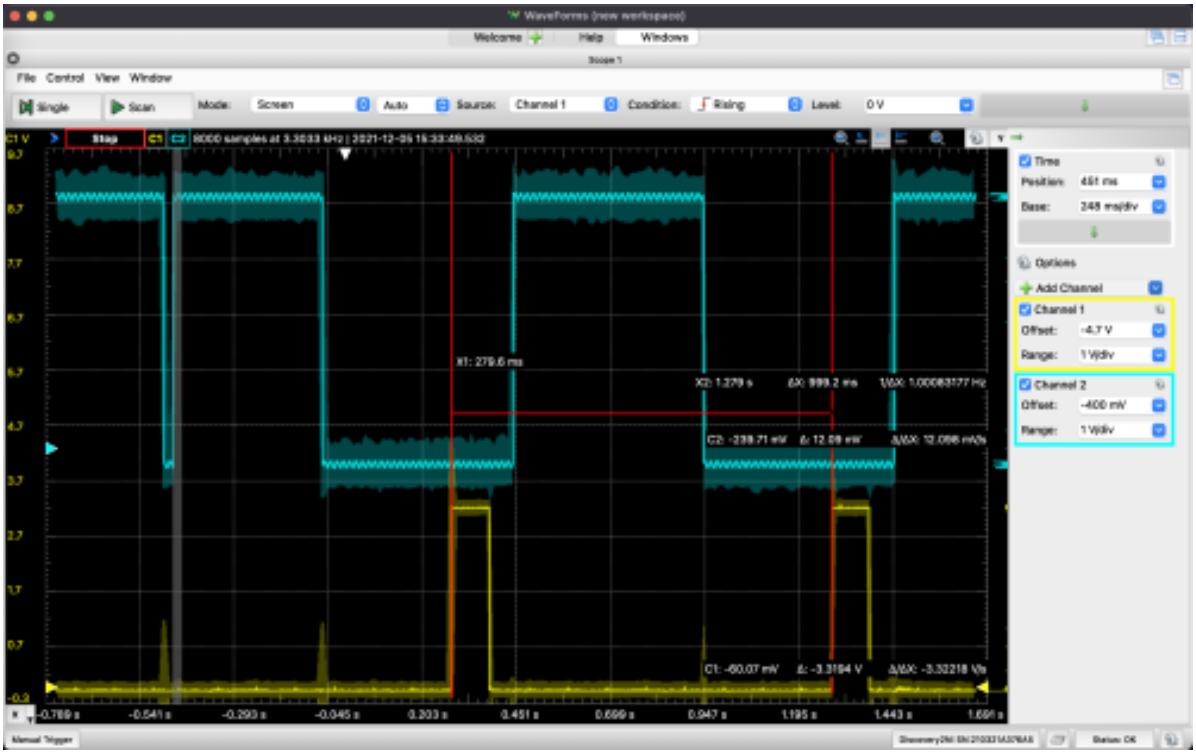
Output of the 1hz PPS signal and the 1hz signals. (yellow signal is PPS) (blue signal is 1hz with 50% duty cycle)



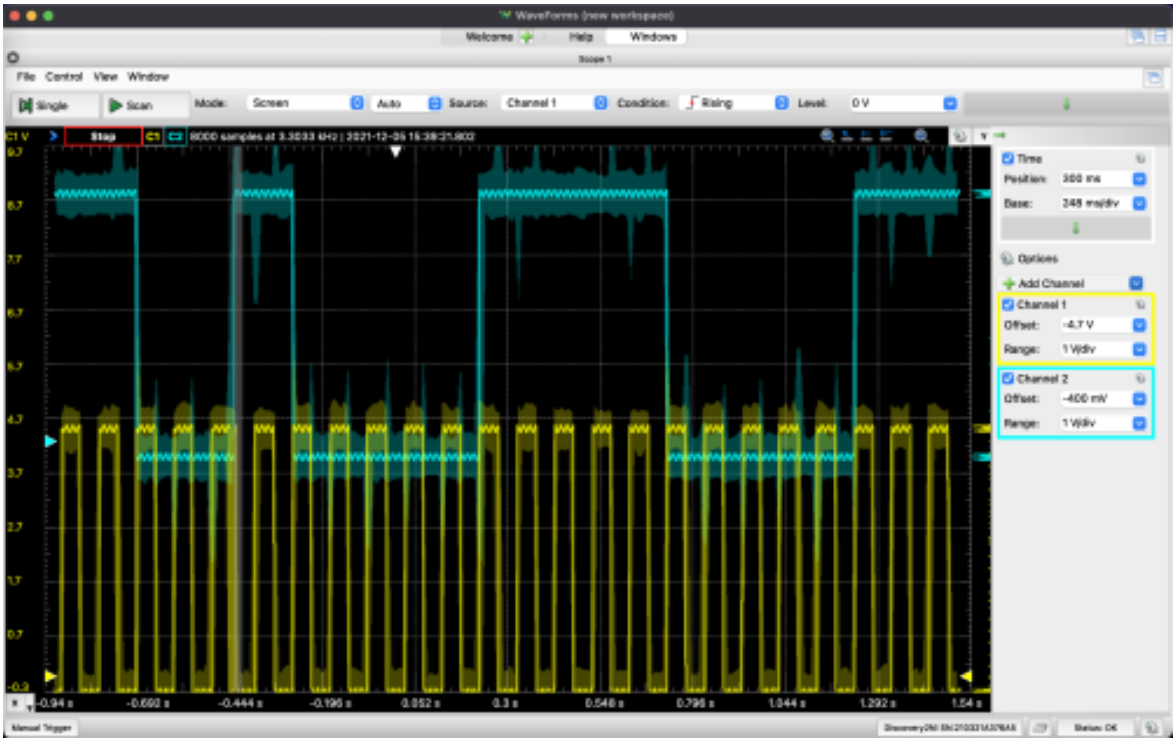
Measuring the delay between the 1hz PPS and 1hz rising edges: (delay is 163.5ms)



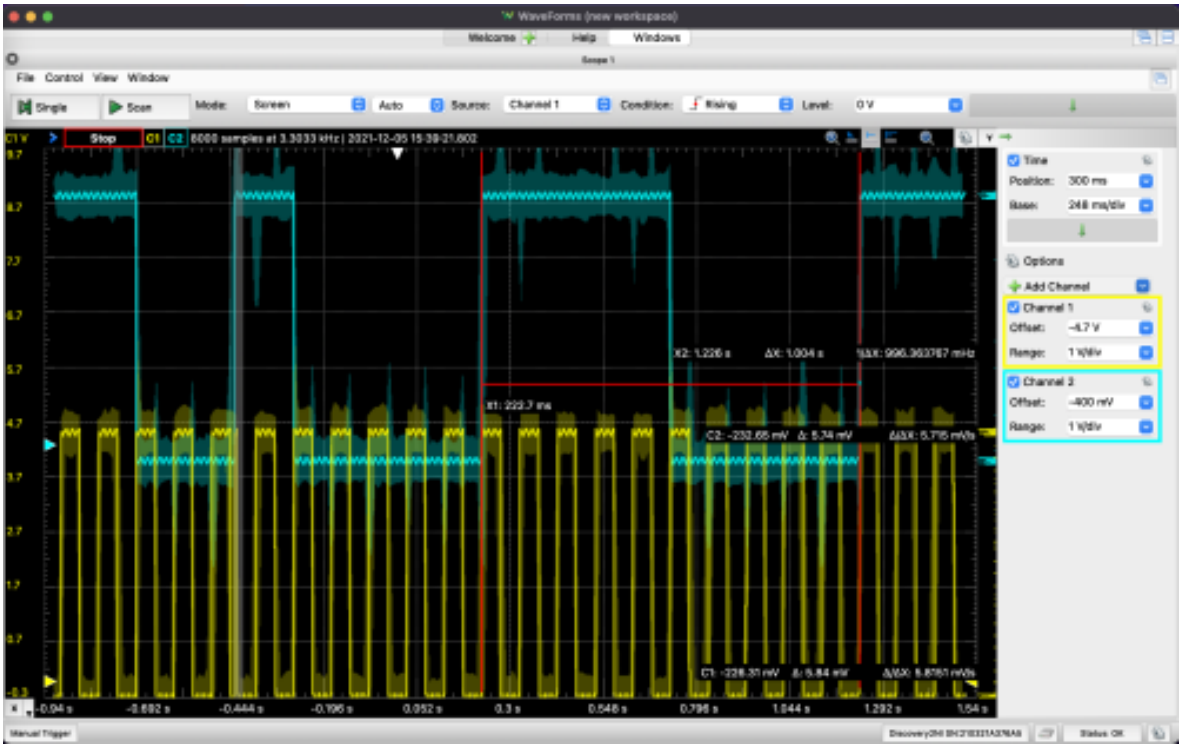
Measuring period of 1hz PPS signal: (period of 1hz PPS is 1.001s)



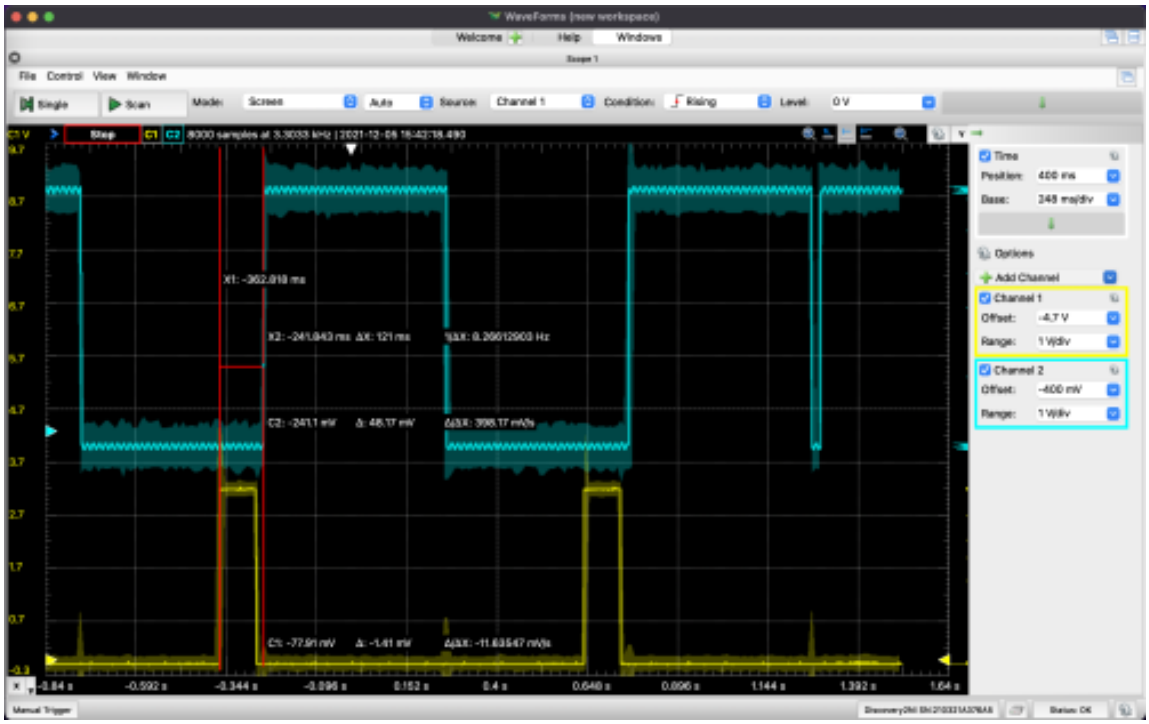
Measuring period of 1hz signal 50% duty cycle: (period of 1hz is 999.2ms)



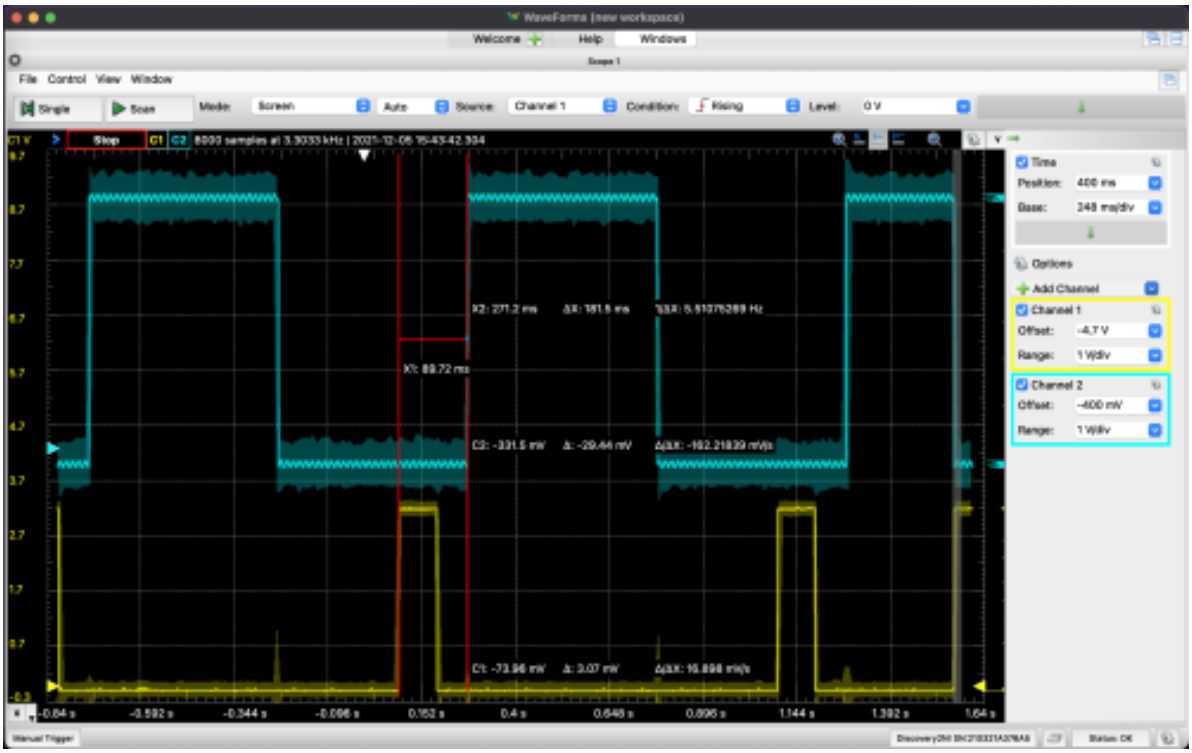
Output of the 1hz signal and 10hz signal. (yellow signal is 10hz) (blue signal is 1hz)



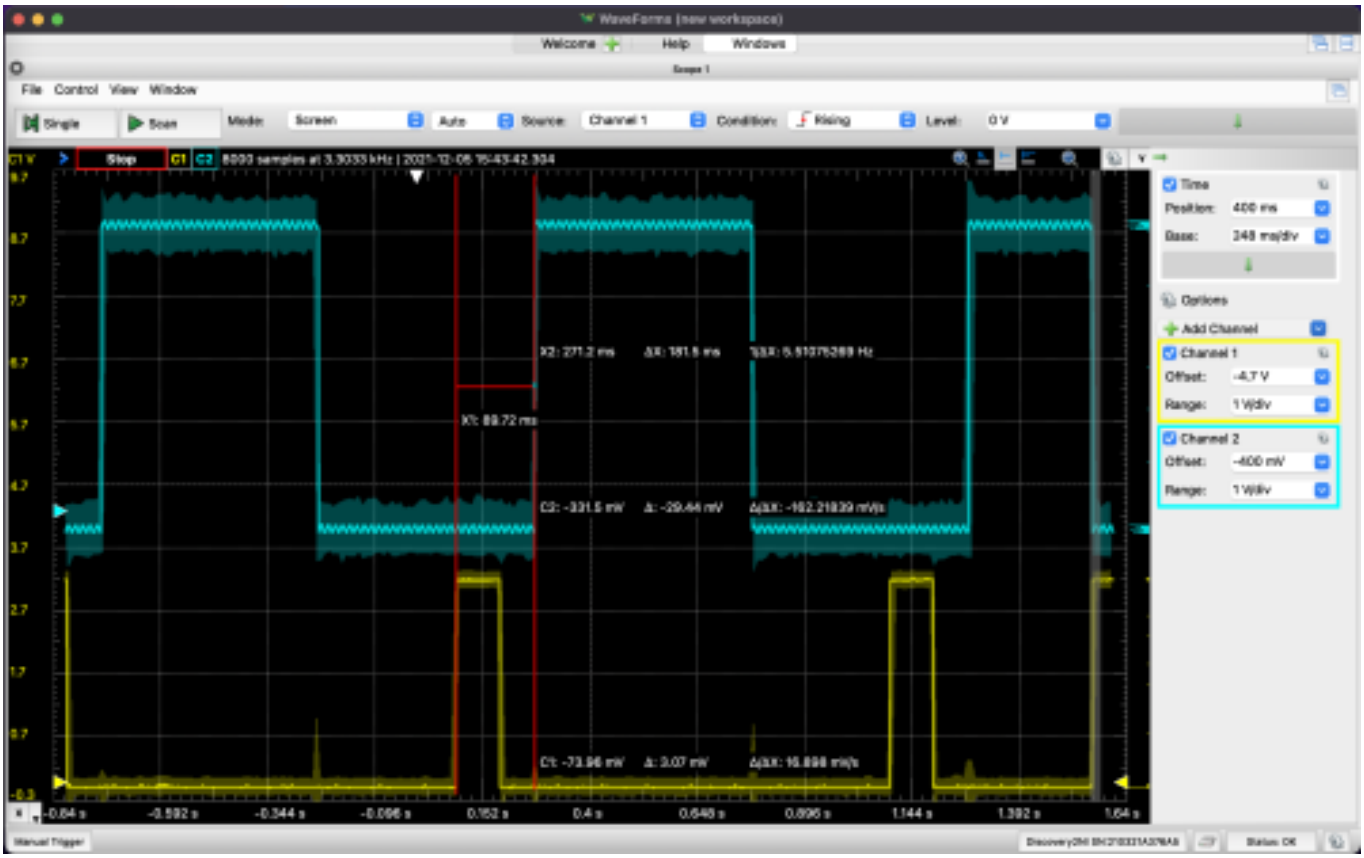
Measuring period of 1hz: (period of 1hz is 1.004s)



Measuring period of 10hz: (period of 10hz is 121ms)

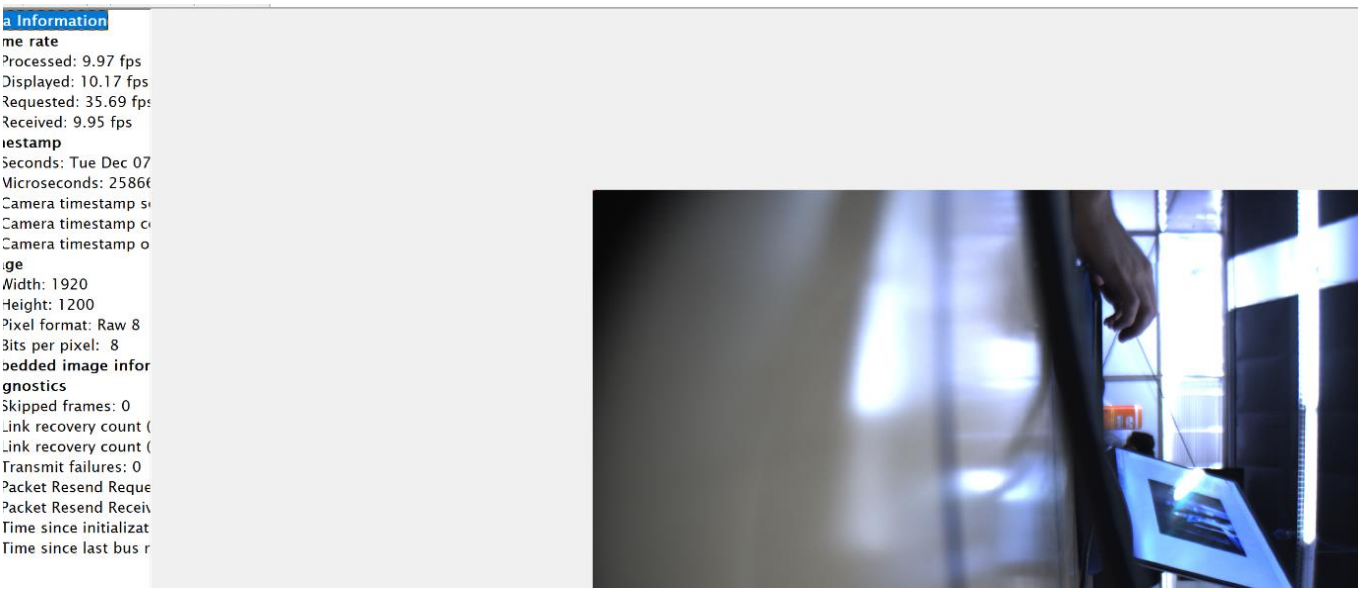


Measuring the delay between the 1hz and 10hz rising edges: (delay is 163.5ms)



Measuring the delay between the 1hz and 10hz rising edges: (delay is 181.5ms)

PointGrey FlyCapture 2 Output Validation:



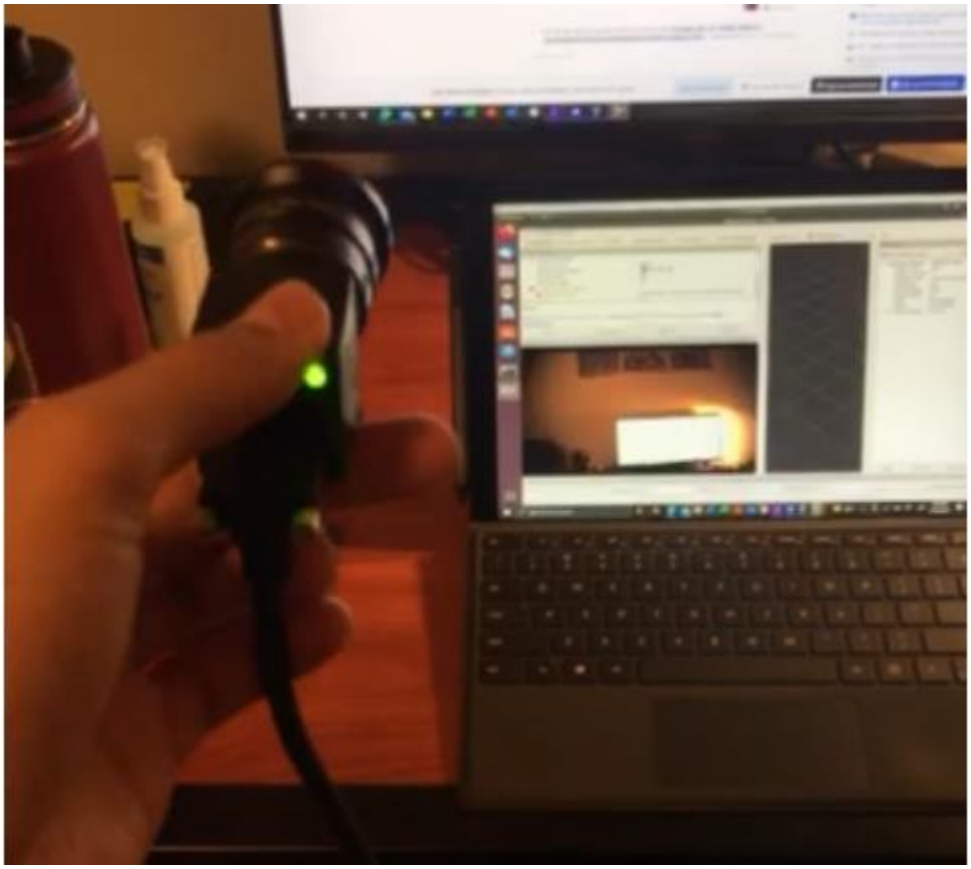
The PointGrey Software is seen outputting 10 fps in response to receiving the 10 Hz input

ROS Camera driver:

```
ritwik@ritwik-VirtualBox:~/catkin_ws$ roslaunch hardware_sync hardware_sync.launch
... logging to /home/ritwik/.ros/log/331df52e-5c78-11ec-a7b1-080027c0e811/roslaunch-ritwik-V
irtualBox-3903.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ritwik-VirtualBox:40113/

SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.12

NODES
  /
    camera_node (hardware_sync/camera_node.sh)
    rviz_node (hardware_sync/open_rviz.sh)

auto-starting new master
process[master]: started with pid [3914]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 331df52e-5c78-11ec-a7b1-080027c0e811
process[rosout-1]: started with pid [3925]
started core service [/rosout]
process[rviz_node-2]: started with pid [3928]
process[camera_node-3]: started with pid [3929]
... logging to /home/ritwik/.ros/log/331df52e-5c78-11ec-a7b1-080027c0e811/roslaunch-ritwik-V
irtualBox-3931.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ritwik-VirtualBox:35963/
```

```
SUMMARY
========

PARAMETERS
 * /camera/camera_nodelet/frame_id: camera
 * /camera/camera_nodelet/serial: 0
 * /rosdistro: melodic
 * /rosversion: 1.14.12

NODES
  /camera/
    camera_nodelet (nodelet/nodelet)
    camera_nodelet_manager (nodelet/nodelet)
    image_proc_debayer (nodelet/nodelet)

ROS_MASTER_URI=http://localhost:11311

process[camera/camera_nodelet_manager-1]: started with pid [3970]
process[camera/camera_nodelet-2]: started with pid [3971]
process[camera/image_proc_debayer-3]: started with pid [3972]
```

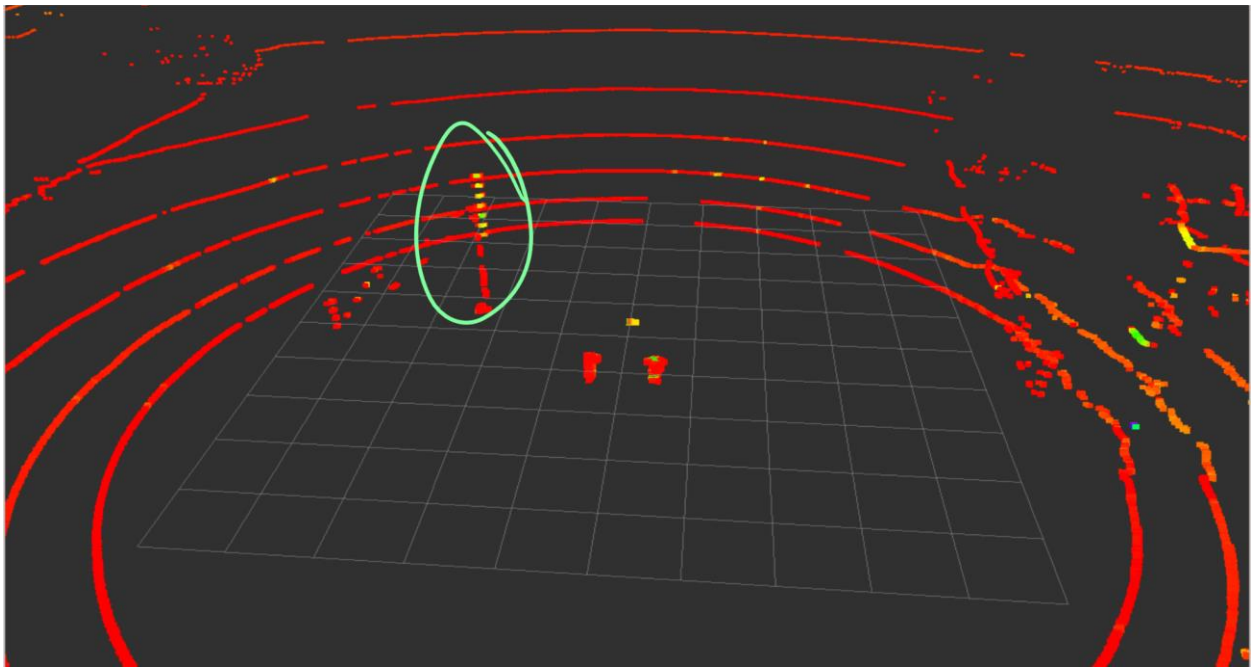
```
 * /camera/camera_nodelet/parameter_descriptions [dynamic_reconfigure/ConfigDescription] 1 p
ublisher
 * /camera/image_proc_debayer/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 1 publisher
 * /camera/camera_info [sensor_msgs/CameraInfo] 1 publisher
 * /camera/image_mono/compressed [sensor_msgs/CompressedImage] 1 publisher
 * /rosout [rosgraph_msgs/Log] 4 publishers
 * /initialpose [geometry_msgs/PoseWithCovarianceStamped] 1 publisher
 * /camera/image_color [sensor_msgs/Image] 1 publisher
 * /camera/image_raw/compressedDepth/parameter_descriptions [dynamic_reconfigure/ConfigDescr
iption] 1 publisher
 * /camera/image_raw/compressedDepth/parameter_updates [dynamic_reconfigure/Config] 1 publis
her
 * /camera/image_color/compressedDepth/parameter_updates [dynamic_reconfigure/Config] 1 publ
isher
 * /camera/image_color/compressed/parameter_descriptions [dynamic_reconfigure/ConfigDescript
ion] 1 publisher
 * /camera/image_raw/theora/parameter_updates [dynamic_reconfigure/Config] 1 publisher
 * /camera/image_raw/compressed/parameter_updates [dynamic_reconfigure/Config] 1 publisher
 * /camera/image_mono/theora/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
1 publisher
 * /camera/image_color/compressed/parameter_updates [dynamic_reconfigure/Config] 1 publisher
 * /camera/image_color/compressedDepth [sensor_msgs/CompressedImage] 1 publisher
 * /camera/image_mono/compressed/parameter_updates [dynamic_reconfigure/Config] 1 publisher
 * /camera/image_raw [sensor_msgs/Image] 1 publisher
 * /camera/image_color/theora/parameter_updates [dynamic_reconfigure/Config] 1 publisher
 * /camera/image_mono [sensor_msgs/Image] 1 publisher
 * /camera/image_color/compressedDepth/parameter_descriptions [dynamic_reconfigure/ConfigDes
cription] 1 publisher

Subscribed topics:
 * /tf_static [tf2_msgs/TFMessage] 1 subscriber
 * /rosout [rosgraph_msgs/Log] 1 subscriber
 * /camera/camera_nodelet_manager/bond [bond/Status] 3 subscribers
 * /tf [tf2_msgs/TFMessage] 1 subscriber
```



Snap of image captures being shown on Rviz.

Synchronization validation of camera and lidar captures
Raw ROS bag file: filtered-mcity_full_route_40.bag

Camera data: The pole in the circle.

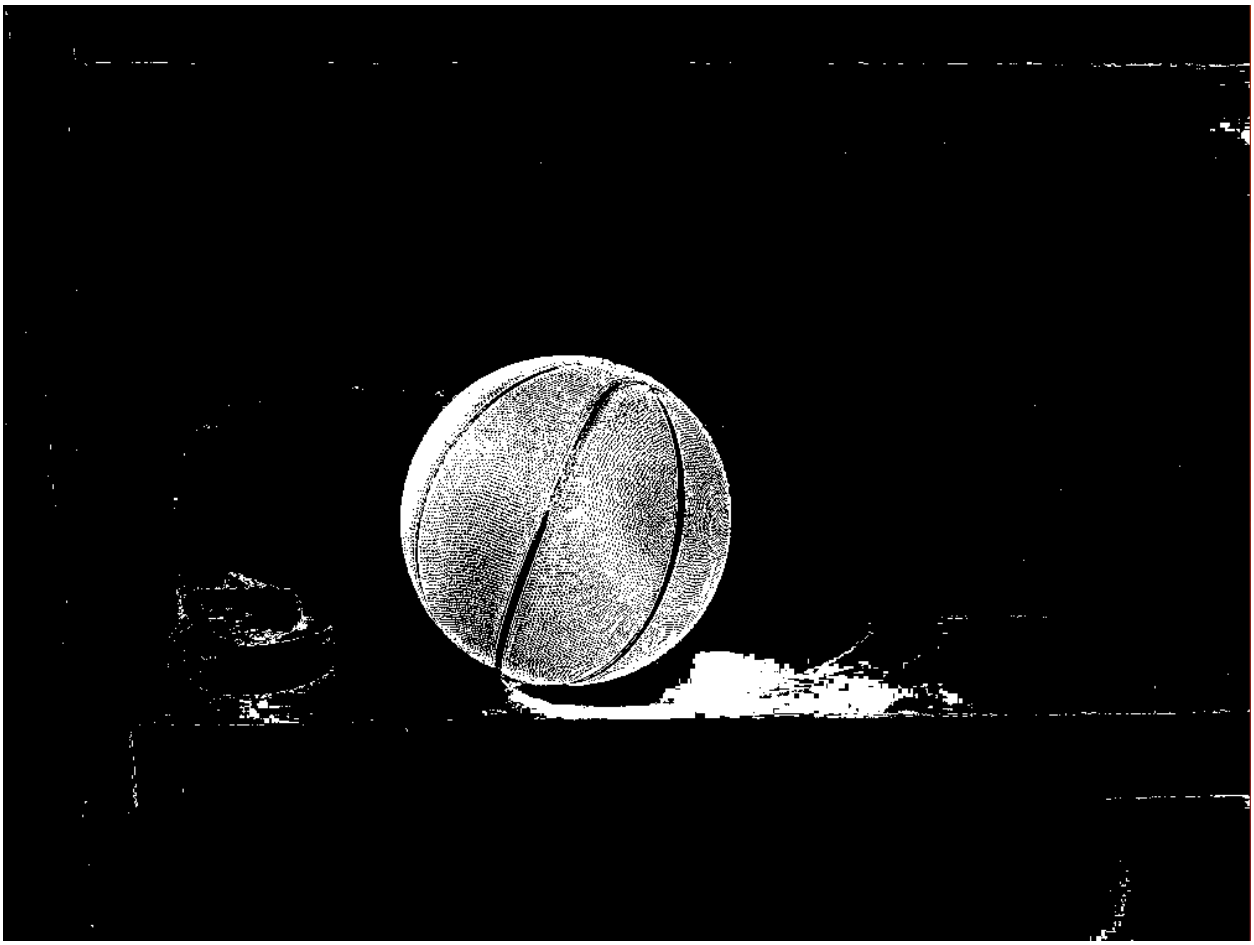


Lidar data: The pole, colored yellow by Rviz, in the circle.

Arduino code:

```
57  void setup()
58  {
59    Serial.begin(115200);
60    SerialGPS.begin(9600);
61
62
63    //  setSyncProvider(RTC.get);   // the function to get the time from the RTC
64    //  if (timeStatus() != timeSet)
65    //     Serial.println("Unable to sync with the RTC");
66    //  else
67    //     Serial.println("RTC has set the system time");
68
69    pinMode(ledApin, OUTPUT);
70    digitalWrite(ledApin, HIGH);
71
72    pinMode(2, INPUT);
73    attachInterrupt(0, pps_interrupt, RISING);
74
75    // set up the LCD's number of rows and columns:
76    lcd.begin(20, 4);
77    lcd.setBacklight(HIGH);
78    lcd.begin(20, 4);
79    lcd.print("GPS Interruption:");
80    lcd.setCursor(0, 1);
81    lcd.print("Time:     :  :  .");
82    //  lcd.setCursor(0, 2);
83    //  lcd.print("Fix:");
84    //  lcd.setCursor(0, 3);
85    //  lcd.print("Bat:     V        mA");
86
87
88
```

Validation



Camera Object Detection for a basketball on a shelf

```
coltmulk@ubuntu:~/validation/Sensory$ python decodebag.py filtered-mcity_full_route_40.bag 1625513773
.07 1625513774.64
('mean: ', 0.021130404117898877)
('standard deviation: ', 0.022078199541392257)
coltmulk@ubuntu:~/validation/Sensory$ python showCamDetect.py Left.jpg 0 10
438.819463412
```

Results when run for previous yellow pole in camera and lidar. Since both are at the same height the time error should also be low here.

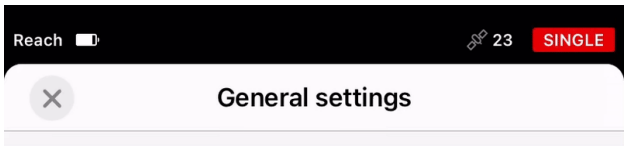
6 User's Manuals

6.1 Hardware Setup

Before powering on the system, all jumper wire connections must be verified to make sure that nothing has come undone. Also, this system must be outside for the GPS antenna to successfully connect to a satellite.

1. First, power on the system by plugging the lidar, camera, and arduino need to be to a power source (wall outlet or USB).

2. Wait for the Arduino to load the program. This is done when there is an orange LED that verifies the program is uploaded.
3. When everything is powered on, hold the GPS antenna up vertically to the sky or position it where it is pointing straight up.
    a. Load up the ReachView 3 app to verify that the GPS is sending out a PPS signal. In the top right-hand corner, a red box with the text “Single” should be displayed (shown below).



4. After all of this is verified, the system should work as intended and you can move to the software setup.

6.2 Arduino code for Waveform generation

Before running the Arduino code make sure that all of the connections have been established and that the Arduino has been powered on.

1. Download the Arduino IDE using the following link
    a. https://www.arduino.cc/en/software
2. Download the file called “Arduino10HZ1HZ” from the project github and open it in the Arduino IDE
3. Verify that the Input pins and Output pins are connected and correspond to the code.

```
5   #define OUTPUT_PIN 2  // 10Hz Output
6   #define OUTPUT_PIN2 4 // 1Hz Output
7
8   #define INPUT_PINGPS 8 // GPS PPS input
9
```

    a.
4. To change the variables of the 1 Hertz output change the corresponding variables to the desired output.

```
37    // 1hz Variables
38    onehzCurrent = 0;
39    onehzPeriod = 10; // Set this to 10 because of the 10hz
40    onehzDutyCycle = 5; // 5 is 50 percent
41    onehzPhaseShift = 2; // Phase shift compared to start of the 10hz
42    onehzBacksideCycle = onehzPhaseShift + onehzDutyCycle;
43  }
```

    a.
5. Connect the Arduino to the Computer using the USB to USB micro cable
6. Make sure that the port is connected to the Arduino is selected
    a. Go to Tools - Port - then select the port that has Arduino in the name
7. Upload the code to the arduino using the “right arrow” on the top left
8. Now your code will be outputting the desired 10 Hertz and 1 Hertz when it receives the GPS PPS.

6.3 Installing FlyCapture2 SDK

FlyCapture SDK is necessary to boot up the camera before it can capture and transmit image data.

1. Download FlyCapture2: On your Internet browser, go to https://www.flir.com/products/flycapture-sdk/.

Click "Download Now". A window will pop up. Click "Linux" then "flycapture2-2.13.3.31-amd64-pkg_Ubuntu18.04" The tar file will download and appear in your computer's "Downloads" folder. Go to the "Downloads" folder, right click the file called "Linux.zip", and then click "Extract here" in the options menu that pops up. A new file called "Linux" will appear. Double click it to enter that folder. Then enter into the "flycapture2-2.13.3.31-amd64-pkg_Ubuntu18.04" folder, and then "flycapture2-2.13.3.31-amd64" folder. Right click in the white space and click "Open in Terminal" to launch the terminal at the folder.

2. Install the dependencies. In the terminal, type *sudo apt-get install libraw1394-11 libgtkmm-2.4-1v5 libglademm-2.4-1v5 libgtkglextmm-x11-1.2-dev libgtkglextmm-x11-1.2 libusb-1.0-0*

   and hit Enter.

3. Install the SDK. In the terminal, type *sudo sh install_flycapture.sh*

   and press Enter. Whenever you're asked "Do you want to continue?", type *y*

   and hit Enter. When you get asked about signing up for udev, type *n*

   and press Enter.

   You have now installed FlyCapture2.

6.4 ROS camera driver launch file

The camera captures can be visualized in real time from the PPS triggering.

When the camera is connected to the computer via USB 3.0, it is initially on standby, as indicated by the orange light.

1. First, the PC's USB memory core needs to be expanded to at least 1024 bytes or else there will not be enough memory available for the camera to send the image messages. The memory core's current status is checked as shown below:
   a. In the terminal, type */sys/module/usbcore/parameters/usbfs_memory_mb*

      and hit Enter. If the value returned is less than 1024, then type the following command in the terminal:
      sudo sh -c 'echo 1024 > /sys/module/usbcore/parameters/usbfs_memory_mb'

      and press enter

2. Second, the camera needs to be booted by the FlyCapture2 SDK.
   a. In the terminal, simply type *flycap* and hit Enter. The FlyCapture2 GUI launches. If the camera is properly connected (including that USB 3.0 is enabled), then you will see the Blackfly camera on the Camera List. At the bottom of the screen, click "OK". This changes the status indicator on the camera from orange to green. Now, the camera is ready to send image messages. Do not worry about the image captures displaying in flycap. You can also now quit flycap.

      Now the camera driver and Rviz can be launched.

3. Navigate to the “Sensory” folder and open the terminal there. The files “camera_node.sh” and “open_rviz.sh” need to be changed to executable. In the terminal, enter the following commands:
   sudo chmod +x camera_node.sh
   sudo chmod +x open_rviz.sh
4. Next, under the “hardware_sync” folder where the catkin environment was created, create a new folder called “launch”, and extract “launch.zip” (from git) to that folder.
5. Then go back to “catkin_ws”, the parent folder of the ROS environment (hardware_sync), and source it:
   source devel/setup.bash
6. Now, type
   roslaunch hardware_sync hardware_sync.launch.
   a. The PointGrey camera driver will start up and Rviz will be booted up. To see the images in real time, click “add” on the bottom of the left panel, then on top of the window that pops up, click “By topic”. The topic “/front_camera/image_color” topic should be visible. Under that, click “Image”, and now an Image panel will appear on the bottom left of Rviz, and the live images should be visible. The PPS triggering does not affect the configuration of the camera. The camera sends image messages whenever triggered, and that can be seen in Rviz.

6.5 Data Validation Code

Environment setup-

First install ROS and necessary libraries

Install pypcd using pip

pip install pypcd

Install rospy

```
sudo apt update
sudo apt install python-rospy
```

Decode bag usage: python decodebag.py bagFile timeLow timeHigh

Replace bagFile with the name of the bag file you're using. Replace timeLow and timeHigh with the time range you want the program to run. Helps to look through them first in rviz or in the command prompt to see what range you want.

Values to change

The first values to change are the topics, change them to whatever they are called in the bag file you are using.

Next change the camera values. You need 5 values. First the center y pixels that the basketball (or other object) drops from and touches the ground. Next you need the height of that distance. Finally you need to change the hue0 and hue1 values to the right range for whatever object you are detecting.

Next change the lidar values. You need the x, y and z range to look for the object in. The best way to get these is to load the lidar message into rviz and check the values around where the object is. Next you need the z value that the object starts dropping from (also can be found in rviz).

```
def validate(bagFile, timeA, timeB):
        #takes in bag file and the time range at which the error will be calculated
        t0 = rospy.Time(timeA).to_sec()
        t1 = rospy.Time(timeB).to_sec()
        bag = rosbag.Bag(bagFile)

        img_msgs =  bag.read_messages(topics=['/front_camera/image_raw', '/lidar_left/velodyne_points'])

        bridge = CvBridge()

        lidarH = []
        cameraH = []
        for topic, msg, t in img_msgs:
                if t.to_sec() >= t0 and t.to_sec() <= t1 and topic == '/front_camera/image_raw':
                        cv_image = bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
                        cv_image = cv2.resize(cv_image, (800, 600))
                        #gets camera y pos (using ground pixel position and pixel drop height and height droped in meters) then hue values
                        var = (cd.cameraFindObj(cv_image, 250, 450, 2, 150, 170), t.to_sec())
                        cameraH.append(var)
                if topic == '/lidar_left/velodyne_points' and t.to_sec() >= t0 and t.to_sec() <= t1:
                        pc = pypcd.PointCloud.from_msg(msg)
                        #gets lidar with bouding cords x, y, z (z should be height) and then the height droped from (in z cords)
                        var = (cd.lidarFindObj(pc, 6.85, 6.95,3.70, 3.95,-2.3,0),t.to_sec())
                        lidarH.append(var)

        bag.close()
```

Topics

Camera Height

Camera Ground and Start Pixels

Hue Values

X range

Y range

Z range

Z Drop Value (lidar)

Once the necessary values are set simply run the program with the bagFile name and the time range desired.

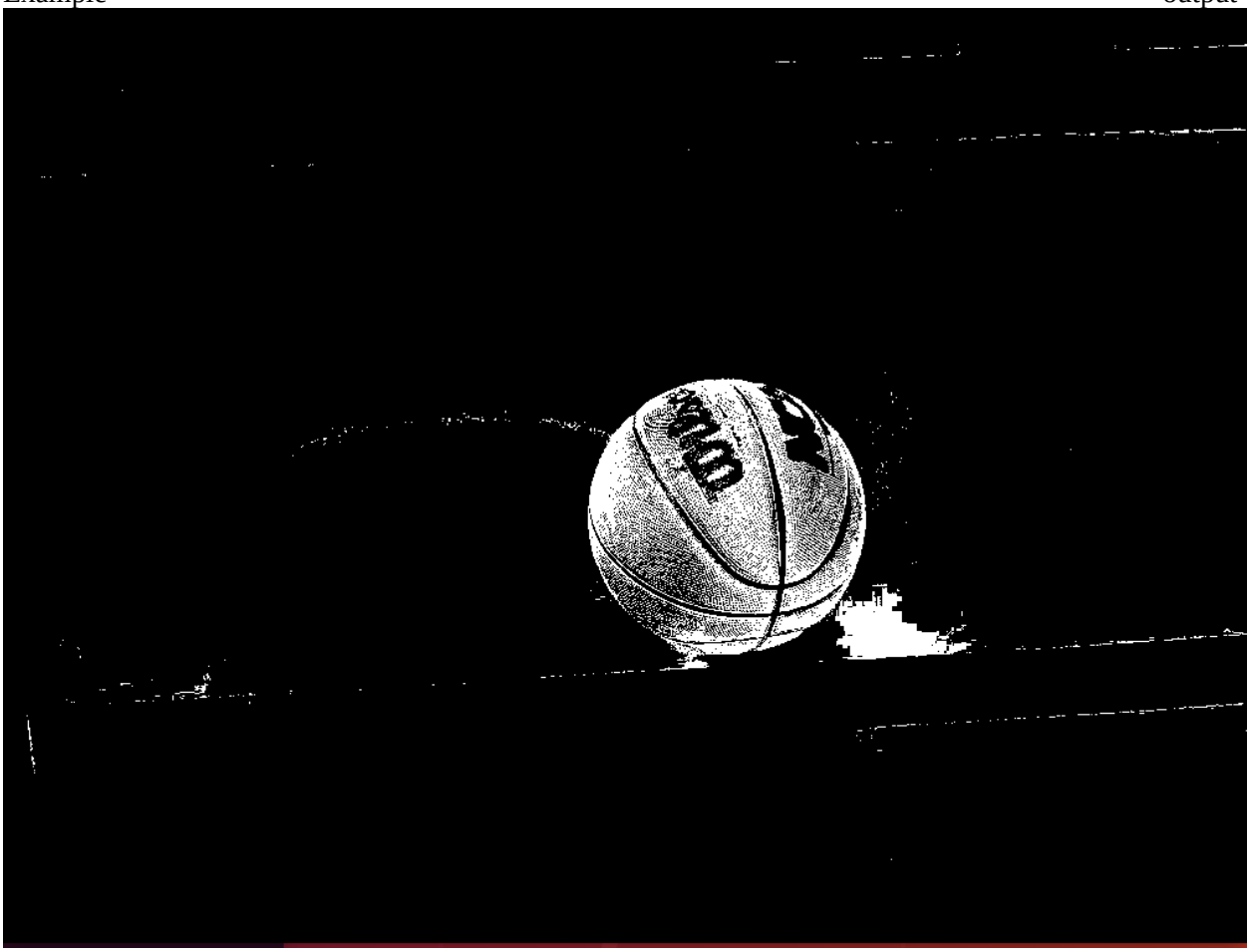
Camera Object Detection-

-Used to run the camera detection on an image file and show the results to the user.

Ex Usage: python showCamDetect.py ImgName hue0 hue1

where hue0 is the lower bound for hue and hue1 is the upper bound. ImgName is the name of the image file you want to run this on. This program will also output the average y of the object found in the specified image based on the hue value chosen.

Example output-



7 Course debriefing

7.1 Provide a thorough discussion of your *team management* style. If you were to do the project again, what would you do the same, what would you do differently?

- Ritwik Sathe- With the circumstances given to us, the team management style went very well. Dividing the team into two groups, one for hardware and one for software, simplified the division of labor and made the assignment of tasks more organized. This was reflected by the creation of three channels in our Discord server, where two were for each group plus a "general" channel for messages relevant to all of us. Each member could still see the communications of the opposite team so they stay "in the loop" and figure out how the other team's developments affect his own team's developments. The in-person team meetings allowed all of us to update each other on our specific accomplishments and objectives still to reach. The instructors also helped us determine step-by-step what tasks should be completed and in which order. There was very little that could have been done differently simply because all of us were new to the ROS and Arduino framework. We all had to learn together, and it took a while for us to figure some things out so we could at least start somewhere and build on from there.
- Wyatt Hansen
- Matt Boyett- The team management style worked perfect for me. By splitting off into two teams and having individual Discord channels to communicate through and separate objectives to accomplish through each week, micromanagement was non-existent. With our full team

meetings, we were able to come together and collaborate more thoroughly and debrief each other which was very helpful when the hardware team would hit bumps that we needed further input on. I also felt that the frequency of full team meetings gave me time to not only collaborate and learn, but to also bond with all the team members. It was a productive and efficient style and I wouldn't have changed anything.

- Colton Mulkey- The team management style worked quite well. We had regular meetings to check up on our work and update what thoughts we had on the project. If one of us was unclear about something or needed clarification, meeting up several times weekly helped significantly. If we were to do the project again the only thing I might suggest doing differently is to have short meetings on discord a bit more often that way we can continually stay on the same page even more often.
- Michael Mengistu- I enjoyed our team management style. If I were to do this project again the things I would do the same would be to keep having the same meetings. What I would do differently would be to have the hardware equipment on time.

7.2 Are there any particular *safety* and/or *ethical* concerns with your product(s)? What steps did your group take to ensure these concerns were addressed? Are there any additional steps you would have taken if you were to do the project again?

- Ritwik Sathe- Absolutely. If the sensors were not synchronized properly and the captured data were not validated smoothly, then the car would definitely fail to keep its passengers safe. My contributions addressed these concerns by ensuring that enough memory was available so that the sensor data could be processed efficiently.
- Wyatt Hansen
- Matt Boyett- From the start, we realized the importance of this project and what the implications are surrounding the necessity of a completed output. Without fully synchronized sensors, an autonomous vehicle could collide into either an object or, much worse, property or another person. By documenting everything fully and making sure to use airtight coding practices, we strove to have a product that was not only complete but also fully understood so someone could diagnose any potential problems when future iterations are made. Because of how we went about developing this system and the care that we took, I actually do not have any safety concerns.
- Colton Mulkey - Our product has the potential to cause a failure in the system it is used which can spark all kinds of safety concerns. This is why we made sure to do our utmost to create a system that worked well so a system using this would not fail. In order to test how good our product was we also made sure to focus on our validation so that we could verify that our product would be effective at what it is designed to do. If we were to do the project again the only thing I would have done was to take additional steps to obtain hardware faster so that we could have more time testing the system.
- Michael Mengistu- the only safety concerns with our product is that we don't have an alert system. The steps we took to ensure this was addressed was by testing our product thoroughly. The additional steps I would have taken would be to make an alert system.

7.3 Did you test your product(s)? Do they work as proposed? Can you think of any relevant situations in which you haven't tested your product(s)? If you were to do this project again, what additional *verification* and *testing* procedures might you add?

- Ritwik Sathe- The only part that I could test was the display of the captured images by the camera in real time. Fortunately, it worked as expected after hours of configuring the PC, installing dependencies, and figuring out how to boot up the camera from the PC itself. If we were to have more time, we could try to get the lidar to work as well. One thing I wish the lidar provided was

the ability to connect via USB rather than Ethernet because the network connection setup was nearly impossible with the current configuration of my Ubuntu VM. Then it would be easier to test the data capture discrepancies between the lidar and the camera, and we would have tested basketball motions like we were supposed to.

- Wyatt Hansen
- Matt Boyett- Yes, we extensively tested the product at each stage of development and we have a fully working system. However, our design has in fact changed from our initial proposal. Due to sensor availability and extraneous circumstances, we had to cut the radar functionality entirely and were not able to test our lidar functionality due to the loss of our power cable. However, everything for the lidar is in place from the hardware and software side. If we were to do this project again, we would of course want to test the lidar and verify that our 1 Hz output would trigger it without any hiccups.
- Colton Mulkey - We have tested our project and it seems to work well for the time we tested it. There are some situations where we have not tested which is how the system performs over a long period of time. If we were to do this project again a focus would definitely be on allowing us to test our product over extended periods of time.
- Michael Mengistu- yes we tested our product. The product does work as it was proposed to. If we were to do this again, the additional verification and testing procedures would still be the same.

8 Budgets

While the initial design has remained unchanged, our budget has been revised to more accurately represent our current and future expenditures.

Hardware and Cost		
Hardware	**Amount**	**Cost($)**
Arduino Uno SMD	1	30
Arduino PSU	1	7
Breadboard and Wires	1	10
Basketball	1	16
Total Cost		63

9 References

1. “Configuring synchronized capture with multiple cameras,” *Teledyne FLIR*, 22-Dec-2017. [Online]. Available: https://www.flir.com/support-center/iis/machine-vision/application-note/configuring-synchronized-capture-with-multiple-cameras/. [Accessed: 14-Dec-2021].
2. “Flycapture2::ErrorType 33 error starting isochronous stream. edit,” *FlyCapture2::ErrorType 33 Error starting isochronous stream. - ROS Answers: Open Source Q&A Forum*. [Online]. Available: https://answers.ros.org/question/361009/flycapture2errortype-33-error-starting-isochronous-stream/. [Accessed: 14-Dec-2021].
3. “Reading messages from a bag file,” *ros.org*. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/reading%20msgs%20from%20a%20bag%20file. [Accessed: 14-Dec-2021].
4. “rosbag/Code API,” *ros.org*. [Online]. Available: http://wiki.ros.org/rosbag/Code%20API#CA-51a905f36bc234bd1e1cd538cbb0da1aea6300c4_8. [Accessed: 14-Dec-2021].
5. PyPI. 2021. *pypcd*. [online] Available at: <https://pypi.org/project/pypcd/> [Accessed 14 December 2021].