Sensor Team One Final Report

Authors:

Elijah Morgan
Morgan Bounds
Avriel Harvey
Soumil Verma

Table of Contents

1. Abbreviations and Acronyms

Abbreviation	**Definition**
CSI	Camera Serial Interface
FPS	Frames per Second
GPIO	General Purpose Input/Output
GND	Ground
HDMI	High Definition Multimedia Interface
IR	Infrared
IMU	Inertial Measurement Unit
MIPI	Mobile Industry Processor Interface
PCB	Printed Circuit Board
SD	Secure Digital
SDIO	Secure Digital Input/Output
TRIG	Trigger
USB	Universal Serial Bus
VCC	Positive Supply Voltage

2. Sensor Package Requirements

2.1 Overview

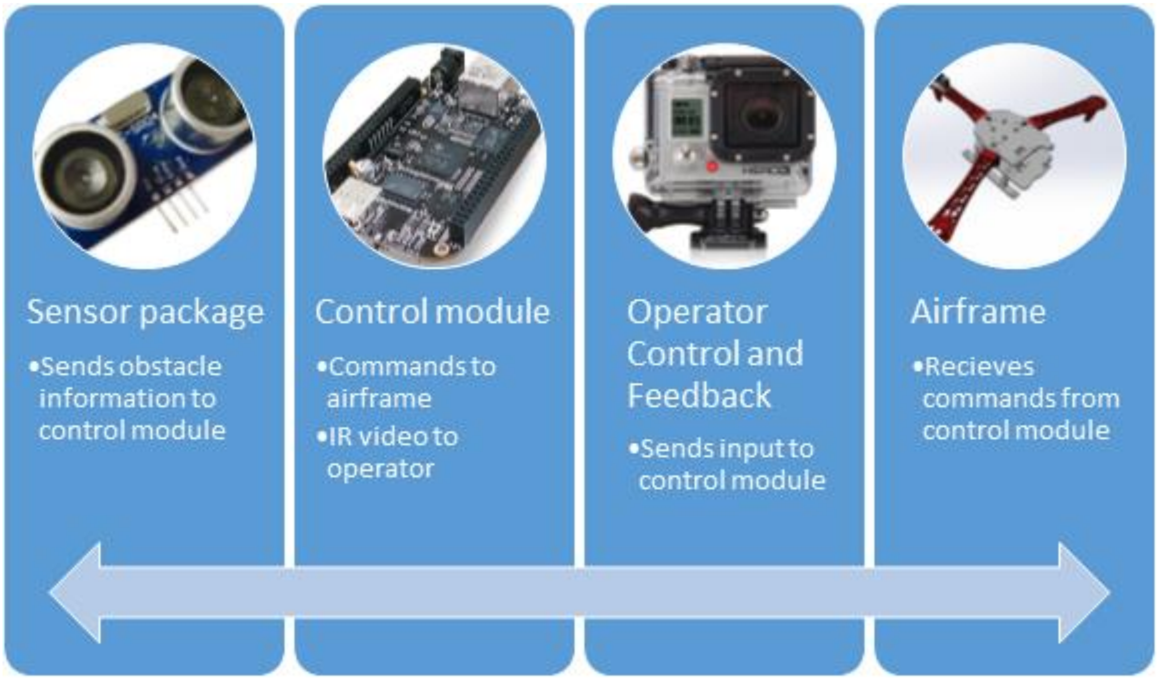

Sensor package
•Sends obstacle information to control module
Control module
•Commands to airframe
•IR video to operator
Operator Control and Feedback
•Sends input to control module
Airframe
•Recieves commands from control module

Figure 1.1.1

The basic requirements for the sensor package are simple: detect obstacles within three meters of the drone and report those obstacles to the control module. The way in which the sensor module reports to the control module is by calculating the largest possible rectangular prism of clear space around the drone. The six points that define this rectangular prism of clear space are passed to the control module as distance measurements. This is a simple yet robust system for communicating clear space. However, there are many challenges in implementing this. First, the sensors must work in an environment with fire and smoke. The entire module must be light enough to be carried by the airframe. It must balance accuracy and precision with frames per second, or how often it updates the measurements to the control module. Finally, the entire package must be low cost. Adding up all of these restrictions and parameters means the sensor package must contain a careful balance of many different attributes.

2.2 Major Goals

The primary goals of the sensor module are to:

- Accurately and quickly detect obstacles within zero to three meters
- Be lightweight and inexpensive
- Operate in a smoky and hot environment

3. Initial Research & Overall Design

3.1 Choosing a Sensor

The first part of the design process was to evaluate what sort of sensors were on the market and readily available to us. Since our entire design will be based off of using these sensors it was important to solidify that choice as quickly as possible. We split up our efforts into looking at a variety of possible types of sensors.

- Infrared
- Laser
- Pressure sensors
- <u>Ultrasonic</u>
- Radar
- Temperature

Our first impression was that ultrasonic sensors and infrared sensors would not be effective in this environment due to ambient noise and heat. Any other technologies we looked into were either expensive, heavy, or undeveloped for our purposes. We had initially hoped that radar sensors would work, but our research turned up nothing useful for the close range scale we needed. After eliminating other options we returned to researching ultrasonic and IR sensors. The IR sensors could possibly be adapted to be used in an environment with fire and smoke if we implemented a lot of digital signal processing. The benefit of IR would be a very high FPS (rate at which it updates the distance to an object). The biggest drawback is that it would only detect obstacles in a very small cone. We then turned to ultrasonic sensors. Ultrasonic sensors would require much less processing and if the right one was used it would take little time to get it running. Like IR sensors, ultrasonic sensors also have a cone in which objects can be detected. The disadvantages are that it is somewhat less accurate and has a slower FPS, because sound travels slower than light.

We decided to go with ultrasonic sensors because of the ease of use, lower cost, and the wide cone of detection. Once we had decided on ultrasonic sensors we started looking for a specific sensor. Our criteria for choosing a sensor follows:

- Cost
- Weight
- Community of hobbyists and support
- Accuracy of object detection
- Probability of working in a smoky environment
- Ease of use

Most cheap ultrasonic sensors cost somewhere in the range of $15 to $30 each. They had an effectual angle of detection somewhere between 45-60 degrees with a range beginning under one meter and ending between three to seven meters. We soon discovered the HC-SR04 ultrasonic sensor, which is a low cost sensor that met our criteria. Below we will talk about some of the most important aspects of the HC-SR04 for our applications. We will also link to another document that has more detailed information about the sensor.

- Maximum sensing range up to 4 meters
- Angle of detection approximately 30º
- Weight: 10 grams
- Used by the hobbyists community for a variety of applications
- Sensor will perform in smoking and heat conditions

More detailed sensor specifications are below, these specifications are from Cytron Technologies (**source 1**).

- Power Supply: +5V DC
- Quiescent Current: <2mA
- Working current: 15mA
- Effectual Angle: <15º
- Ranging Distance: 2-400 cm
- Resolution: 0.3 cm
- Measuring Angle: 30º
- Trigger Input Pulse width: 10uS
- Dimension: 45mm x 20mm x 15mm

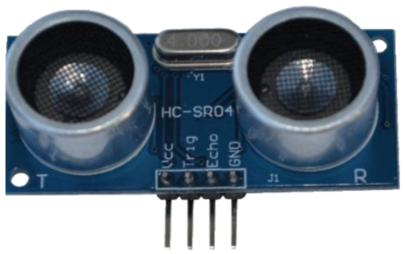


Figure 3.1: HC-SR04

The HC-SR04 caught our eye for several reasons. The first was the price, it is available from a variety of retailers for less than $4. The second was that it is used by hobbyists on many different projects. This means there are many helpful articles and blogs about the sensor and how to use it. There are also sample bits of code to run the HC-SR04 on the Raspberry Pi B+ and BeagleBone Black. Third it has an effective range of 2-400 cm which meets our requirements. Finally the sensors setup is ideal for using multiple sensors in conjunction with a Microcontroller. Since the HC-SR04 met our requirements, especially its price, we decided to go ahead and use it in our design.

3.2 Choosing a microcontroller

There are several microcontrollers that we considered for operating the sensor package: the Arduino Uno, Arduino Mega, Raspberry Pi Model B+, TI MSP430, and the BeagleBone Black. Based on our research and experience, the Raspberry Pi has proven to be the best microcontroller for our requirements this semester. Although the BeagleBone Black contains more processing power than the Raspberry Pi, we determined that it was not our best option, primarily due to the fact that the community around the BeagleBone Black is much smaller than that of the Raspberry Pi. A high level of community and support was important to us due to our lack of experience with microcontrollers.

Some of the benefits of the Raspberry Pi include:

- Low power consumption
- One location for quick compilation and execution of code (vs. Arduino or other microcontrollers that require an upload of code)
- Support for C and Python
- Runs Linux OS: kernel can do multitasking
- 40-pin GPIO header
- Large community of hobbyists for software and hardware support
- Has the processing power to support all components of the drone (control, sensor, operator camera)
- Real-time kernel to reduce latency in the processes
- Compatible with Arduino

3.3 Choosing a power supply

Based on the microcontroller and sensor power requirements, each utilized up to 5 volts and 1200 mA (Raspberry Pi), 15mA (sensor). The recommendation was a small battery pack that is portable and can last up to 2 hours. We chose the Anker® 2nd Generation Astro 6400 mAh External Battery Pack. The reasons for this power supply suggestion are as follows:

- Lithium Polymer battery
- 5V with 2A, enough to supply power to microcontroller
- 6400mAh is approximately 6hrs
    - (In comparison to MacBook Pro 15" with a 7000mah = 7hrs according to Apple)
- Lightweight and mobile
- Recharge time is 5-6 hours

4. Overall Design

We decided to use the HC-SR04 and the Raspberry Pi B+. After those decisions we had to use those pieces to come up with a design that would meet our design goal. It would have been nice to put 30 sensors on the drone and provide 100% coverage of the room, but that is impractical. Using 30 (this is not an accurate number) sensors would have increased our weight to 300g, which would have made our total package weight over the limit. These facts led us to the realization that we could not provide total coverage of the room. Instead we would have to organize a smaller number of sensors to provide enough coverage to avoid a collision. This led us to write a design goal.

DESIGN GOAL

“Our goal is to create a sensor package that detects enough
obstacles to protect the drone from a destructive collision, is lightweight,
cheap and easily integrated into the final design.”

Everything about our design has to compromise between accuracy, speed, coverage and weight, cost and difficulty. In order to do this we decided to use 10 sensors, this would give 1 up and down and then 8 in the horizontal plane. Our initial concept was to space the out the 8 sensors evenly with 45º between them. Below you can see two of our concept drawings of this layout.

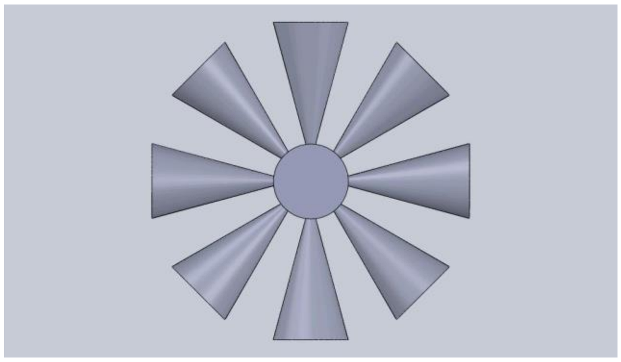


Figure 4.1: Top View

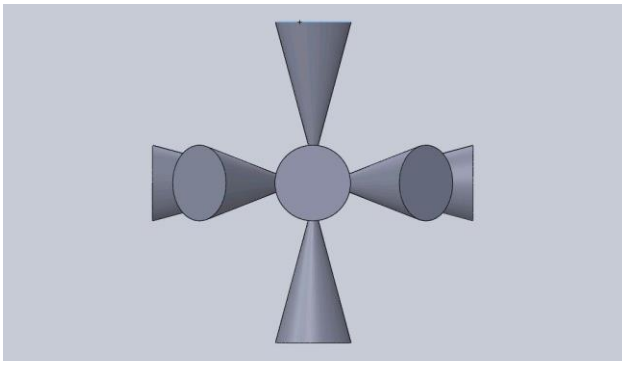


Figure 4.2: Front View

We eventually dropped this layout because it would be extremely difficult to convert these dimensions into a rectangular prism of clear space. More about the final layout of the sensors can be found in the sensor layout section. That change occurred after we did the rest of our design, not the actual layout affected it much. So we will continue talking about the big picture of our design with how we planned to get a distance measurement from an individual sensor.

The HC-SR04 has 4 pins VCC, GND, TRIG and ECHO. Simply put the VCC and GND pins power the sensor and TRIG starts the sensor looking for an obstacle (sends out an ultrasonic pulse) and ECHO stays HIGH until the pulse is received back. In other words, setting TRIG to HIGH sends an ultrasonic pulse and the ECHO pin helps keep track of how long it takes the pulse to hit an object and return. The way in which we get a distance from this is by starting a clock when ECHO goes to HIGH and then stopping the clock when ECHO goes to low. The time in the clock is the time it took for the sound wave to travel to the object and back. Since we know how fast sound travels we can calculate the distance traveled easily with Equation 1.

$$Distance = time \times speed$$

Equation 1: Distance calculation

For more information about the HC-SR04's pins, take a look at our document on the HC-SR04. For more information on how we calculate the distance based off the pins please look at the wiring and code sections of this report.

The final part of our design is how we take the sensor information, process it and send the six numbers that define a rectangular prism to the control module. The detailed information for this can be found in the Code section of this report. The overview is this, when running a single sensor we picked up time outs once or twice per hundred readings. As we add more sensors operating together the amount of noise increases quite a bit. We explain this phenomena more in the HC-SR04 document. For now let it suffice to say that there is noise in our raw readings. We wish to remove as many of these erroneous readings as possible because if they are passed on to the controls team uncorrected they could cause unexpected behavior.

In order to clean up the erroneous readings we decided to filter our data. We tried three different methods to do this:

1. Moving median
2. Exponential moving average
3. Standard deviation

We tested all of these methods and they helped, each with its own strengths and weaknesses. The process that provided the most accurate, smoothest data with the most noise eliminated was running all the filters in conjunction. More detail on all of these noise filters can be found in the code section of the report (7.4).

5. Wiring Design

5.1 Wiring a single sensor

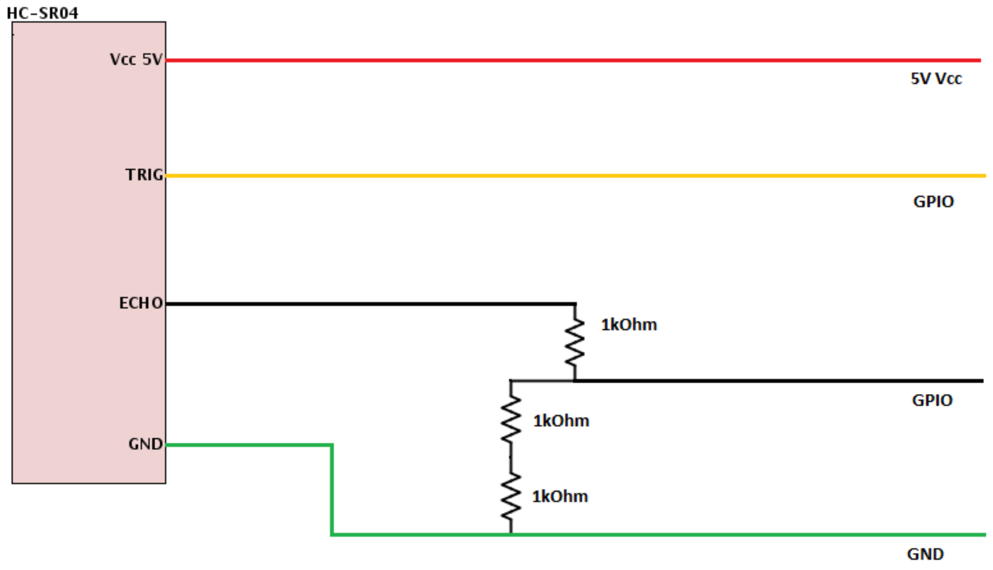


Figure 5.1.1: Initial wiring of a single HC-SR04

The HC-SR04 requires 4 connections to a microcontroller to function, these connections control the VCC, TRIG, ECHO, and GND pins described in the previous section. They are shown in **Figure 5.1.1** above. The voltage divider circuit shown on the ECHO pin is necessary due to the maximum 3.3V that the Raspberry Pi GPIO pins can handle, the HC-SR04 sensor returns 5V when an ECHO is returned. The resistance values shown are 1kOhm, but these can change as long as the 2:1 ratio is maintained. This is the initial wiring diagram we followed, but when we changed over from 5 to 10 sensors we switched from this parallel voltage divider to a simple series resistance shown in **Figure 5.1.2** below. This allows for the use of less resistors and wires. As a result, making the final product cheaper, much simpler to assemble and troubleshoot while not changing the functionality of the sensor.

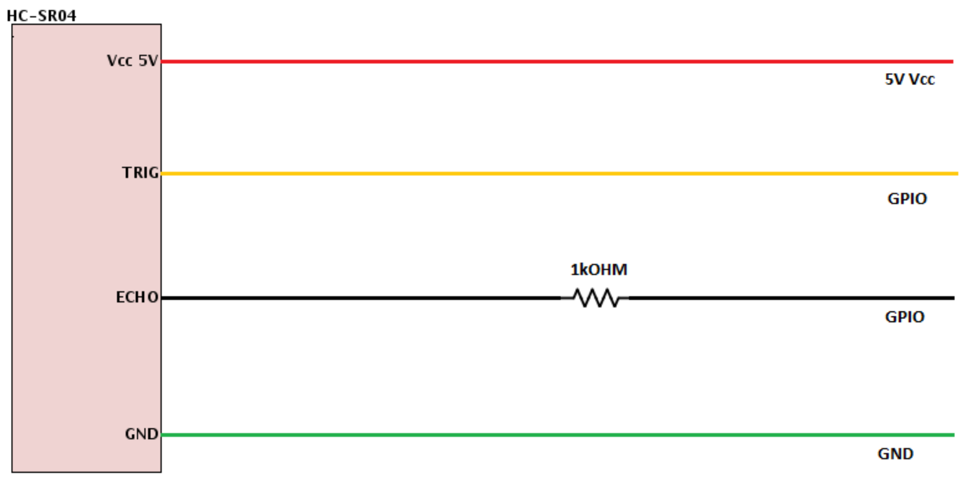


Figure 5.1.2: Final wiring of a single HC-SR04

5.2 Wiring multiple sensors

When wiring multiple sensors there are a few things to consider. The voltage and grounds needed by the sensors is the same for each one. Depending on the needs of the project multiple HC-SR04 sensors can be triggered at the same time using a single GPIO pin from the microcontroller. In order to cut down on weight, wire clutter, and the number of GPIOs required the sensors share common ground, voltage, and trigger pins. The circuit for five sensors is shown in **Figure 5.2.1** sensors can be added using a new GPIO for each new ECHO.

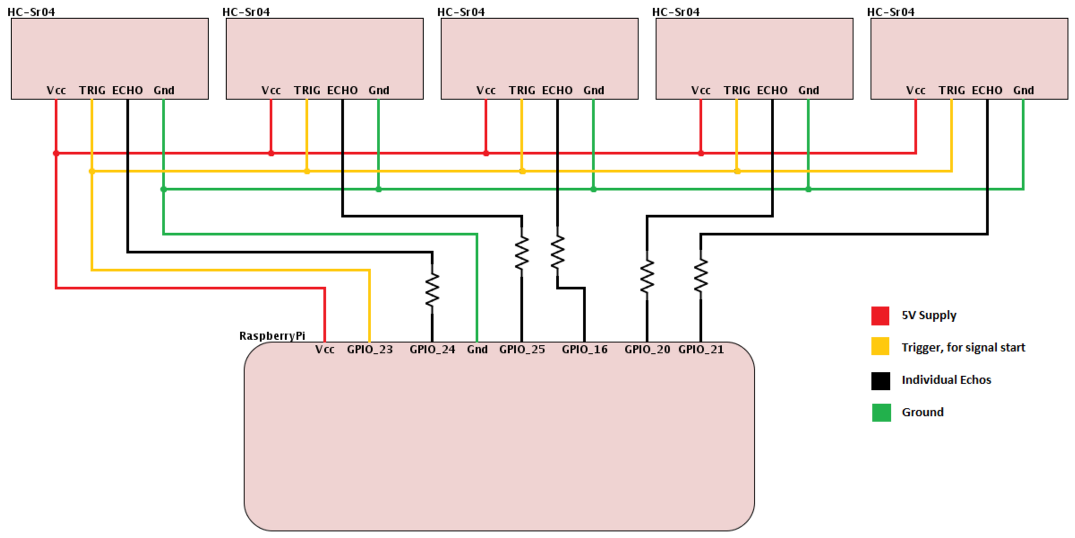


Figure 5.2.1: Circuit for five HC-SR04 sensors

To manage all of these connections, the circuit was soldered onto the PCB as shown in **Figures 5.2.2 & 5.2.3**.

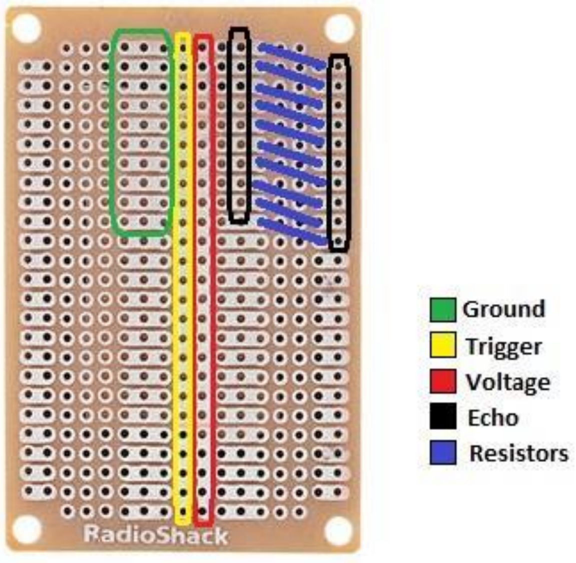


Figure 5.2.2

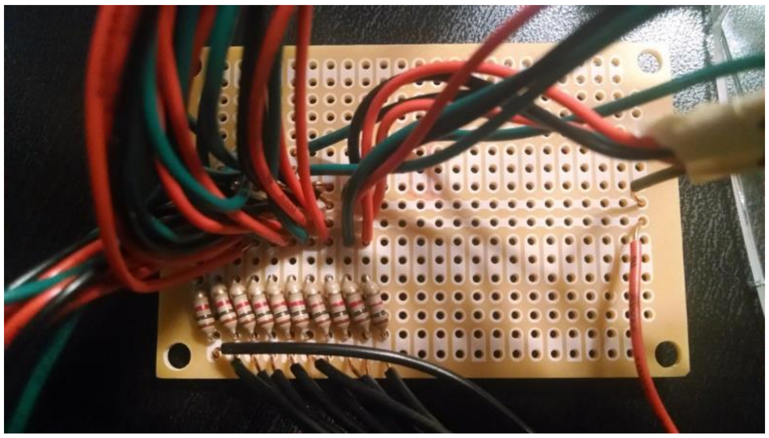


Figure 5.2.3

5.3 Timing of Sensor (HC-SR04)

The timing diagram figure is shown below. The sensor has 4 important pins, VCC, TRIG, ECHO, and GND. In order for the activation of sensor, supply the voltage to TRIG pin and set it to high for 10µS impulse. Once the trigger input is set to high, the signals are sent and the module will send out 8 cycle of ultrasound at 40 kHz in order to set the ECHO pin to high. The ECHO pin is set to high when the signal has reached back to the sensor. Once the signal has reached, the distance is determined based on the difference between the starting and the ending time. **Equation 2** shows how to compute the distance in microseconds per centimeter.

$$Distance = \frac{Speed}{170.15\ m} \times \frac{Meters}{100\ cm} \times \frac{1e6\ \mu S}{170.15\ m} \times \frac{58.772\ \mu S}{cm}$$

Equation 2: Distance Calculation

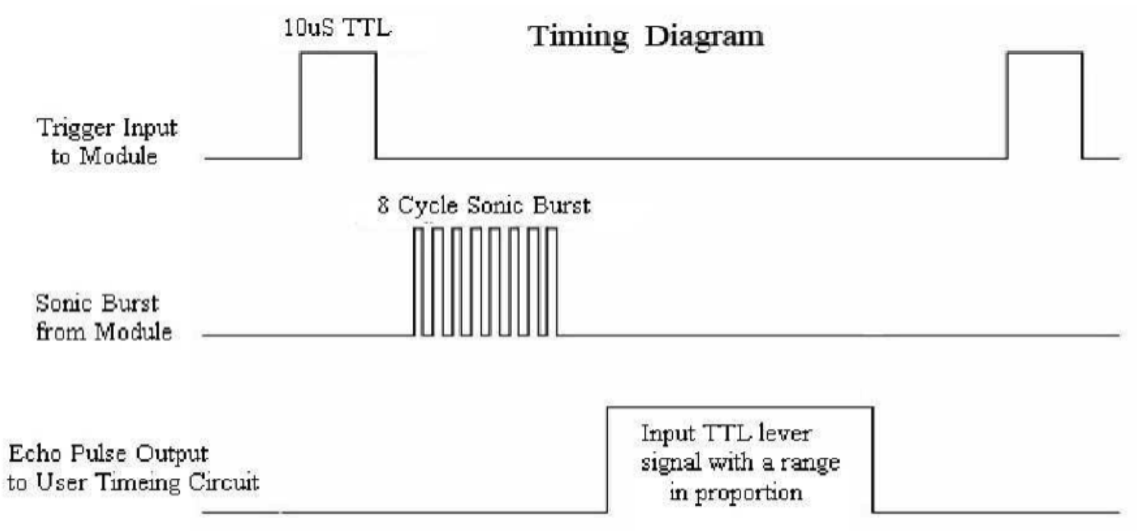

10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion

Figure 5.4.1: Timing Diagram
Source 2

6. Sensor Layout

6.1 Test Platforms and Mounting Method

A test platform was needed to contain the microcontroller, printed circuit board (PCB), power supply, and mounting points for multiple sensors. The test platform was needed to allow for multiple mounting angles so that the sensors could be tested in different configurations. The initial design is shown in **Figure 6.1.1**(below), it utilized octagonal rings to give flat surfaces for easy mounting. This also left space in the middle where a shelf of some kind could be added for the non-sensor hardware as well as wire management.

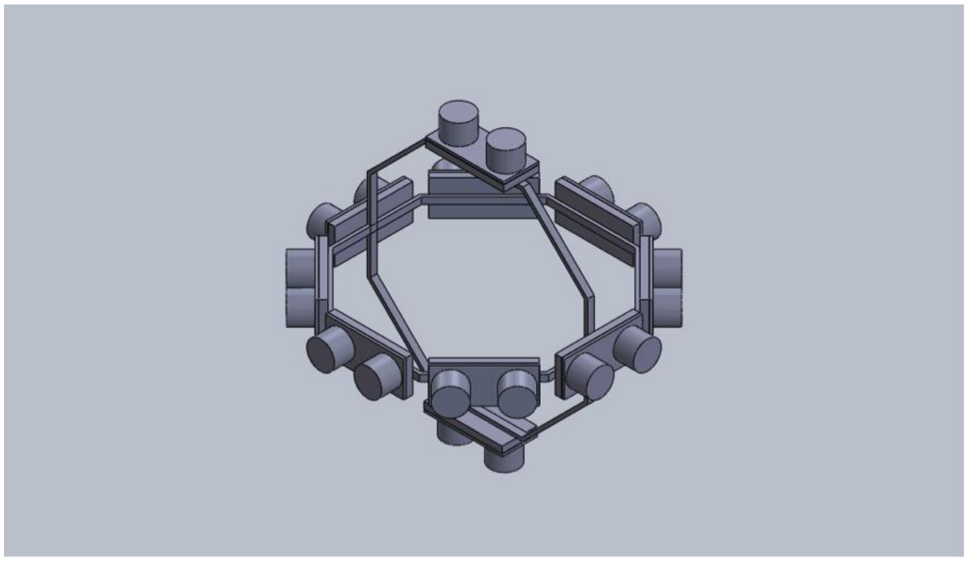


Figure 6.1.1: Sensor Layout

While comparing sensor layouts we considered the drone operation. The conclusion was that it would maneuver horizontally, as it is likely to be indoors, and navigate at a constant height. Therefore, the best resolution of the horizontal plane is required. At this stage of planning, the HC-SR04 sensor was already utilized as the model for the designs. This sensor has a detection angle of 15 degrees off of center, along with approximating with 30 degree cones to provide a visual reference of what the sensors would see. Dispersing eight sensors around the platform with a single sensor pointed up and one pointed down reference shown in **Figure 6.1.2**(below). This is to allow a good coverage in front, behind, and next to the platform. This design was rigid and did not allow for many alternate locations for sensors. The final design needed to be more modular.

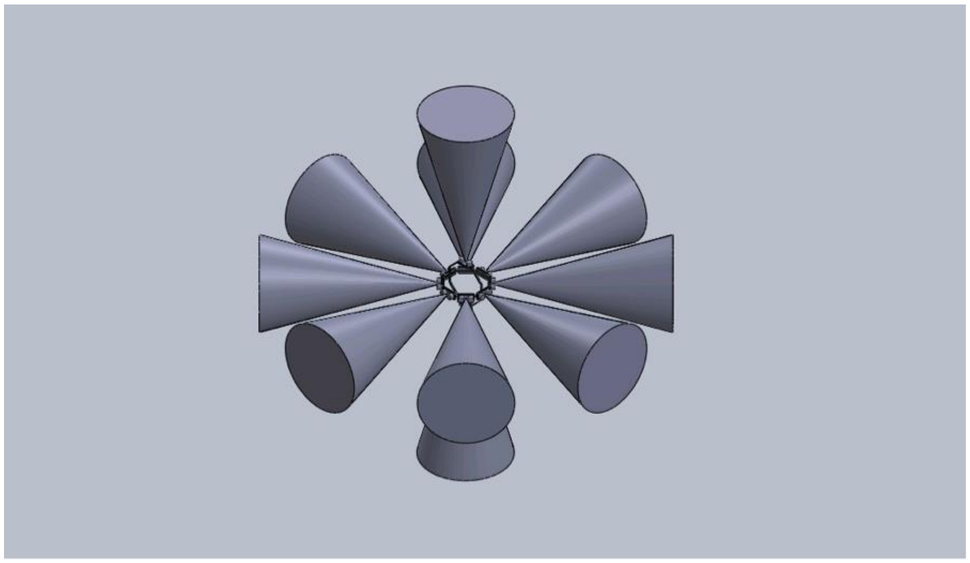


Figure 6.1.2: Sensor Projected Angles

The construction of first platform was simpler than the original designs. It was built to hold four sensors and work as a proof of concept for working with multiple sensors. Platform 1.0 was a simple square made of cardboard which was used to get the initial single sensor readings against various surfaces. Once the other three sensors were added objects closer than 30cm would sometimes show up in the readings from the sensors on either side of the sensor that was really facing from the object. To prevent this a set of cardboard fins were added off of each corner to block the reflected waves. Later a second set was added to improve the results. Platform 1.2 is shown in **Figure 6.1.3** after fins had been added. These removed a substantial amount of interference, but the shielding was somewhat ineffective in certain situations especially once a fifth sensor was added pointing straight up. The fifth sensor caused interference with all of the other sensors as the ultrasonic pulse came back off the ceiling.

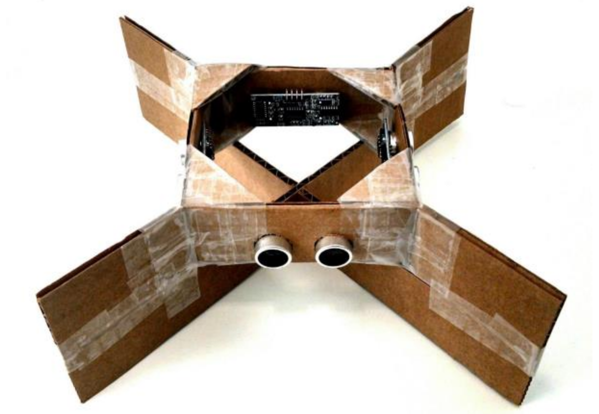


Figure 6.1.3: Test Platform 1.2

Test platform 2.0 was designed as a box that would have space for all non-sensor hardware inside and mounting surfaces on all six sides of the platform. To organize the hardware and wires shelves were included inside of the platform, these are visible in **Figure 6.1.4** (below).

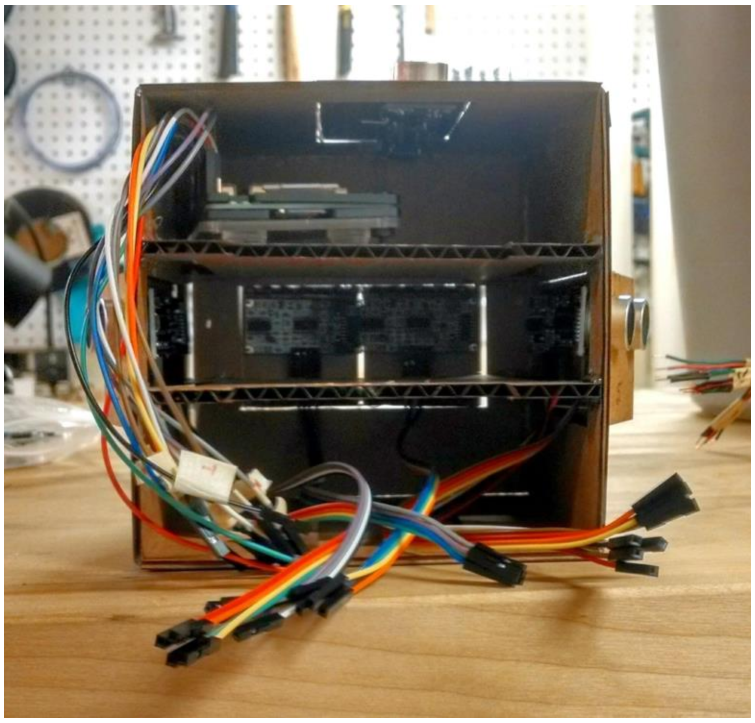


Figure 6.1.4: Test Platform 2.0 inside view

The mounting locations on the outside of the platform had space for two sensors each. There were bad results when the sensors were hitting a wall at an angle of 30 degrees or more from perpendicular to the sensor. These were caused by the ultrasonic waves reflecting off of the surface and not returning straight back to the sensor. To clean up the measurements in this case we planned to put two sensors side by side but offset from each other by some angle as shown in **Figure 6.1.5**.

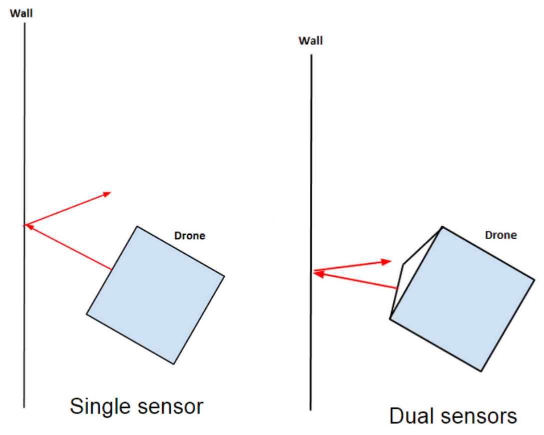


Figure 6.1.5

This would allow one of the sensors to always be pointed enough towards an object to see it even if that object was at a steep angle to the platform. Each side of the platform had a pair of sensors ranging from 5 to 25 degrees apart in increments of 5 degrees this is visible in **Figure 6.1.6** which shows the platform before any shielding had been added to the sensors.

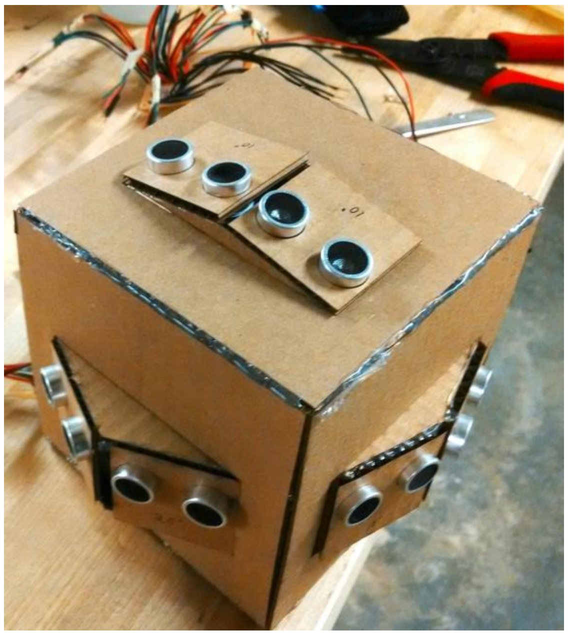


Figure 6.1.6: Platform 2.0 external view

(Sensors visible are 10 degrees apart on top, 25 degrees apart facing left, and 15 degrees apart facing right.)

The angles between the sensors were chosen because they were plus or minus 10 degrees from 15 degrees. An angle of 15 degrees between sensors means that their 30 degree fields of view should line up with one another without any overlap. The range on either side of 15 degrees allowed testing to see if a slight overlap or a gap between the sensors would give better detection over all. After each pair of sensors were pointed both perpendicular to a wall and at 45 degrees to the wall, the 15 degree gave the most complete coverage in both cases. Using the 15 degree offset 3D printed permanent mounts with built in shielding were designed and made. The temporary cardboard shielding and one of the 3D printed mounts are visible in **Figure 6.1.7**.

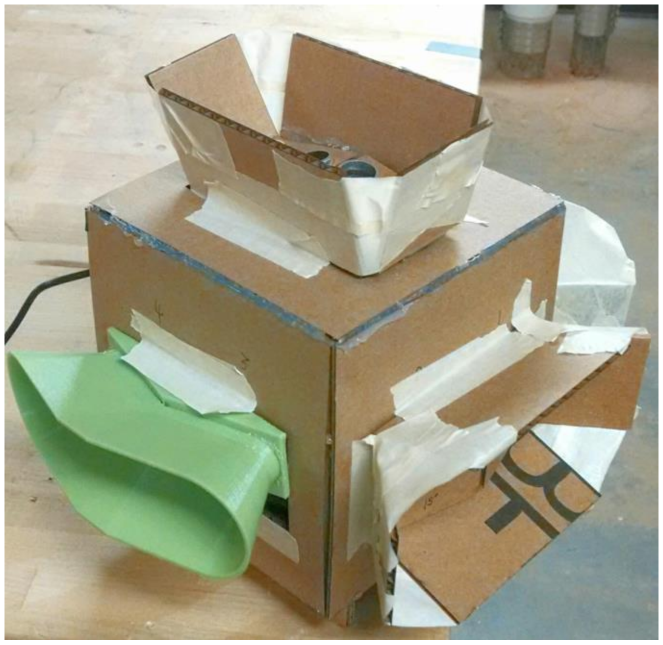


Figure 6.1.7: Platform 2.2

(3D printed mount is green and visible on the left)

To give space for moving sensors around and performing final testing, a circular platform was designed and built. Platform 3.0 was a disk cut from a thin sheet of acrylic, with lines dividing the surface into eight wedges allowing sensor mounting in increments of 45 or 90 degrees. There was also a tray centered on the disk to hold all of the non-sensor hardware, and surfaces to mount sensors on the top and bottom. It is shown in **Figure 6.1.8**.

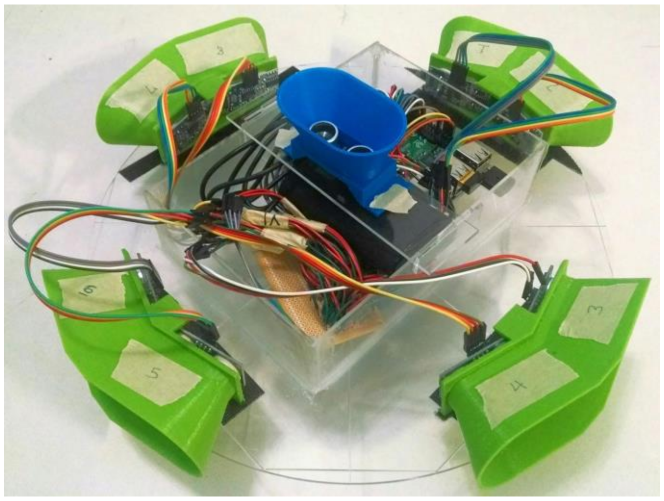


Figure 6.1.8: Platform 3.0

7. Code for Sensor Data

The problem of creating the largest possible prism for the drone to operate in has two basic requirements for the code:

1. Distances must be measured accurately
2. Measurements must be completed quickly enough for the drone to respond to the environment

Because the ultrasonic sensors used require a distance to time conversion, both of these requirements require the microcontroller to be able to run our code as quickly as possible.

7.1 Operating system setup

The first issue we had to deal with was getting real-time system performance out of the microcontroller. For time-critical problems like sensing obstacles, it is important to get as close to real-time results as possible. Although modern computers seem to perform simple tasks and multitasking nearly instantly to the user, each core in a computer's CPU can only execute one task at a time. To achieve the illusion of multitasking, the operating system performs what is called process scheduling. The base layer of the OS, the kernel, performs this task to switch back and forth between processes very quickly. This switching looks instantaneous to the user, but in reality, not every process is executed as fast as it possibly can be. Often, the tasks performed by the operating system's kernel also take precedence over user operations. The delay that exists due to these issues is called latency.

Because both the BeagleBone Black and Raspberry Pi have single-core processors and run Linux operating systems, this idea of latency exists as the process scheduler decides what processes get priority to run. In our case, the use of the Raspberry Pi or BeagleBone Black meant we would need to use of a real-time kernel to decrease latency and give more processing time and priority to our sensor code.

We were successfully able to install the following operating systems on the microcontrollers: Raspbian on the Raspberry Pi and Debian on the BeagleBone Black. These instructions are available from sources 3 and 4 in the *Sources* section of this report.

Following is a comparison of the latency encountered in each kernel on the Raspberry Pi. We conducted these tests using the following command while running each kernel:

```
cyclictest -l100000 -m -n -a0 -t1 -p99 -i400 -h400 -q
```

Kernel version	Min latency (μs)	Average latency (μs)	Max latency (μs)
3.12.28-rt40+	13	25	62
3.12.28+	16	26	1996

Table 7.1.1: Latency comparison on the Raspberry Pi

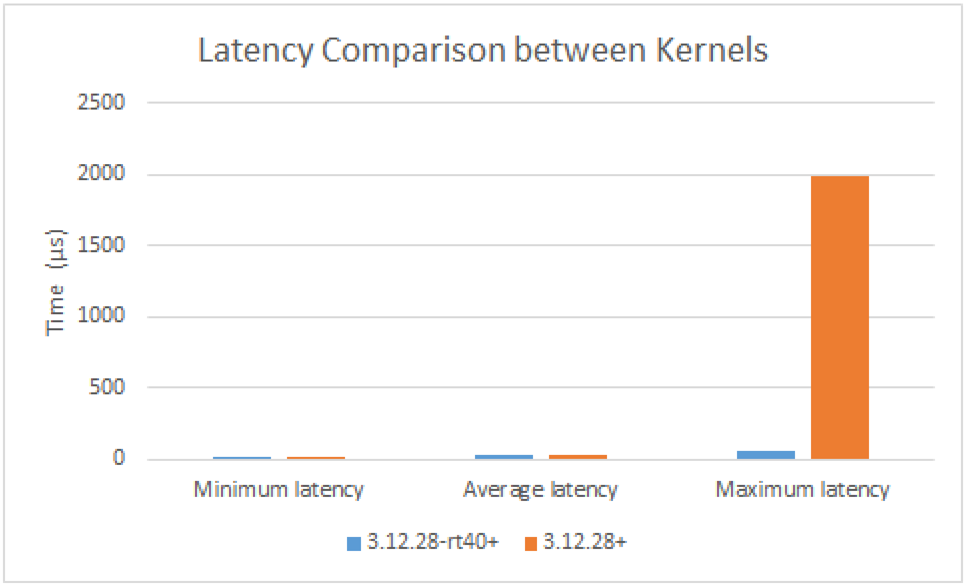


Figure 7.1.1: Latency comparison

As seen by the measurements in **Table 7.1.1** and **Figure 7.1.1**, the max latency allowed by the real time kernel over a period of 100,000 `cyclictest` samples is significantly shorter for the real-time kernel than the max latency for the default kernel. The following graph in **Figure 7.1.2** further shows the benefits of using a real-time kernel to defeat the challenge of high latency.

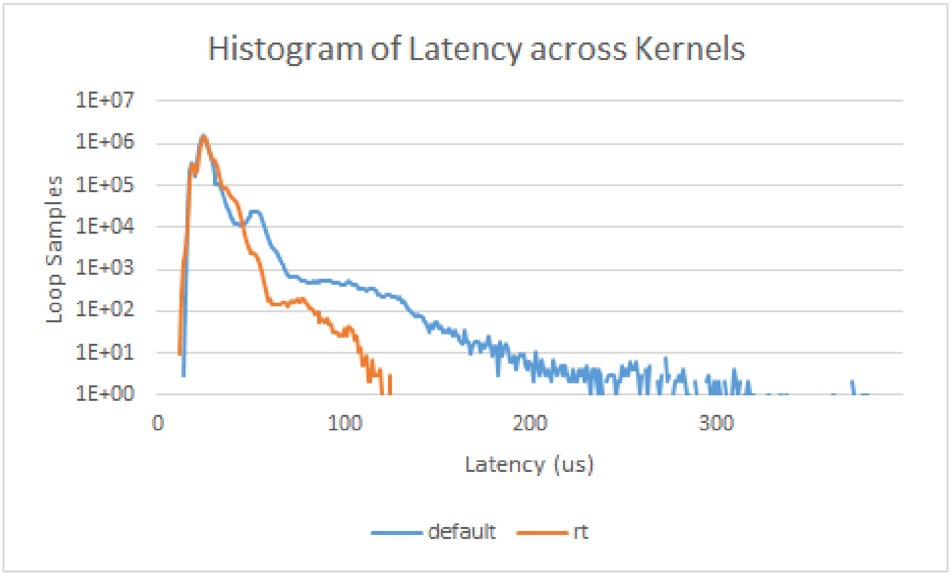


Figure 7.1.2: Overlapping histogram of latency between real-time and default kernel

We were successfully able to cross-compile a real-time kernel for the Raspberry Pi and install it on the Pi. Namely, we used the Raspberry Pi's 3.12.28 kernel source from the official git repository and applied the rt patch available from kernel.org. We followed the directions in a forum (see source 5) to do this. The version of the kernel we had success with is 3.12.28-rt40+. We have encountered no stability issues with it, which is often a problem when compiling kernels from scratch. We were also able to compile a real-time kernel on the BeagleBone Black using the kernel source from the BeagleBone kernel git repository. However, the kernel was extremely unstable and resulted in the system shutting down after ten minutes. This was one of the major reasons we decided to proceed using the Raspberry Pi.

7.2 GPIO mapping

The next step was mapping the GPIO (general purpose input and output) pins on the Raspberry Pi. GPIO mapping is needed in order to plan how the pins on the sensor connect to the microcontroller.

We decided to map the trigger pin for each of the ten sensors to a single GPIO pin on the Raspberry Pi. This would cause the ultrasonic waves to be released from all ten sensors at once. However, each ECHO pin on the sensor was mapped to a separate GPIO pin so that we could determine individual distance readings from each sensor. Information about the specific pins that we used on the Raspberry Pi can be found in Appendix B.

7.3 Algorithms and design

We then used the datasheet for the sensors to write an algorithm to determine the length (in cm) from each sensor to the nearest detected object:

```
while (take more readings)
```

```
Tell sensor to start measuring by raising TRIG pin for 10 µs
        Sensor sends out ultrasonic waves
Record start time
Hardware raises ECHO pin
Wait for 23,000 µs (timeout equivalent for 400cm), or for ECHO pin to return to low
Record end time
Subtract start time from endtime → x
Divide x by 58 to determine cm travelled
```

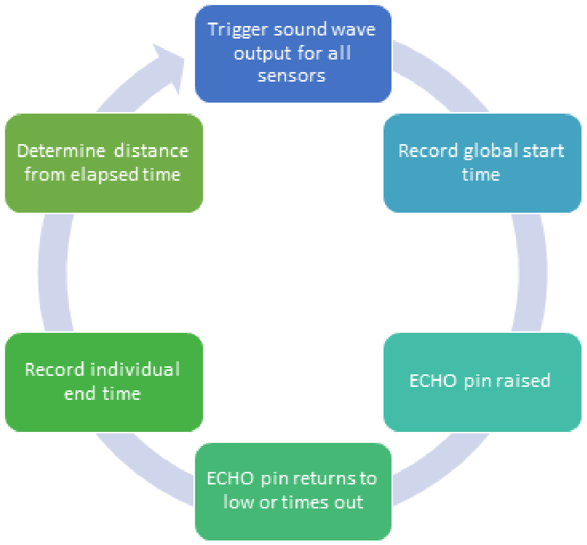

Trigger sound wave output for all sensors
Record global start time
ECHO pin raised
ECHO pin returns to low or times out
Record individual end time
Determine distance from elapsed time

Figure 7.3.1: Illustration of original algorithm

We narrowed down our programming language options on the Raspberry Pi to C and Python. Python is an interpreted language, and although it is much easier to learn and use, the extra latency was a concern for us. It is important to achieve as high a refresh rate as possible when sending the data to the control module to give feedback to the motors on the drone. We decided to use Python for the prototype development, and then transition that code to C to try to achieve the highest refresh rate possible. After developing a working solution in Python, we immediately transitioned the code to C. We elected to use the wiringPi library (http://wiringpi.com/) for access and management of the GPIO pins on the device. We chose this library due to its popularity and relative ease of use. The C code using this library is available in our GitHub repository. The filename in our source code that uses this library is `multisense.c`.

One of the main features we needed was the ability to set interrupts on the GPIO pins to determine when the ECHO pin went from high to low. When this occurred, it signaled that the sound sent during the trigger stage had returned back to the sensor. Since our distance calculation was based on time elapsed between the sending and receipt of the sound, this interrupt was vital to obtaining an accurate measurement.

We implemented this code for one sensor, five sensors, and ten sensors working concurrently with moderate success. The general algorithm for a single sensor is shown in **Figure 7.3.1**. With the one and five sensor setups, we noticed that there were several periods where the sensor timed out (no ECHO received) for several consecutive readings, increasing with distance. As we increased the number of sensors, we started getting highly inaccurate results back in certain scenarios, which was probably an amplification of the original problem.

We realized that the issue was probably happening due to the following situation:

1. TRIG pin is triggered and sound waves are sent
2. ECHO pin is raised
3. 23,000 µs is reached and program continues
4. Loop repeats while ECHO pin is still raised, which is not what the program is expecting

In this scenario, the ECHO pin is still raised when the loop goes through the second time. Although triggering the TRIG pin eventually brings the ECHO pin back down, there was not enough time allotted between steps 1 and 2 for the ECHO pin to lower again, so the falling edge interrupt for the ECHO pin occurred at incorrect times. This resulted in either very short readings or a timeout. It took the sensor several readings to break out of this loop.

While still using the wiringPi library, we were able to create a quick remedy for this by putting an extra 20,000 µs delay at the end of the loop, but we wanted to find a better solution, and began looking for alternative libraries. One of the limitations of the wiringPi library was its inability to set a pin to catch both rising and falling interrupts, and distinguish between the two. The result of this search was the pigipio library, which was designed to use a small footprint for real-time applications.

Using the pigpio library, we were able to set both rising and falling interrupts on the ECHO pin, and could distinguish between the two. The algorithm was modified to the following:

```
while (take more readings)
        Tell sensor to start measuring by raising TRIG pin for 10 µs
                Sensor sends out ultrasonic waves
        On ECHO rising interrupt (hardware raises ECHO pin), set start time for sensor
        Wait for 23000 µs, or for ECHO pin to return to low, whichever comes first
        Record end time
        Subtract start time from endtime → x
        Divide x by 58 to determine cm travelled
```

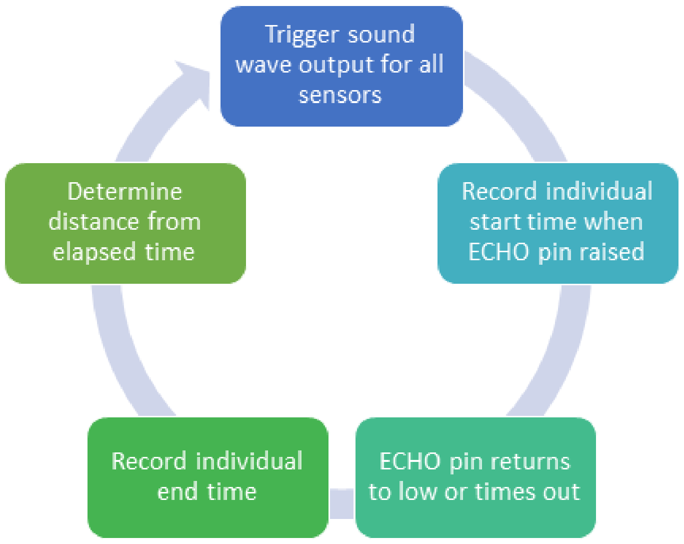

Trigger sound wave output for all sensors
Record individual start time when ECHO pin raised
ECHO pin returns to low or times out
Record individual end time
Determine distance from elapsed time

Figure 7.3.2: Illustration of modified algorithm

The distinguishing feature in this case was the addition of the start time measurement for each sensor. By recording each sensor's start time individually when each ECHO pin raised from 0 to 1, this would theoretically increase both the accuracy and precision of the calculated distance measurements, since the elapsed time would be more accurate. The modified algorithm is shown in **Figure 7.3.2**. The results returned from this code met and exceeded our expectations. Although some noise still existed, the number of extremely inaccurate results decreased significantly from our original algorithm.

7.4 Data analysis and filters

Due to the noisy nature of our data, especially while we were using the wiringPi library, we decided that it would be important to apply a software filter on the raw distance calculations we determined from the sensors. During some of our tests, we received timeouts from the sensors with objects as close as 20 cm away. This kind of noise could be fatal in an operating environment. If the drone is only 20cm away from a wall and a false 400cm reading comes in, the drone could crash into the wall.

Realizing the importance of the need for the software filter led us to look at several different methods to reduce noise:

- Moving median of five observations
- Exponential moving average
- Single point removal using range-determined standard deviation
- Combination of all of the above

Moving median of five observations

This method was devised because the results were good enough that they were correct most of the time, but sometimes we would get bad readings. We wanted to minimize the bad readings sent to the sensor. We didn't want to average the readings because doing so would skew the correct readings. Instead, we decided that using a moving median would yield the best results. This works by taking the median of the distance just measured along with the four previous measurements. It reports that median as the distance measured. This method excels at removing a single erroneous reading every couple of readings showing a general trend of the data. This method is poor at handling a variety of different readings because it will jump around abruptly. Using a moving median would also allow us to maintain the same frame rate as we would with raw data. The algorithm for this is as follows:

```
Create a five-position array
Fill the array as readings come in, putting last reading at the end, shifting left as needed
If the array is not full (less than five readings), return the current value
Else return the median of the array
```

Although this algorithm can result in a delay of up to three readings for a sudden change, it seemed to work well to eliminate the major outliers in the readings. It is less susceptible to single shocks, which gives it an advantage the EMA filter. However, consecutive shocks can cause the readings to jump around very quickly. This is visible in **Figure 7.4.1**.

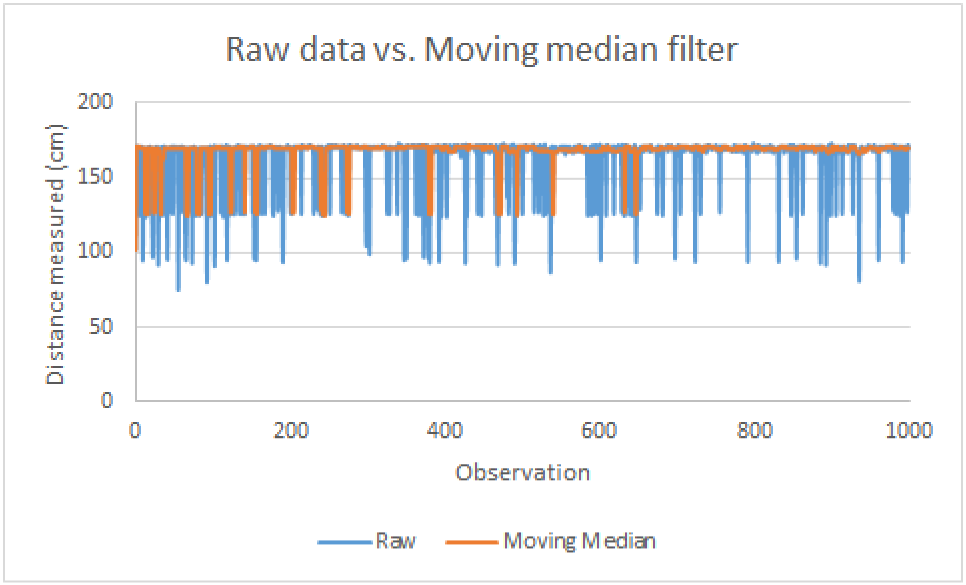


Figure 7.4.1: Raw data vs. Moving median filter

Exponential moving average (EMA)

This method was devised to help to smooth out some of the minor bad readings. EMA is a type of impulse response filter that applies weighting factors that decrease exponentially. Unlike the moving median filter, each observation has always an effect on the data returned by the filter. This means that the distance reported will trend towards the value actually measured. It is defined recursively with **Equation 3**:

$$S_1 = Y_1$$

$$\text{for } t > 1,\ S_t = \alpha * Y_t + (1 - \alpha) * S_{t-1}$$

Equation 3: Exponential Moving Average

- The coefficient α represents the degree of weighting decrease. This is a constant factor between 0 and 1. The higher α is, the faster older observations are discounted.
- Y_t is the value at a time period t.
- S_t is the value of the EMA at any time period t.

In our code, S1 is initialized by setting it to the value of the first reading. To determine the value of the coefficient alpha, we had to take into account the weight that we wanted each value to carry. We had to balance the requirement for data that is responsive to the environment with the requirement for accuracy in the readings. We determined that using an alpha value between 0.2 and 0.3 worked best for creating a smooth curve with appropriate response time. This method is very susceptible to shocks in the raw data, but it is able to smooth out some of the smaller ones and creates a good view of the general measurement trend. This smooths out abrupt jumps due to erroneous readings and at the same time provides an immediate response to most recent reading. This method's shortfall is that a single erroneous reading can cause quite an impact on the reported distance. These instances are shown below in **Figure 7.4.2**, where the orange line takes time to return to a correct reading after experiencing erroneous data.

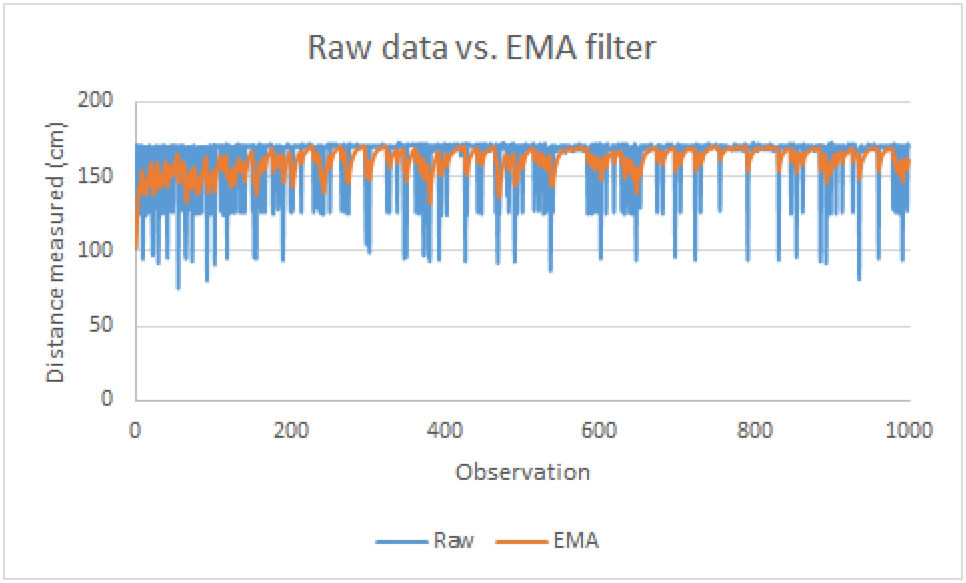


Figure 7.4.2: Raw data vs. Exponential moving average filter

Standard deviation point removal

We devised this filter to eliminate single outliers in the sensor readings. This filter only works for single outlier removal by checking if the difference between the current reading and the last reading is greater than a predetermined value. This algorithm can result in a delay of one reading if the sensors encounter a real sudden change.

Since the sensors are more accurate in close-range situations than long-range situations, the predetermined value dynamically changes based on the value of the previous readings. In other words, the filter is stricter for short-range data than for long-range data. This is implemented by increasing the predetermined value for long range data.

Although this filter is good for eliminating single values, consecutive bad readings are not handled and can result in shocks that are equally as large as the raw data. This point removal method is shown below in **Figure 7.4.3**. The filter completely removed several bad data points, but in instances where there were multiple bad readings they were left in because the filter interpreted them as good.

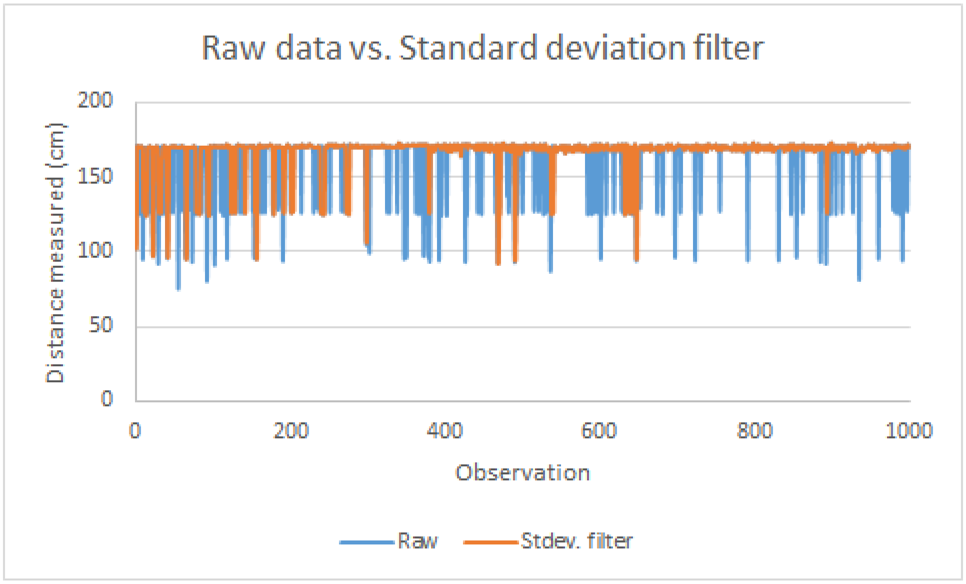


Figure 7.4.3: Raw data vs. Standard deviation filter

Combination of all filters

The combination of all the filters results in very consistent, smooth data. First, the data is run through the standard deviation filter to eliminate large shocks. Next, the data returned by that is run through the median filter to eliminate anomalies not picked up by the standard deviation filter. Finally, the data is run through the EMA filter to smooth out any shocks resulting from the previous filter. Although this method returns the most consistent, smooth data, it also takes several milliseconds longer for all the data processing to happen. The graph below in **Figure 7.4.4** shows the scrubbed data.

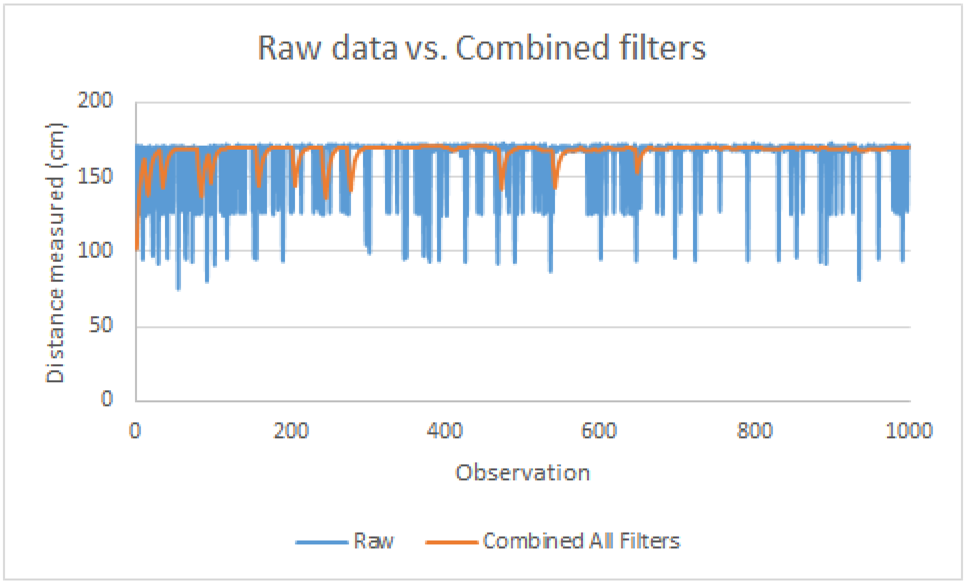


Figure 7.4.4: Raw data vs. Combined filters

Filter Summary

Each of these filters has its advantages and disadvantages. It is up to the user of the code to determine which filter (if any) best fits the situation.

7.5 Data consolidation and output

As seen in the layout of our sensor package, each side of the prism has two sensors on each side separated by a 15 degree angle. The top and bottom only have one sensor. The readings returned by these sensors are the top and bottom measurements for the prism. Since each side has two sensors, we take the minimum of the two values returned to determine the amount of space available on that side of the drone.

In other words, data from the ten sensors is consolidated in the following manner:

- Front = Minimum of Sensor 1 and Sensor 2
- Back = Minimum of Sensor 5 and Sensor 6
- Right = Minimum of Sensor 3 and Sensor 4
- Left = Minimum of Sensor 7 and Sensor 8
- Up = Sensor 9
- Down = Sensor 10

The output prism sent to the control team consists of six integers. The data is sent in groups of these six integers over a named pipe (FIFO) in the operating system. We chose to use this mechanism because we assumed the control and sensor code would be running concurrently on the same microcontroller. Interprocess communication (IPC) via named pipes in this scenario is much faster than sending the data over PWM. The integers are sent over this FIFO in the following order: front, back, right, left, up, and down. When the process terminates, the value -1 is sent over the pipe to indicate that the pipe can be closed on the input end. Sample C code that demonstrates reading data on the input side is included in `pipe_in.c`. For additional reference on FIFO IPC communication, see source 6.

8. Testing

We needed to determine the number and placement of sensors to use in our final layout. To do this we needed to understand how a single sensor behaved in as many situations as we could conceive encountering, and also how adding other sensors in various arrangements would affect the results. This section covers the process of how we gathered data. We then evaluated this data in making design decisions.

8.1 Single Sensor: Distance Testing

Our first tests were very straight forward. A sensor was pointed at a wall 1 meter away and left stationary, then it was moved back to 2 and finally 3 meters. This test was to check that the sensors did work, and also to check our math on the distance calculations. The results were reasonably consistent and accurate.

8.2 Single Sensor: Material Testing

From our research we knew that the acoustic properties of different materials could affect the results as an acoustically absorbent material could cause long or inconsistent results. We tested against a wall as a control test of a hard non-absorbent material, then against the back of a couch as a softer but still dense material, and finally against a window curtain with the window open. All of these measurements were made in a static 1 meter test and the results are shown in **Figures 8.2.1 - 8.2.3**.

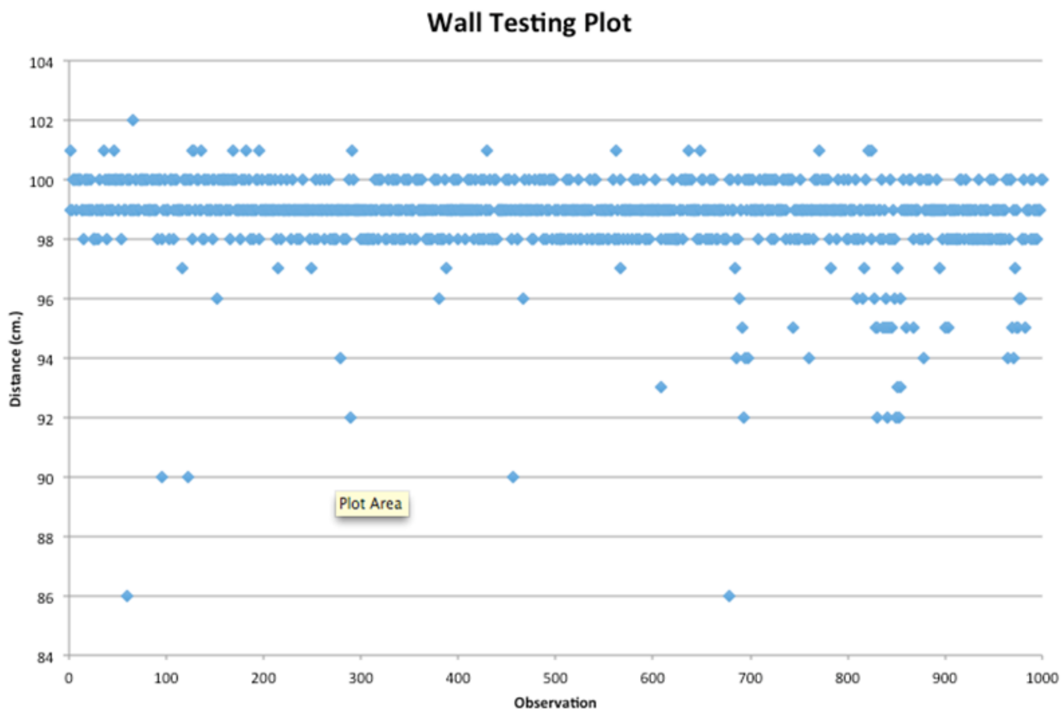


Figure 8.2.1: Wall at 1 meter

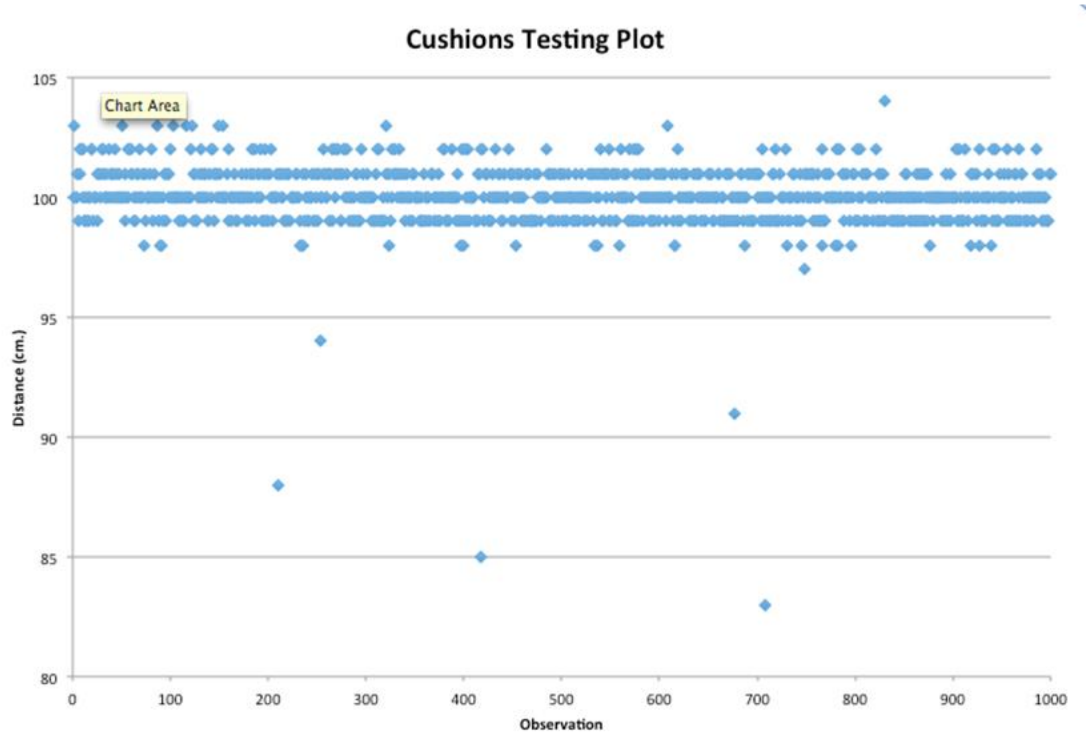


Figure 8.2.2: Cushions at 1 meter

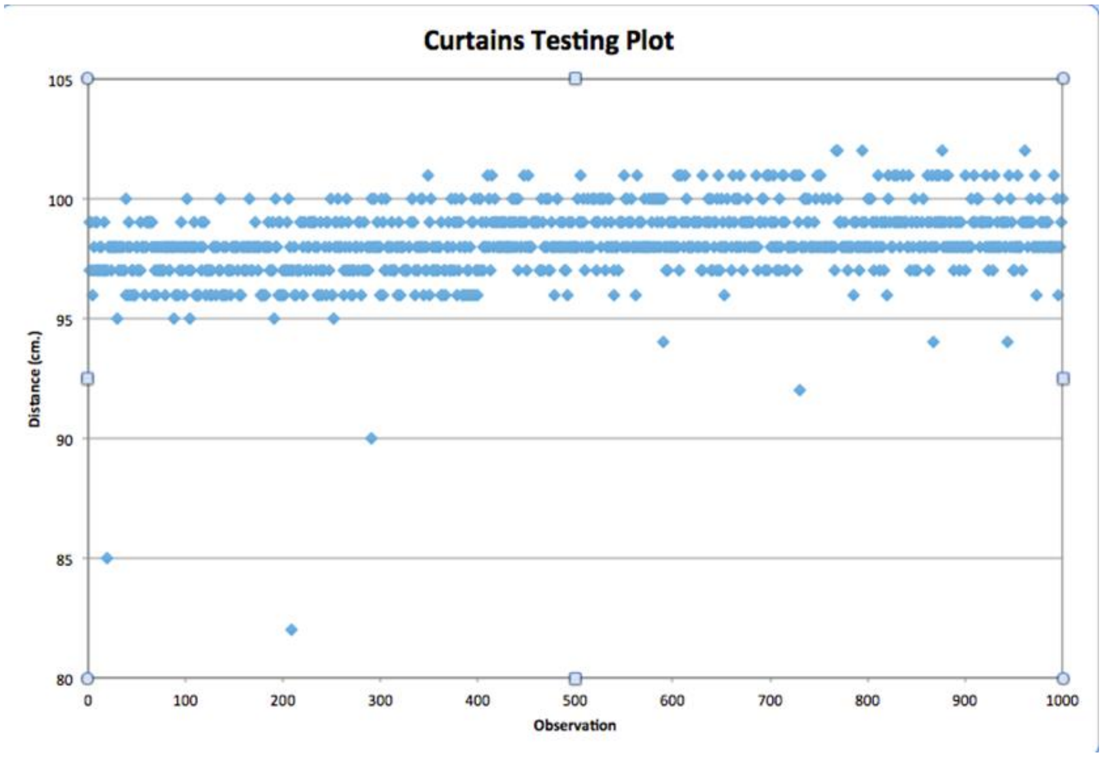


Figure 8.2.3: Curtains at 1 meter

8.3 Single Sensor: Angle of Detection Cone

Ultrasonic sensors have an advertised angle that represents a cone that the sensor can detect objects in. For the HC-SR04 this angle is 30 degrees, but this number is slightly misleading. It implies that the cone is constant. The cone actually varies based on distance from the sensor. Up close to the sensor the cone is wider than it is at the maximum range of the sensor. This effect is shown in **Figure 8.3.1** below.

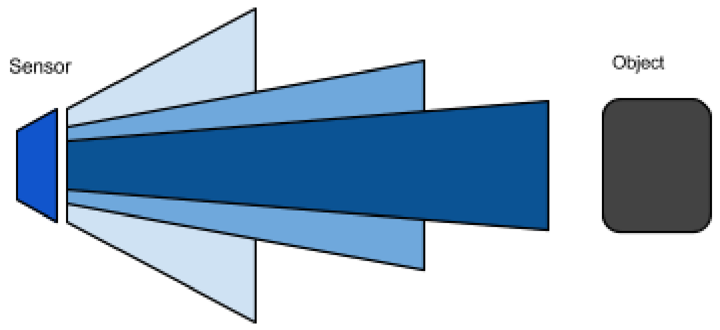


Figure 8.3.1: Angle of detection

To find out what these regions looked like we set up a sensor in some open space with marks in front of it at 1, 2, and 3 meters. Then at each distance we rolled a square object in from the side and marked the first point that the sensor recognized it was there. The values for these tests are shown below in **Table 8.3.1**.

Distance	**1m**	**2m**	**3m**
Angle of detection	**70°**	**26°**	**13°**

Table 8.3.1: Angle of detection

8.4 Single Sensor: Heat Testing

As the final goal of the project is to prepare a drone for flight in a burning building we researched the effects of heat on the speed of sound waves through air. Temperature and velocity are related as shown in **Equation 4**. This shows that as the temperature increases so does the speed of sound waves. Reading the distance of an object that is some distance away with some form of heat source that is far enough above room temperature should result in a closer reading than the actual distance.

$$v = 331\, m/s \; + \; 0.6\, m/s/C \; \times \; T$$

Equation 4: speed of sound propagation related to temperature *T* in Celsius

To test this we placed a sensor in front of a fireplace and measured to the back wall of the fireplace before lighting the fire, and afterwards. We measured about 80% of our initial distance when the fire was burning. This false short reading should act as a buffer, convincing the drone that it is closer to objects that are on fire than it actually is.

8.5 Multiple Sensors: Doorway

It is important that the drone be able to fly through doorways. We tested this scenario by walking our sensor platform through a door to test how it would see the door. The forward facing sensor pair saw the door as a wall until the platform was 61-65cm from the door at which point the sensors “saw through” the door. The range at which the sensor package “sees” through the door can be increased by using a single sensor. This is because there would be a narrower cone of detection allowing the sensor to see through the door earlier. However, this would create larger blind spots and limit the detection of object that were at an angle to the front of the drone. Our conclusion was that the sensor pair worked fine and the drone could go through a doorway, but it would have to slow down to do so.

8.6 Blind Spots in Final Sensor Geometry

With our final sensor layout there are blind spots on all four corners in the horizontal plane. During our testing we found these could be reasonably large at three meters, but at closer distances from a meter inwards the blind spots are small. Items such as floor lamp posts and table legs can remain in a blind spot if they are approached diagonally. We considered adding more sensors to check these locations but there are two problems with this. The first is the fact that the more sensors operating the lower the accuracy of the readings. The second is that a sensor in the corner is extremely difficult to incorporate into the prism. These problems led us to leave the blind spots as they should be minimal.

9. Total Cost

Item	**Unit Cost (USD)**	**Quantity**	**Total Cost (USD)**
Raspberry Pi Model B+	35	1	35
HC-SR04 sensor	2	10	20
Jumper wires	8	2	16
Battery pack	30	1	30
Sensor shielding	N/A	6	N/A
Acrylic sheet	10	1	10
Resistors	0.15	10	2
Total			**$113**

Table 9.1: Total Cost

10. Suggestions

These are some of the ideas we had to improve the sensor module or improve the drone upon integration that we did not have time to implement.

- Integrate an IMU to help clean up data.
    - Determine if the drone is moving, based on that estimate how much readings should change. Use this to remove bad readings, e.g. drone is not moving, all measurements should stay the same.
- Use two separate triggers, one running the left and right sensors, and the other running the front and back sensors to eliminate noise.
    - This would help with reducing bad readings caused by echoes
    - This would slow down the overall FPS
- Use another type of ultrasonic sensor that operates at a different frequency to reduce echo noise.
- Use a different type of sensor in certain regions, e.g. laser sensor pointing forward to help find doorways.
    - Use IR sensors in the corners to lessen the blind spots
    - Use directional thermal sensors to detect fire
    - Use 3 lasers on the front to help find doorways, integrate this data with the user interface in the video to give a visual representation of where there is clear space in front of the drone
- Use the sensor shielding as a bumper on the drone.

11. Conclusion

The sensor module has several criteria to balance when reporting the clear space about the drone. These are summarized below.

- Accuracy and speed when detecting obstacles within zero to three meters
- Weight and cost
- Capable of operating in a smoky and hot environment

The sensor module we have designed can detect obstacles between 2-400 centimeters and it gives accurate results between 15 and 29 frames per second. The weight optimized sensor module should weigh in at 200 grams and cost roughly $100. The ultrasonic sensors were chosen because they have the ability to operate in a particulate heavy environment (smoke) and the ambient heat causes sound to travel faster. This will cause the sensors to report a shorter distance than is accurate, but obstacles will still be detected, the distance will be shorter than is true. The sensor module we have designed meets the overall requirements. It does have several weaknesses, but we believe this sensor module is an appropriate balance between the different criteria.

Sources Cited

1. “HC-SR04 User's_Manual.” *docs.google.* Cytron Technologies, May 2013 Web. 5 Dec. 2009. <https://docs.google.com/document/d/1Y-yZnNhMYy7rwhAgyL_pfa39RsB-x2qR4vP8saG73rE>
2. “Attiny2313 Ultrasonic distance (HR-SR04) example.” *CircuitDB*. n.a. 7 Sept. 2014 Web. 5 Dec. 2014. <http://www.circuitdb.com/?p=1162>
3. “Installing Operating System Images.” *RaspberryPi.* Raspberry Pi Foundation n.d. Web. 5 Dec. 2014. <http://www.raspberrypi.org/documentation/installation/installing-images/>
4. “BeagleBoardDebian.” *Elinux.* n.a. n.d. Web. 5 Dec. 2014. <http://elinux.org/BeagleBoardDebian#eMMC:_BeagleBone_Black>
5. “CONFIG_PREEMPT_RT on Raspberry PI.” *Raspberrypi.* n.a. n.d. Web. 5 Dec. 2014. <http://www.raspberrypi.org/forums/viewtopic.php?t=39951>
6. “Inter-Process Communication.” *TLDP.* n.a. n.d. Web. 5 Dec. 2014 <http://www.tldp.org/HOWTO/RTLinux-HOWTO-6.html>

Appendix A: Source Code Retrieval, Compilation, and Usage

Our code is hosted on the public GitHub server. Basic tutorials for using Git are available at https://try.github.io/. To check out the source code for this project, run the following command:

```
$ git clone https://github.com/arharvey918/turbo-sense.git
```

The C code used for the final design is located in `turbo-sense/hcsr04/raspberry-pi`. A description of the files can be found in the corresponding README files in each directory.

There is a file in this directory called `sensor.cfg` that contains configuration parameters for running the code.

To run the code using the output pipe for six-integer output, it is necessary to open two SSH sessions. This is because the sensor code will not send any output through the FIFO until another program connects to the other end of the pipe. If you do not enable the pipe in `sensor.cfg`, you only need to execute the commands in SSH Session 1.

SSH Session 1:

```
$ cd turbo-sense/hcsr04/raspberry-pi
$ make
$ sudo ./multisense_rt
```

SSH Session 2 (only if the pipe is enabled in `sensor.cfg`):

```
$ cd potential-hipster/raspberry-pi/c_code
$ make pipe
$ ./pipe_in
```

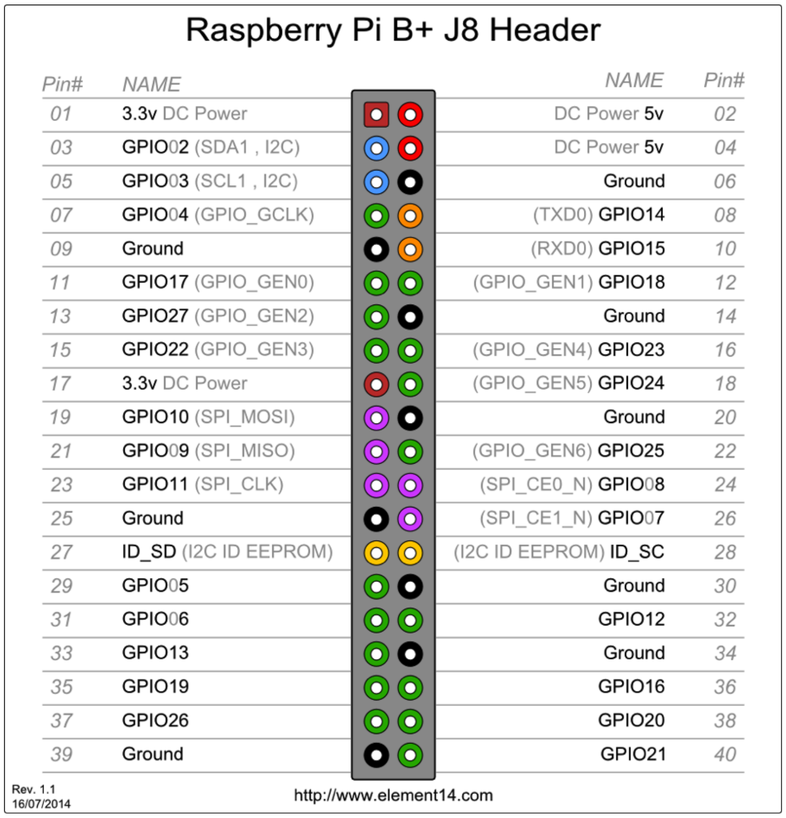
The output in session 1 will show the distances measured for each of the 10 sensors along with the timestamp each measurement was taken. The output in session 2 will read from the pipe and show the distances measured on each side in the following order: front, back, left, right, up, then down.

The code is well-documented in the each file’s comments. See each file for specifics on what the code does.

Appendix B: Sensor connection to Raspberry Pi

The next step was mapping the GPIO pins on the Raspberry Pi. The figure below has the GPIO pin information, along with the Broadcom numbered layout for the GPIO pins on the device. This was utilized to map the GPIO pins for the layout:

- 02 [5V]
- 06 [GND]
- 23 [TRIG]
- 24 [ECHO]
- 25 [ECHO1]
- 16 [ECHO2]
- 20 [ECHO3]
- 21 [ECHO4]
- 12 [ECHO5]
- 18 [ECHO6]
- 04 [ECHO7]
- 17 [ECHO8]
- 27 [ECHO9]



Raspberry Pi B+ J8 Header

Pin#	NAME	NAME	Pin#
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I2C)	DC Power 5v	04
05	GPIO03 (SCL1 , I2C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)	(I2C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

Rev. 1.1
16/07/2014

http://www.element14.com

Figure 5.3.1: GPIO Pins Layout

Appendix C: Raspberry Pi Model B+ Specifications

CPU	700 MHz Low Power ARM1176JZFS Applications Processor
GPU	Dual Core Processor Provides Open GL ES 2.0 hardware-accelerated OpenVG 1080p 30 H.264 high profile decode
MEMORY	512MB SDRAM
OPERATING SYSTEM	Boots from Micro SD card, running Raspbian
DIMENSIONS	85mm x 56mm x 17mm
POWER	Micro USB socket 5V, 2A
ETHERNET	10/100 BaseT Ethernet socket
VIDEO OUTPUT	HDMI (rev 1.3 & 1.4)
AUDIO OUTPUT	3.5mm jack, HDMI
USB	4 x USB 2.0 Connector
GPIO	Connector 40-pin 2.54 mm expansion header: 2x20 strip. Providing 27 GPIO pins, +3.3V, +5 V and GND supply lines.
CAMERA CONNECTOR	15-pin MIPI Camera Serial Interface (CSI-2)
MEMORY CARD SLOT	SDIO