Clinton Bess

Dr. Venkatesan Muthukumar

2-D Mapping with the BeagleBone Black

CPE 403

UNLV DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Checklist for Final Project

- ☑ *Working Javascript code*
- ☑ *Javascript code with in depth explanation of every section.*
- ☑ *Flow chart of Javascript code*
- ☑ *Screenshots of debugging process along with pictures of actual circuit*
- ☑ *Video link of demonstration.*
- ☑ *Schematic of project*

Introduction

Today there is an increasing demand for 3-D applications. With 3-D processing we are able to technologies that make virtual realities seem more and more possible. Also, the growing market for 3-D printers call for more economic solutions for 3-D capturing. However, before we can effectively work with 3-D data it is essential that we are able to process 2-D data properly. There are many ways to do this, while sonar and lidar implementations are most common.

For my project I implemented a 2-D mapping system by using a stepper motor, a LIDAR-Lite lidar sensor, a BeagleBone Black, and ULN2003A stepper motor driver. This project serves as a proof of concept and also a gauge on the limitations and capabilities of using a BeagleBone Black. Throughout the project I ran into several problems. Originally I wanted to use the BeagleBone Black that had the Robot Operating System (ROS) installed. The benefits from using ROS come from the many readily available packages. In particular I wanted to take advantage of gmapping which maps data from laser scanners while implementing SLAM (simultaneous localization and mapping). After working through the kinks of installing ROS correctly I was never able to properly map dummy data. Due to time constraints I had to switch courses and decided to implement the project with Javascript. I chose Javascript because there are many resources that are available for regular applications, but not specifically for the BeagleBone. Although it was quite the challenge to interface each device with another, I managed to provide a working scanner. Below is a summary of my journey.

Project Goals

- Use Javascript to control the BeagleBone Black and all of the components
- Provide a visual representation of the processed data

Components List

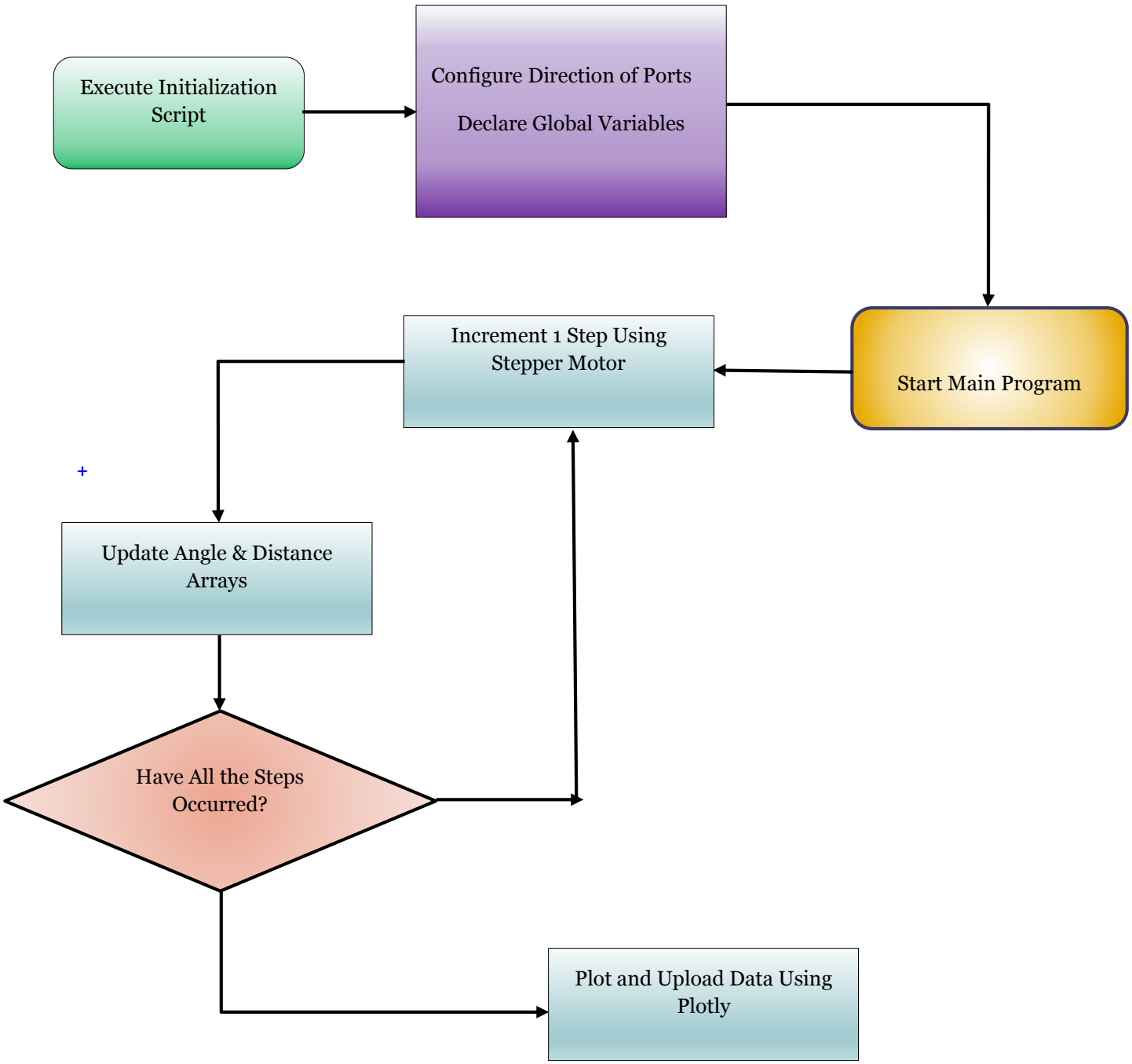
- **Beaglebone Black:** The BeagleBone Black is a low-cost microcontroller that attracts designers and hobbyist alike because of its many benefits. Since it is a popular unit on todays microcontroller market there is a large community fueled support group. It uses the AM335x 1GHz ARM Cortex-A8 processor and allows the user to install many different distributions of link. Also it supports a wide variety of programming languages which include, but are not limited to: C++, Python, and Javascript.

- **ULN2003A:** The ULN2003A is a high-voltage and high-current Darlington transistor array. There are seven NPN Darlington pairs which allow a high maximum switching point voltage. It also serves a clamp which is useful for driving inductive loads. In the case of driving stepper motors this is an ideal and effective solution.

- **LIDAR-Lite:** The LIDAR-Lite is a laser rangefinder that emits laser radiation. It has a maximum reading range of 40m and a resolution of 1cm. The sensor's light weight make it a good choice for projects that require drones. There are two ways of retrieving data from the LIDAR-Lite, pulse-width-modulation measurement and also I2C.

- **Vexta PX243 Stepper Motor:** The stepper motor allows for the LIDAR-Lite to be rotated with extreme accuracy. An advantage to using a stepper motor is that there is no accumulated error. This means if the a revolution is off by 1 degree it will always be off by that 1 degree. The input of the motor comes from the ULN2003A.

Flow Chart of Code



Breakdown of Code Used For Final Project

```
function stepMotor(thisStep){

    // each step is ~1.8 i use this to get 3 reads
    // per step
    var angleCounter = 0;
    var degInc = 0; // increments the total degrees read

    console.log('This is the counter: ' + globalCounter);
    globalCounter++;
    switch(thisStep){
        case 0: // 1010
            b.digitalWrite(controller[0], b.HIGH);
            b.digitalWrite(controller[1], b.LOW);
            b.digitalWrite(controller[2], b.HIGH);
            b.digitalWrite(controller[3], b.LOW);
        break;
        case 1: // 0110
            b.digitalWrite(controller[0], b.LOW);
            b.digitalWrite(controller[1], b.HIGH);
            b.digitalWrite(controller[2], b.HIGH);
            b.digitalWrite(controller[3], b.LOW);
        break;
        case 2: // 0101
            b.digitalWrite(controller[0], b.LOW);
            b.digitalWrite(controller[1], b.HIGH);
            b.digitalWrite(controller[2], b.LOW);
            b.digitalWrite(controller[3], b.HIGH);
        break;
        case 3: // 1001
            b.digitalWrite(controller[0], b.HIGH);
            b.digitalWrite(controller[1], b.LOW);
            b.digitalWrite(controller[2], b.LOW);
            b.digitalWrite(controller[3], b.HIGH);
        break;
    }
```

stepMotor only needs one argument, the current step. The possible values range from 0-3. The case statement is used to output the correct PWM sequence which is based on the current step. Another function named **step** will ensure the step order increases from 0-3 for counter clockwise rotations or decrease from 3-0 for clockwise rotations.

```
while(angleCounter < 3){

    var current_angle_time = microtime.now();

    if(current_angle_time - motorOb.last_step_angle >= motorOb.step_delay/4){

        motorOb.last_step_angle = current_angle_time;
        if(motorOb.degree > 360 && motorOb.direction ==1){
            motorOb.degree = 0;
            degInc = 0.597;
        }
        else if((motorOb.degree - 0.010) < 0 && motorOb.direction == 0){
            motorOb.degree = 360;
            degInc = -0.597;
        }

        motorOb.degree += degInc;
        console.log('This is the angle: ' + motorOb.degree);

        if(motorOb.direction == 1){
        angleStack.push(motorOb.degree);
        readDistance();
        }
    }
    angleCounter++;
}
```

This particular while loop in **stepMotor** allows me to read 3 distances from the LIDAR-Lite every step. I found each step incremented was approximately 1.8 degrees. At a minimum I wanted to be able to retrieve data within each degree of a rotation. By dividing the degrees per step by 3 I was able to retrieve the distance in increments of 0.6 degrees. This provides a resolution of 600 reads per revolution.

```
function setSpeed(desiredSpeed, motorOb){
    motorOb.step_delay = 60 * 1000 * 1000 / motorOb.number_of_steps / desiredSpeed;
}

// increases the actual step

function step(steps_to_move, motorOb){

    var microtime = require('microtime');
    var steps_left = Math.abs(steps_to_move);

    if(steps_to_move > 0)
        motorOb.direction = 1;
    else
        motorOb.direction = 0;

    while(steps_left > 0){

        var current_step_time = microtime.now();

        if(current_step_time - motorOb.last_step_time >= motorOb.step_delay){
            motorOb.last_step_time = current_step_time;
            if(motorOb.direction == 1){
                motorOb.step_number++;
                if(motorOb.step_number == motorOb.number_of_steps)
                    motorOb.step_number = 0;

            }
            else{
                if(motorOb.step_number == 0)
                    motorOb.step_number = motorOb.number_of_steps;
                motorOb.step_number--;
            }
            steps_left--;
            stepMotor(motorOb.step_number % 4);
        }
    }
}
```

step requires two arguments, the remaining steps to move in one revolution, and also the object that has been keeping track of the data in the current session (motorOb). The delay (which is calculated from the rpm in the function **setSpeed**) is compared with the time it takes to finish one step. Since the comparison is in microseconds this method is able to provide a smooth turning stepper motor.

```
function readDistance(){

    var b = require('bonescript');
    var micro = require('microtime');

    b.pinMode('P8_19', b.INPUT);
    b.pinMode('P8_17', b.OUTPUT);

    var counter = 0;
    var startTime = 0;
    var endTime = 0;
    var avStart = 0;
    var avEnd = 0;
    var averageArray = [];
    var average = 0;
    var timeout = 0;

    while(1){

        if(b.digitalRead('P8_19')){
            startTime = micro.now();

            while(b.digitalRead('P8_19')){}

            endTime = micro.now();

            if((endTime - startTime) > 40000)
                break;

            averageArray[counter] = endTime - startTime;
            counter++;
            if(counter == 5){

                for(var i = 0; i < 5; i++)
                    average += averageArray[i];
                average /= 5;

                distanceStack.push((endTime - startTime)/10);
                //console.log((endTime - startTime));
                break;
            }
        }
    }
}
```

readDistance requires no parameters. It measures the "on" time of a pulse wave in generated by the LIDAR-lite. In this particular case it reads the data five times and computes the average before saving the data into the distance array. The pulse wave is measure din microseconds. The distance is equivalent to 10us/1cm.

```
function plotData(distance, angle){
    var plotly = require('plotly')('clintonbess', 'lhk2li7zd2');


    var trace1 = {
      r: [],
      t: [],
      mode: 'lines',
      name: 'Figure8',
      marker: {
        color: 'none',
        line: {color: 'peru'}
      },
      type: 'scatter'
    };

    for(var i = 0; i < distance.length; i++){
        trace1.r[i] = distance[i];
        trace1.t[i] = angle[i];
    }

    var data = [trace1];
    var layout = {
      autosize: false,
      width: 500,
      height: 500,
      margin: {
        l: 0,
        r: 0,
        b: 0,
        t: 65
      }
    };
    var graphOptions = {layout: layout, filename: "plot1", fileopt: "overwrite"};
    plotly.plot(data, graphOptions, function (err, msg) {
        console.log(msg);
    });
}
```

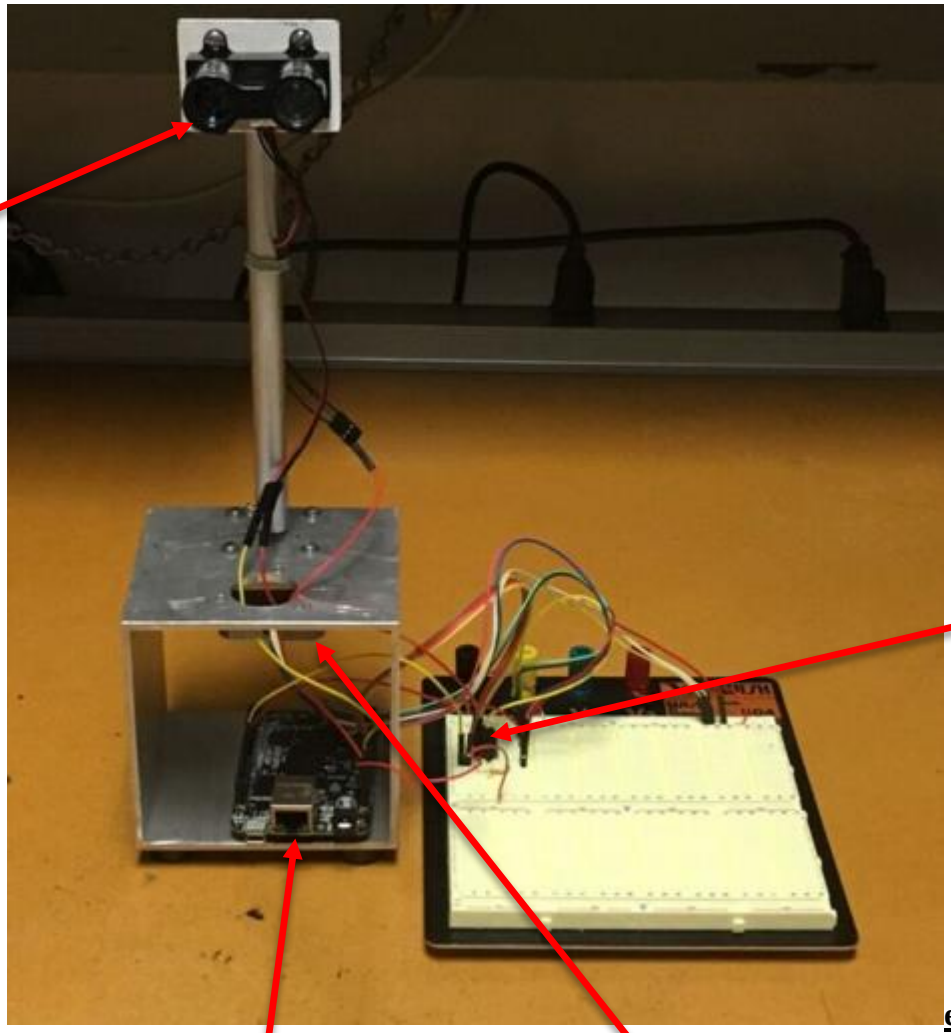
plotData requires two parameters, the distance array, and the angle array. Once executed it stores the two values into a trace object. Afterwards, the plotting function from the Ploty library plots the data and sends it to my Plotly account online.

```
var b = require('bonescript');
var microtime = require('microtime');
var util = require('util');

var controller = ["P9_11", "P9_13", "P9_15", "P9_17"];
var thisStep = 1;
var steps_per_rev = 200;
var desiredSpeed = 30;
var motorOb = {
    read_iteration: 0,
    direction: 0,
    speed: 0,
    step_delay: 0,
    number_of_steps: 200,
    step_number: 0,
    last_step_angle: 0,
    last_step_time: 0,
    degree: 0,
};

// sets the speed.. measured in rpm
setSpeed(30, motorOb);
// spins counter clockwise for 1 revolution
step(201, motorOb);
// spins clockwise for 1 revolution to untangle wires
step(-201, motorOb);


//console.log('this is the angle stack: ' + angleStack);
//console.log('this is the distance stack: ' + distanceStack);

//plots and uploads the data with plotly
plotData(distanceStack, angleStack);
```

The **main** part of the code is pretty straight forward. I first include the libraries that are needed. Then I declare the output pins that corresponding to the input of the motor driver. I then initialize an object that is used to control the current motor session and set the speed to 30 rpm. Afterwards, the step function is called to rotate the motor for one revolution counter clockwise, then again clockwise to untangle the wires. Finally, the data that was collected during the session is plotted and saved online.

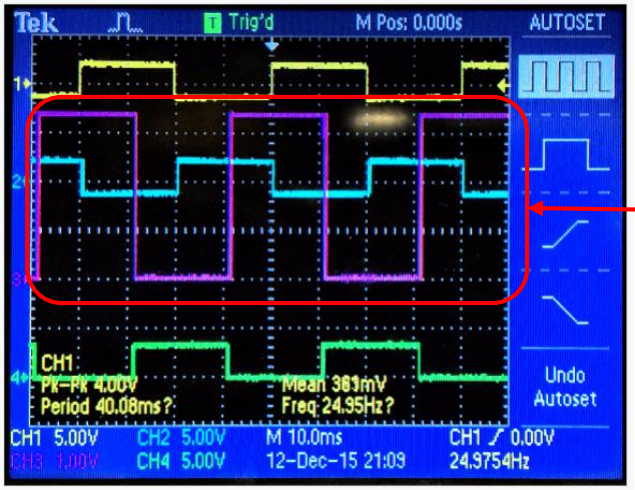
*** **The original code will be included in the Github project repository** ***

Picture of the Circuit

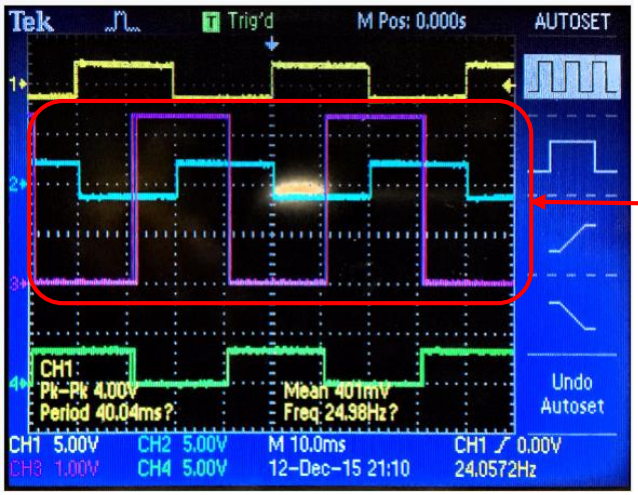


e

Image of Beaglebone Output



Clockwise Sequence

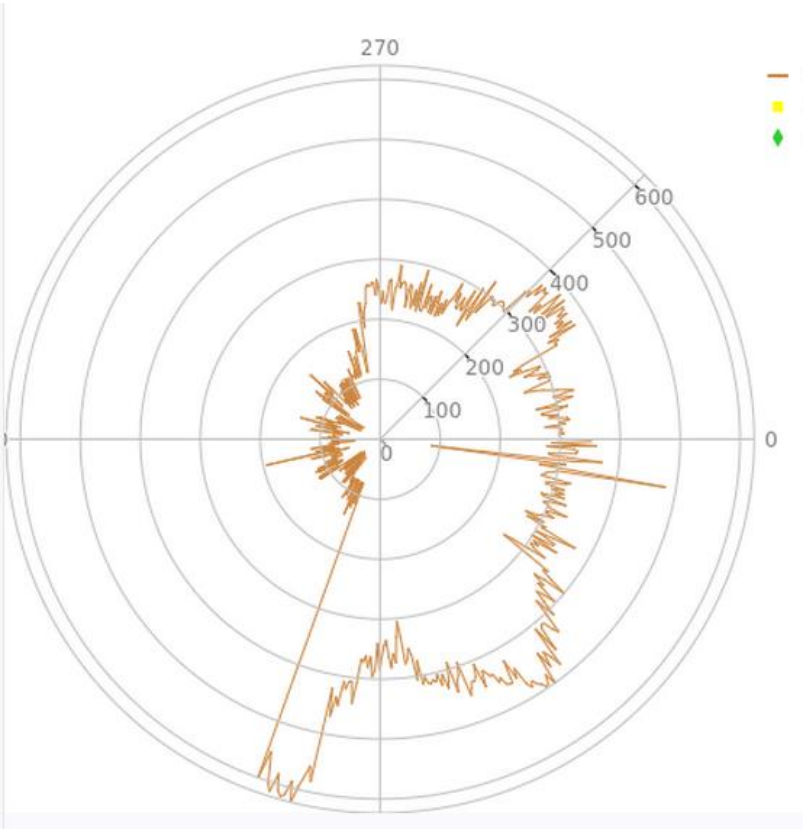


Counterclockwise Sequence

Video Link to Demo

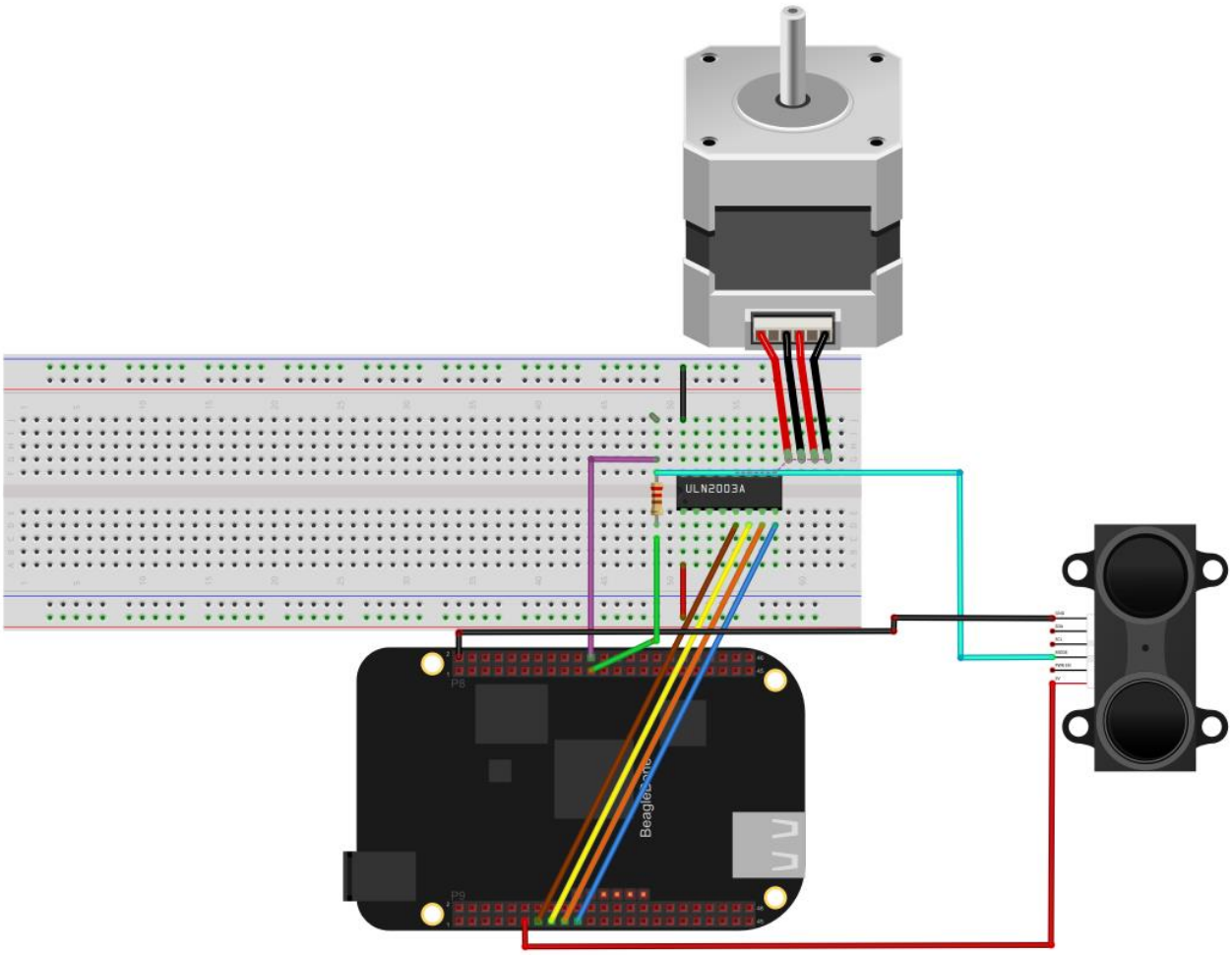
https://www.youtube.com/watch?v=6Nrp7cJtleM

2-D Map of Demo



- Above is an image of the results from the data collected with the LIDAR-Lite. Take note that the plot is from raw data, meaning there are no filtering algorithms present. The spikes in the plot come from the varying time it takes to calculate the pulse widths. Other than that the plot is accurate to the room's dimensions.

Schematic/ Wiring Diagram



Final Words

Overall this project was a lot of fun, but it also came with a quite a few headaches. As stated earlier this project served simply as a proof of concept. Although, I was able to accomplish this, I feel the implementation is quite incomplete. While working on this project I discovered many new features and limitations of the BeagleBone Black. Unfortunately, finding a work-around to a specific limitation aren't always quick process and when time is a factor, sometimes the creative and not-so efficient solution seems more appealing. A major shortcoming from my implementation came from my choice in programming language. Javascript is a great language for quick solutions as it is extremely flexible, requires no compile time, and is syntactically similar to C. A downside to using Javascript is that is not very memory efficient and also it processes tasks out of order. I have reason to believe the latter withheld me from properly utilizing the I2C option of data retrieval. If I was able to use I2C I would be able to gain a higher resolution in terms of gathering distance data which is consistent with my goal of obtaining distance data at high velocities. During the winter break I plan on making a few modifications to the project which include:

- I2C implementation of distance retrieval
- Comparing data with results from a servo motor and DC motor
- Construct a more complete library for interfacing the LIDAR-Lite
- Convert from Javascript to a more memory efficient language
- Implement delays using hardware as opposed to software