Current surveillance systems consist mainly of an array of cameras placed strategically throughout a facility. An increase in the coverage of the system requires a proportional increase in the number of sensors, and as a result, this approach is limited by the cost of manufacturing, installing, and maintaining multiple devices. The cost also increases proportionally with robustness, as it requires the use of additional sensors (FLIR, audio, etc.) with each device.

Current Market:

The most popular solutions for mobile, autonomous surveillance with support for real-time communication between devices include systems that are either too costly (Turtlebot, Knightscope), stationary (Nest), requires construction and maintenance (Turtlebot, Raspberry Pi-based projects), or does not include high-level detection and tracking of objects (Appbot). These details are summarized in the table below.

Name	Description	Price Range ($)
Appbot Link	Wi-fi controlled mobile camera with auto-charging battery	200
Nest Cam	1080p wide-angle stationary camera with zoom, face detection, and live streaming from mobile devices	200
Raspberry Pi-based systems	DIY kits and software widely available on the Internet	200
Turtlebot 2	Open-source mobile Microsoft Kinect-based system, reprogrammable using Robot Operating System, not pre-built	2,000
Knightscope Autonomous Data Machines	Human-size mobile system with GPS and an excess of sensors: LIDAR, sound, temperature, humidity, CO2, barometric.	4,500 per month

1. Solution

To achieve the same coverage and robustness as an array of surveillance devices, we propose an autonomous device that travels by land and performs high-level recognition of objects, signaling a detection to a network that can be accessed by an Internet-connected device, such as a mobile phone or laptop. Surveillance is accomplished by operating the device in one of the three modes: scan, patrol, and follow.

- Scan Mode:

  The system is stationary and pans the sensors in search of anomalies in the environment. After a period of time or at a time specified manually, the device enters Patrol Mode. When a match is found or a sound occurs in the environment that surpasses a specified threshold in volume, it enters Follow Mode and transmits a warning signal to the network.

- Follow Mode:

  The device adjusts its position and location to center the detected object in its viewing window and records optical footage. When a user deactivates this mode, it resumes the previous mode. High speed motion is enabled only in this mode.

- Patrol Mode:

  The device follows a path that is either specified or automatically determined by mapping the environment, changing path at regular intervals. When an anomaly is detected, it enters Follow Mode. After a specified period of time or at a specified time of day, it returns to Scan Mode. A proximity sensor and optical camera paired with object detection software allows the device to avoid collision. If a collision should occur, a distress signal is transmitted to the network.

2. Theory of Operation

Control System:

The system will be equipped with a variety of sensors to ensure accurate navigation and detection, some of which include ultrasonic and IR proximity sensors, vision modules for motion detection, and encoders to measure angular velocity and acceleration. The sensors will relay information about the system's speed, acceleration, and position to the controller, the BeagleBone Black (BBB) embedded system, which in turn controls the motion of the system components through PWM signals sent to the DC motors, for locomotion, and the servo motor, for surveillance. Lastly, the BBB will be programmed with the Linux flavor Debian and will

incorporate the Robot Operating System (ROS) for Simultaneous Localization and Mapping (SLAM).

Vision System:

SLAM allows a mobile system to estimate the relative locations of itself and the time-invariant objects in its environment by making observations and issuing control signals at equally spaced intervals of time. At each time step, the system updates its location and the map of environmental objects by using one of many algorithms that have been implemented by open-source developers in C++, MATLAB, Java, and C. SLAM allows the system to navigate an unfamiliar environment in real-time when it enters Patrol or Follow Mode.

OpenCV accomplishes high-level recognition of objects using a number of established methods implemented in C++ and Python. During Scan and Patrol Mode, when an object of interest is recognized across multiple frames, the system enters Follow Mode and issues a warning signal. The system then attempts to move in such a way that the object of interest remains centered in the field of view. OpenCV is thus useful for performing this real-time recognition in any of the modes of operation.

Communication System:

Events of interest include the detection of an anomaly during Scan/Patrol Mode or a state of distress during which the system is unable to function as expected. When these events occur, the BBB sends an email to an SMTP mail server, namely Gmail, to notify a user of the occurrence. The If This Then That (IFTTT) web application allows these email notifications to serve as triggers for complex reactions, such as automatic Tweets or the changing of settings across Nest, Aros, SmartThings, WeMo, and other devices used for automation.

3. Materials

Hardware:

- BeagleBone Black (Revision C)
- OSEPP Ultrasonic Sensors
- Dagu Rover 5 Tracked Chassis
- Pololu Rover 5 Expansion Plate
- DRV8833 Dual Motor Driver Carrier
- Pololu Carrier with Sharp Analog Distance Sensor
- Logitech C920 HD Webcam
- Asus USB-N10 Network Adapter

Software:

- Debian OS
- Robot Operating System (ROS)
- OpenCV
- SLAM
- Node.js
- If This This Then That (IFTTT)

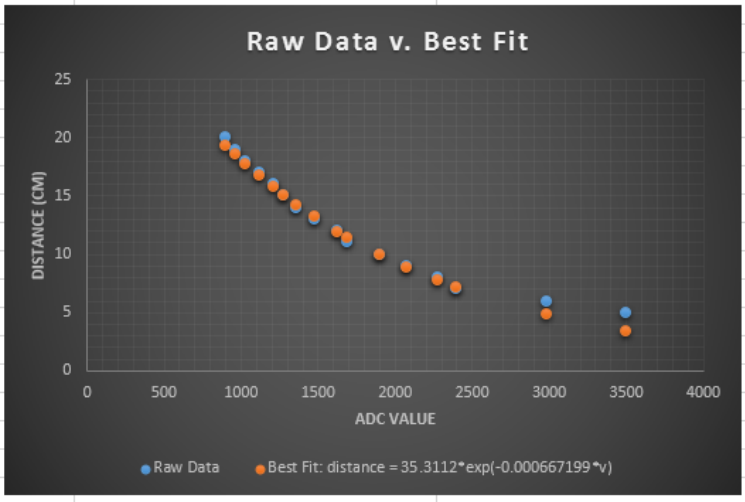
4. Resources

- Dr. Pushkin Kachroo
- Derek Molloy, *Exploring the Beaglebone Black*
- Ferat Sahin and Pushkin Kachroo, *Practical and Experimental Robotics, 1st ed.*
- Pushkin Kachroo and Patricia Mellodge, *Mobile Robotic Car Design, 1st ed.*

Progress

- **Automatic initialization of ADC pins:** The use of Bash shell commands to activate the ADC pins was automated by including the commands in the shell initialization file in Debian OS. This automation allows us to connect to and program the BeagleBone instantly on startup, improving our efficiency in testing and debugging software.
- **Distance sensor characterization:** Using the multimeter to measure the voltage outputs of the distance sensor and the quantized outputs of the ADC, which range from 0 to 4095, we obtained a spreadsheet of data that we could input to the Wolfram Alpha Least-Squares Fit function to generate an exponential model of the distance. We chose to take measurements at distances of 5 to 20 cm from the sensor (the functional range of the sensor) at intervals of 1 cm. The data, fitting values, and mathematical model are shown below.

40	Raw Data				
41	ADC	Distance (cm)			Best Fit: distance = 35.3112*exp(-0.000667199*v)
42	900	20			19.36991509
43	965	19			18.54783664
44	1028	18			17.7843636
45	1117	17			16.75905838
46	1210	16			15.75077238
47	1275	15			15.08229394
48	1360	14			14.25075024
49	1475	13			13.198216
50	1625	12			11.94128614
51	1685	11			11.47269517
52	1900	10			9.93955344
53	2075	9			8.844203468
54	2280	8			7.713611334
55	2400	7			7.120105866
56	2980	6			4.835319339
57	3500	5			3.417822642
58					
59					



- **Programming the BeagleBone:** C++ was used to program the BeagleBone to map the ADC outputs to distances that are calculated by the mathematical model that we obtained in the previous Progress Report by characterizing the sensors experimentally. We wrote a separate program to test the limits of the motor speeds. Once the software for the sensors and motors were operating as expected, we wrote the program (shown below) to integrate the two functions based on a control algorithm that we designed. Here, the error is defined in terms of the distances from the sensors and the PWM voltage given to each motor is adjusted in order to optimize this error, i.e., cause the distance from sensor to object to increase to above 25. See our mathematical model for more details on the relationship between input, output, and state variables of our system. The system behaved as expected, running forward at full speed when no objects obstructed it, turning gently away when an object approached from a distance, turning sharply away from an object when the object was very near, and reversing direction when entirely obstructed from the front.

```
#include <iostream>
#include<fstream>
#include<string>
#include<sstream>
#include<cmath>

#include "motor/DCMotor.h"

using namespace std;
using namespace exploringBB;

#define LDR_PATH "/sys/bus/iio/devices/iio:device0/in_voltage"

int readAnalog(int number)
{
    stringstream ss;
    ss << LDR_PATH << number << "_raw";
    fstream fs;
    fs.open(ss.str().c_str(),fstream::in);
    fs >> number;
    fs.close();

    return number;
}

int main(){
    DCMotor dcm1(new PWM("pwm_test_P9_42.16"), 116); //will export GPIO116
    DCMotor dcm2(new PWM("pwm_test_P9_22.17"), 115); //will export GPIO115
    dcm1.setDirection(DCMotor::CLOCKWISE);
    dcm1.setSpeedPercent(65.0f);   //make it clear that a float is passed
    dcm2.setDirection(DCMotor::CLOCKWISE);
    dcm2.setSpeedPercent(50.0f);   //make it clear that a float is passed
    dcm1.go();
    dcm2.go();

    int counter = 0;
    float distanceThreshold = 25.0;
    float maxPID = 20; // max possible value of the sum of control variables
    float kp = 1;
    float kd = 1; // the gains of proportional and derivative control

    float error1, error2;
    float lastError1 = 0;
    float lastError2 = 0;
    float P1, D1, P2, D2; // proportional and derivative control variables
    float motorSpeed1, motorSpeed2; // the corrected motor speed as a percentage of max
```

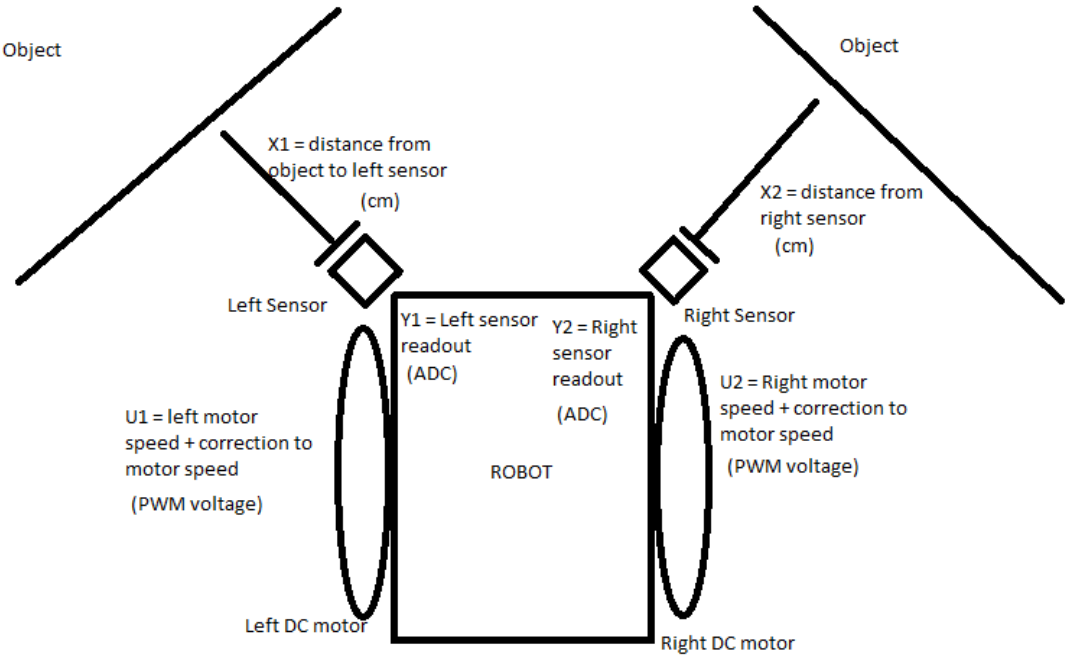
```
motor speed

    while(1) {
        // int value0 = readAnalog(0);  // assign value from ADC 0 pin
        int value1 = readAnalog(1);  // assign value from ADC 1 pin
        int value2 = readAnalog(2);  // assign value from ADC 2 pin
        // float distance0 = 35.3112f*exp(-0.0006672f*(float)value0);  // calculate
distance from sensor 0
        float distance1 = 35.3112f*exp(-0.0006672f*(float)value1);  // calculate distance
from sensor 1
        float distance2 = 35.3112f*exp(-0.0006672f*(float)value2);  // calculate distance
from sensor 2

        /*
        cout << "The distance from the front sensor is: " << distance0 << " cm" << " i =
" << i << "value = " << value0;  // display distance0

        // Back up if sensor 0 detects an object

        if (counter > 0)
            counter -= 1;
        else if (distance0 <= 10) {
            counter = 10;
            cout << "\n Shaka Zulu" << '\r' << endl;  // display warning if neccessry
            dcm1.setDirection(DCMotor::ANTICLOCKWISE);
            dcm2.setDirection(DCMotor::ANTICLOCKWISE);
            dcm1.setSpeedPercent(65.0f);   //make it clear that a float is passed
            dcm2.setSpeedPercent(50.0f);   //make it clear that a float is passed
            dcm1.go();
            dcm2.go();
            usleep(5000000);
            dcm1.stop();
            dcm2.stop();
        }
        else
            cout << "\n acceptable \n" << '\r' << endl;

        cout << "The distance from the first sensor is: " << distance1 << " cm";  //
display distance1
        */

        if ((distance1 <= distanceThreshold) && (distance2 >= distanceThreshold)) {
            dcm1.setSpeedPercent(50.0f);   //make it clear that a float is passed

            cout << "Correcting the distance from the left sensor" << '\r' << endl;
            cout << "Distance from sensor is: " << distance1 << '\r' << endl;

            error1 = distance1 - distanceThreshold;
            P1 = kp*error1;
            D1 = kd*(error1 - lastError1);
            cout << "Error 1: " << error1 << '\r' << endl;
            lastError1 = error1;

            motorSpeed2 = 100.0 - 50.0*(maxPID + P1 + D1)/maxPID;

            cout << "P1" << P1 << '\r' << endl;
            cout << "D1" << D1 << '\r' << endl;
            cout << "Motor Speed 2: " << motorSpeed2 << '\r' << endl;

            dcm2.setSpeedPercent(motorSpeed2);

            int value1 = readAnalog(1);  // assign value from ADC 1 pin
            float distance1 = 35.3112f*exp(-0.0006672f*(float)value1);  // calculate
distance from sensor 1

        }
        else if ((distance2 <= distanceThreshold) && (distance1 >= distanceThreshold)) {
            dcm2.setSpeedPercent(50.0f);   //make it clear that a float is passed

            cout << "Correcting the distance from the right sensor" << '\r' << endl;
            cout << "Distance from sensor is: " << distance2 << '\r' << endl;
```

```
            error2 = distance2 - distanceThreshold;
            P2 = kp*error2;
            D2 = kd*(error2 - lastError2);
            lastError2 = error2;

            motorSpeed1 = 100.0 - 50.0*(maxPID + P2 + D2)/maxPID;

            cout << "P2" << P2 << '\r' << endl;
            cout << "D2" << D2 << '\r' << endl;
            cout << "Error 2: " << error2 << '\r' << endl;
            cout << "Motor Speed 1: " << motorSpeed1 << '\r' << endl;

            dcm1.setSpeedPercent(motorSpeed1);

            int value2 = readAnalog(2);  // assign value from ADC 1 pin
            float distance2 = 35.3112f*exp(-0.0006672f*(float)value2);  // calculate
distance from sensor 1

            counter = 0;
        }
        else if ((distance1 >= distanceThreshold) && (distance2 >= distanceThreshold)) {
            dcm1.setSpeedPercent(50.0f);   //make it clear that a float is passed
            dcm2.setSpeedPercent(50.0f);   //make it clear that a float is passed
        }
        else if ((distance1 <= distanceThreshold) && (distance2 <= distanceThreshold)) {
            dcm1.setSpeedPercent(100.0f);   //make it clear that a float is passed
            dcm2.setSpeedPercent(100.0f);   //make it clear that a float is passed
        }

        usleep(50000);
    }
    return 0;
}
```

- **Compact Power Supply:** To test the mobile capabilities of the system, a compact power supply is needed that is small and light enough to be carried by the system. We achieved this by installing a power supply that uses a 9V battery, replacing the stationary power supply that we initially used during the first testing stage.
- **Motor Drivers:** We tested the circuitry for the motor drivers on a breadboard, and once their operation was satisfactorily demonstrated, we soldered the leads to the perfboard and attached them to the system.
- **Encoder circuitry:** An external circuit will be used to measure the angular frequency of each wheel of the system and subsequently encode each as a digital signal in order to determine the speed of each wheel and prevent any imbalance in speed across wheels in real-time to prevent the system from straying from its path.
- **OSEPP Ultrasonic Sensors:** As the name suggests, an ultrasonic sensor generates and emits an ultrasonic sound at a frequency, undetectable to humans, that is used for detecting nearby obstacles. Given the frequency of the signal emitted and the time delay of the signal when it has returned, the sensor can communicate the distance of nearby objects to the Beaglebone.

- **State-Space (Mathematical) Model:**

The state of the system is represented by two variables, X1 and X2, which represent the distance of an object from the left and right sensors of the robot, respectively, as shown in the figure below:



The control signal of the system is represented by two variables, U1 and U2, which represent the pulse-width modulated (PWM) voltages that are delivered to each pair of motors on the left and right side of the robot, respectively.

The system dynamics are described by the following relationships, where y represents the readouts of the left and right sensors. (Note: in this report, vectors and matrices are indicated by boldface text.)

$$\mathbf{x}[n+1] = \mathbf{A}\mathbf{x}[n] + \mathbf{B}\mathbf{u}[n]$$

$$\mathbf{y}[n] = \mathbf{C}\mathbf{x}[n]$$

C is a matrix that converts the values of **x**, which have units of centimeters, into a quantity with units of time in nanoseconds, namely the pulse widths calculated by the BeagleBone Black (BBB). The conversion factor between the sensor's output distance value and the ADC value is the same for both sensors, and so, the entries of the C matrix are identical, resulting in the following scalar output equations:

$$\mathbf{y}[n] = \begin{bmatrix} y_1[n] \\ y_2[n] \end{bmatrix} \qquad y_1[n] = cx_1[n] \qquad y_2[n] = cx_2[n]$$

The value of c can be determined by using the relationship between distances and ADC values:

$$ADC = 4095 - \frac{d}{0.00977}$$

Defining the following quantity allows us to derive a value for c that will convert distances into untiless values:

$$ADC_{mod} = ADC - 4095$$

$$c = \frac{ADC_{mod}}{d} = -\frac{1}{0.00977} = -102.46\ cm^{-1}$$

B is a matrix that converts the control signal sent to each motor (with units of voltage) into a quantity with units of distance in centimeters, in order to match the units of **x**. Both **A** and **B** will be determined experimentally by observing how the system responds to various configurations of **u**.

Shown below are the scalar equations that describe how each entry of **x**[n+1] (the change in distance of an object from a sensor) depends only on the current distance measured by that sensor and the voltage sent to the motor on the side of the robot that corresponds to that sensor.

$$\mathbf{x}[n] = \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix}$$

$$x_1[n+1] = a_1 x_1[n] + b_1 u_2[n]$$

$$x_2[n+1] = a_2 x_2[n] + b_2 u_1[n]$$

For the purpose of testing and simulation, the values of a_1 and a_2 are assumed to be equal and to yield values on the scale of tens (the scale of the distances in cm that are of interest to our system) when multiplied by x_1 and x_2.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The values of b_1 and b_2 are then chosen to scale the values of u_1 and u_2 , respectively, to the order of tens. Because u_1 and u_2 represent the quantized PWM voltages that are sent to the DC motors, we expect their values to range from 0 to 255. Dividing the maximum expected value of x_1 (35 cm) by that of u_1 (255), the value of b_1 is obtained. Because the motor on the right-hand side of the system has been tested and shown to require a slightly higher voltage to match the speed of the left-hand side motor, the value of b_2 is obtained by multiplying b_1 by a factor of 55/50 (the approximate ratio of voltages needed for the motors to run at the same speed).

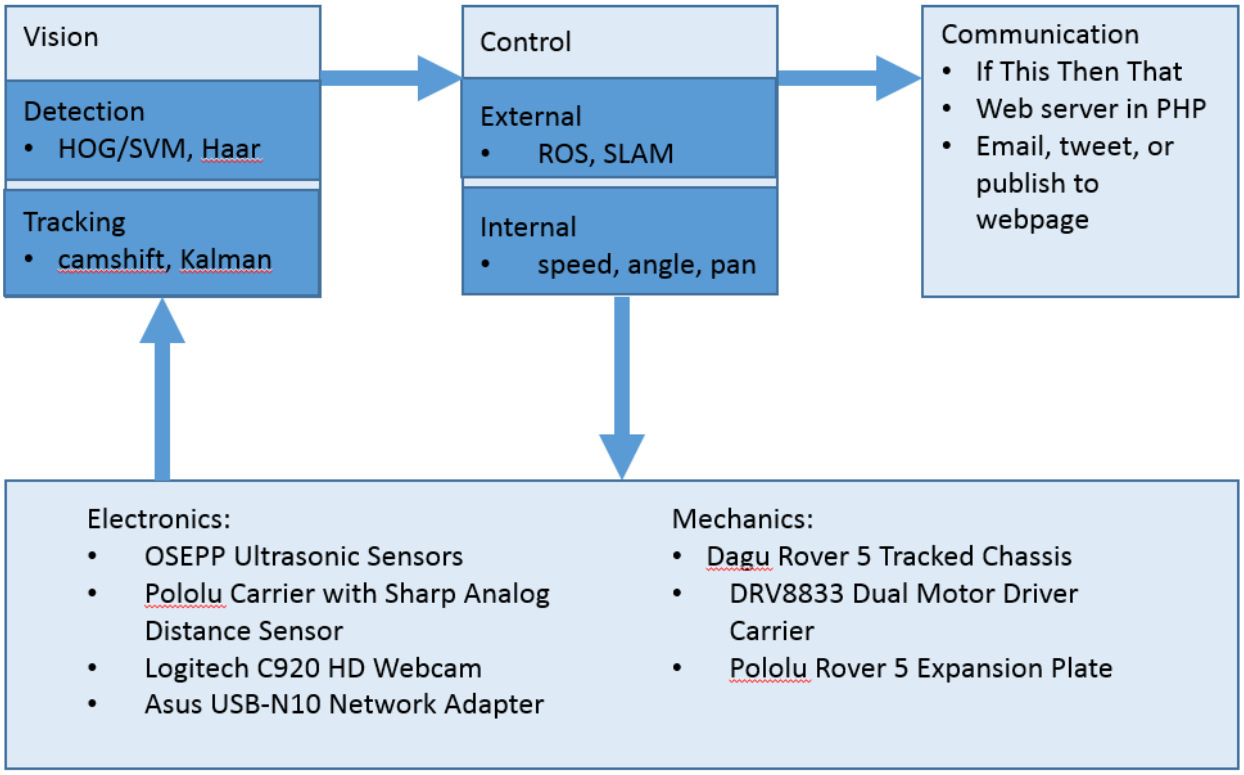
$$\mathbf{B} = \begin{bmatrix} 0.151 & 0 \\ 0 & 0.137 \end{bmatrix}$$

The relationships determined above can be summarized by the following matrices:

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_1 & 0 \\ 0 & b_2 \end{bmatrix} \qquad \mathbf{C} = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix}$$

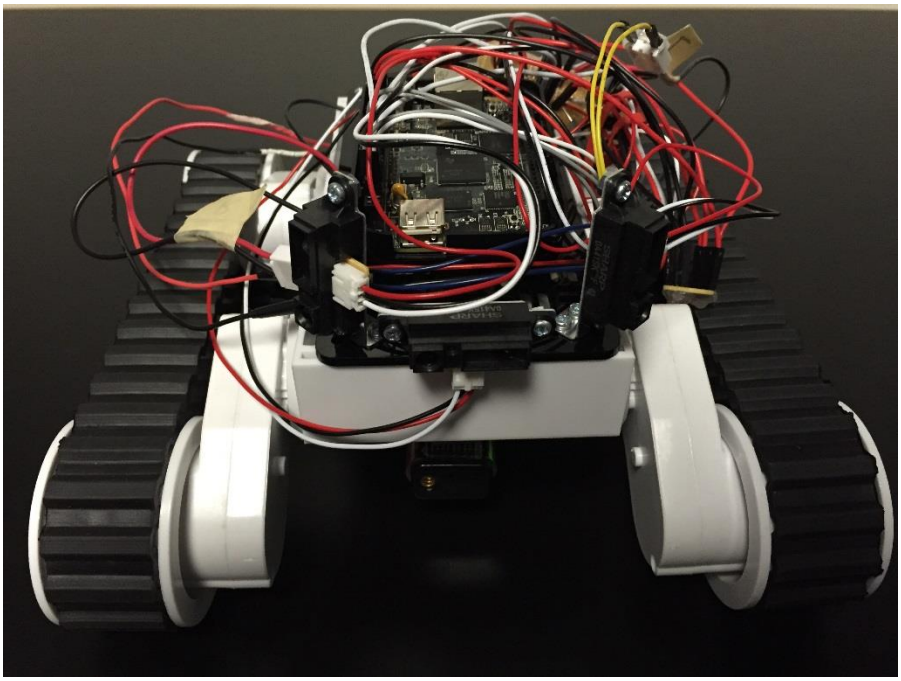
$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0.151 & 0 \\ 0 & 0.137 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} -102.46 & 0 \\ 0 & -102.46 \end{bmatrix}$$

Current state of the project

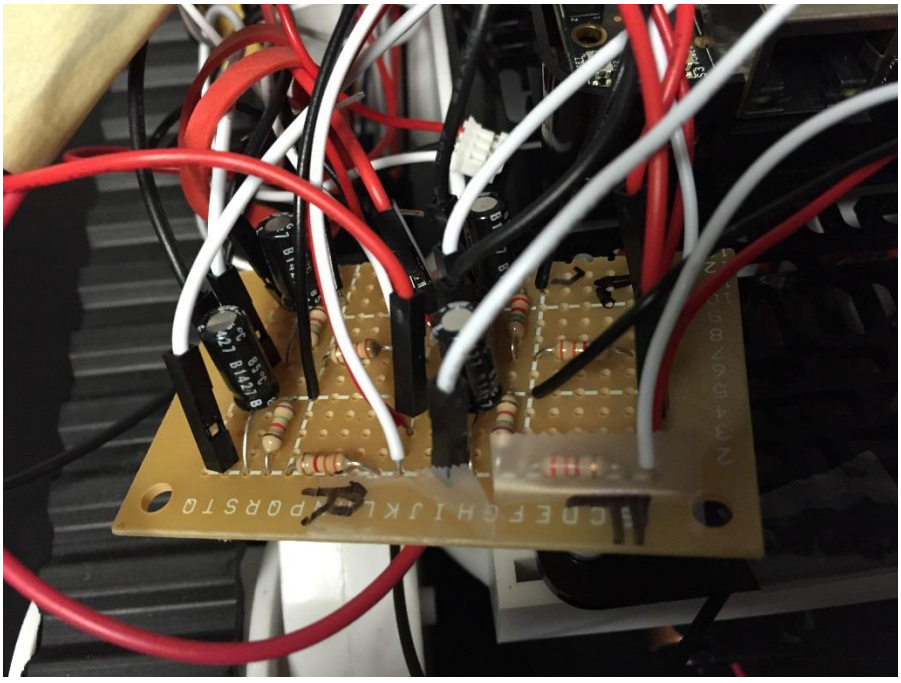


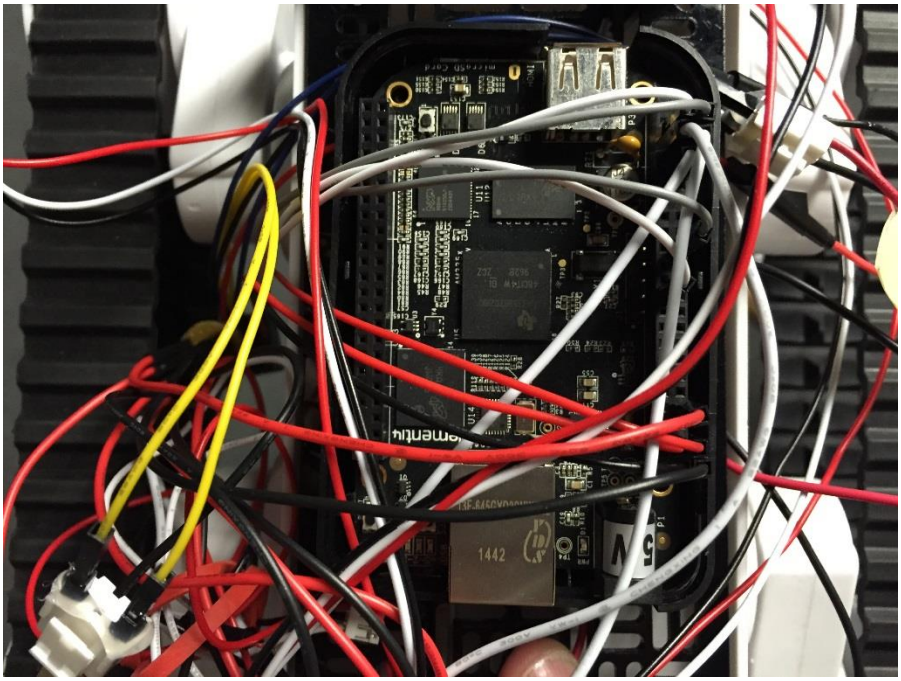
The block diagram outlines the relationship between the various functions to be performed by our system. The arrows lead from a module to the one on which it depends for data. The hardware underlies all functions and includes the electronics and mechanical components that enable each function. The communication module interfaces with a user, which does not appear in the diagram, as the system is expected to be primarily autonomous. The vision module is divided into two functions, each of which entail a specific set of techniques available in OpenCV. Similarly, the control module consists of both software that is available in repositories of open-source software (external) and that the BeagleBone itself provides (internal).

The current progress made on this project falls under the category of Electronics, Mechanics, and Control Functions. This progress is outlined by the images below.

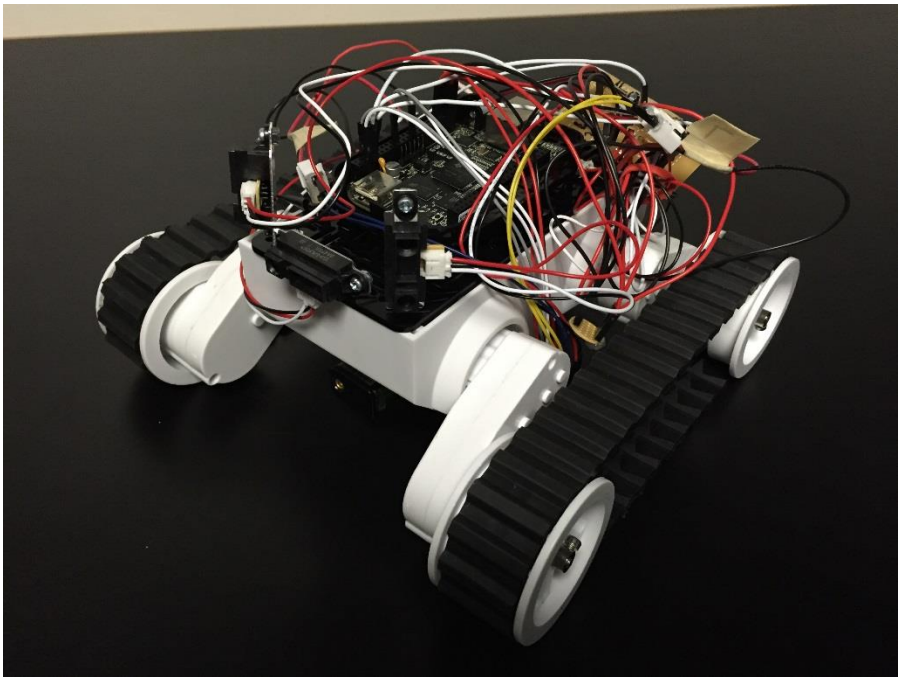


Above: Three IR distance sensors mounted to the front of the chassis.
Below: Pin protector circuitry, wired to the BeagleBone and soldered to the perfboard.





Above: The BeagleBone mini-PC mounted on top of the chassis.
Below: The total system, including chassis, wheels, treads, circuits, sensors, and BeagleBone.



Current roadmap of the project

We plan the progress of our project as the milestones listed below:

- **Test the limitations of continuous-time object avoidance (Jan. 19)**
  Now that the control system has been shown to function as expected, we plan to test the system at high velocities and across multiple settings to determine the practical limitations of the physical system and any weaknesses in the theoretical model that we developed regarding the control system.
- **Implement the tracking system (Mar. 1)**
  First, we will select and test various people detection techniques available in OpenCV and how the characteristics of the Logitech Webcam will limit the robustness of each technique. Then, we will integrate these functions in C++ and Python and demonstrate mobile, real-time tracking of people with our system by extending the control software that we developed in the previous semester.
- **Implement the communication system (Apr. 5)**
  We will extend the control software further to include methods for transmitting or publishing the detection results to other web-connected devices and receiving simple commands remotely from those devices as a way of making decisions regarding the safety and effectiveness of the system in real time.
- **Final report ready (Last week of April)**
  The final report will include experimental results on how the configuration of sensors affects the ability of the software to output usable information to the control system, how well the system performs under a variety of operating conditions, and how to optimize its performance by adjusting the hardware, software, or theoretical model.

Current problems

Currently, we are debugging the obstacle avoidance program that we have compiled on the BeagleBone. However, after multiple sessions of testing and debugging, we have encountered our most recent error which allows our code to execute but nullifies any effect it would have in the system's behavior. When we interface the BeagleBone with MobaXTerm, the error displayed vaguely refers to a string location being underwritten. We hypothesize that the error refers to our attempt to use both the left and right DC motors at the same time (or at least, in the same program). After some research, we have found that this is a bug in the C++ compiler we are using, G++. Thus to fix this problem we have two possible solutions; either we can rewrite the code to accommodate this bug or we can use a different C++ compiler. Currently, we have decided to use another C++ compiler, Clang++, and thus our next step will be installing this compiler on the BeagleBone.