Bravobot

The art of being classy and a robot.

Xiaozheng Xu

12/10/16

Fundamentals of Robotics

Contents

List of figures

Executive Summary

Bravobot is created by Jules Risbec, Isaac Vandor, Raagini Rameshwar, Sunny Suroff, Trent Dye, William Lu, Xiaozheng Judy Xu, and Lydia Zuehsow, all students of Olin College. It is a classy ground robot designed to race around the Olin main O. Wearing a tuxedo and having flashy eyelashes, Bravobot knows how to go fast but also how to do it with style.

Bravobot is created from scratch. Given some wood, motors, controllers and a sensor suit, our team built the entire mechanical system, electrical system and software system. Each member of our team has their own expertise that came in handy in our building process. Although I helped out in building the mechanical and electrical system, I personally contributed the most to the software system.

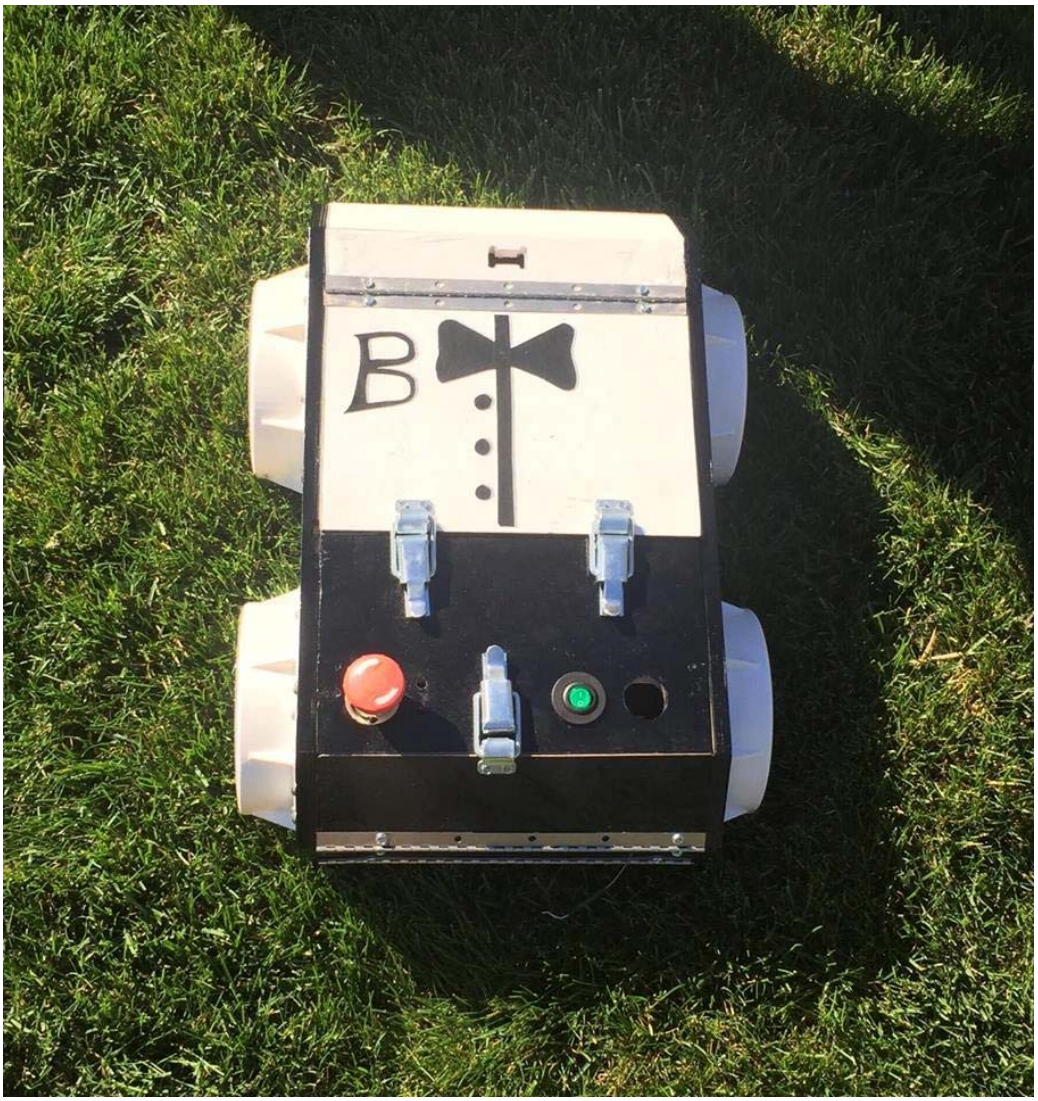

Figure 1- Bravobot on grass and in sunlight

Overall Design Description

Bravobot was designed to be modular, weatherproof, and easily accessible. With these design principles in mind, we designed a flexible and fully sealable mechanical system, a sliding tray based electrical system, and a full software suite designed to take full advantage of onboard sensors and actuators.

Bravobot was designed with the intention of being as weatherproof as possible while maintaining easy access and the ability to debug the electrical system easily. With this overarching design principle in mind, we designed a fully sealable, but flexible mechanical system, an electrical system easily removed via a sliding tray, and a software system designed to take full advantage of the sensing and actuation suites onboard.

Mechanical System:

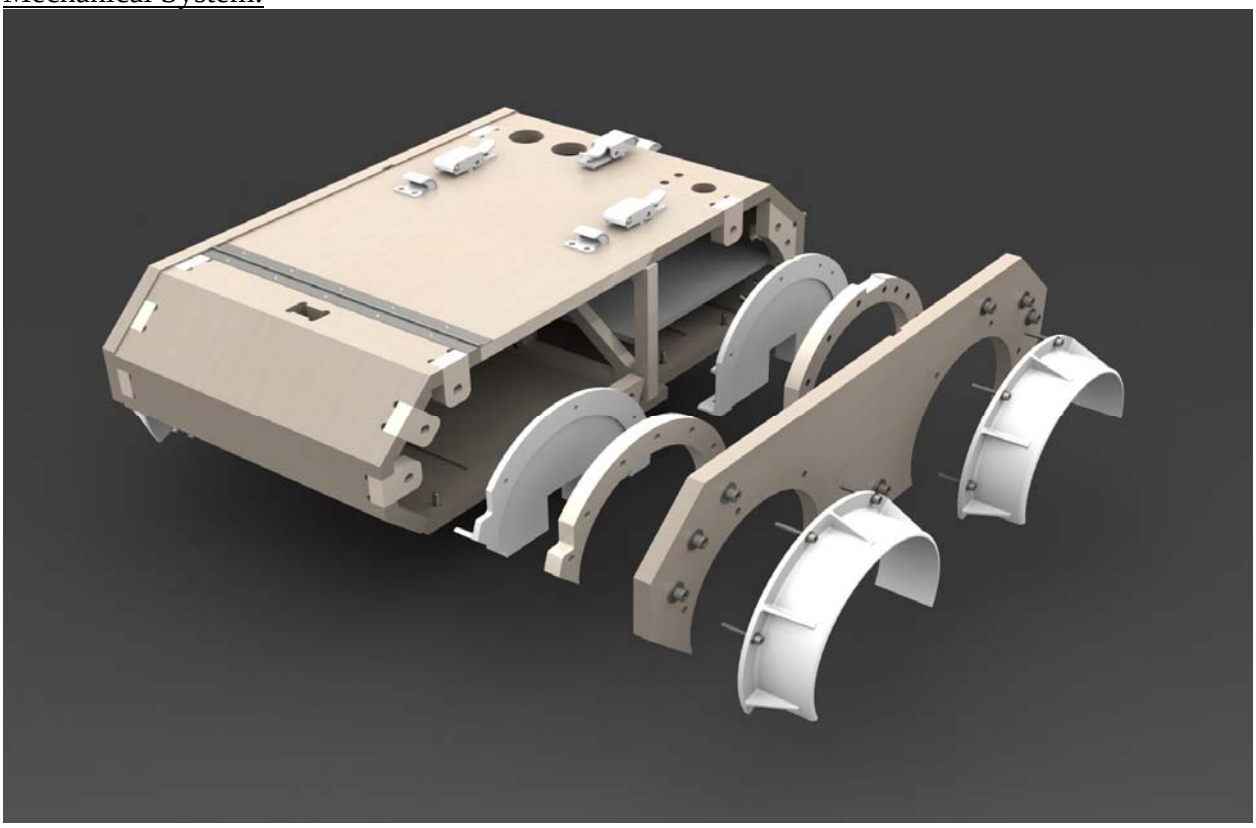

FIGURE 2- CAD RENDER OF CHASSIS

In order to build a robust, rugged mechanical system, Bravobot is split into two compartments separated by flexible wooden joints.

FIGURE 3 - WOODEN JOINTS ASSEMBLED

These joints are held closed by hinges enabling quick access to one or the other or both compartments.

FIGURE 4- HINGE OPENING

In this manner, debugging any one component is quick and easy without having to deal with any of the system components in the other compartment. Furthermore, the flexible design of the wooden hinges enabled us to fold the back panel straight down and slide the electronics tray horizontally out of the back of the robot. Utilizing the hinge design also allows the robot to be almost fully sealed when running, with the only openings in the shell being for sensors, basic controls like the on/off switch, and safety system components like the Emergency stop or behavior LED lights.

FIGURE 5 - FINAL PAINTED MECHANICAL DESIGN

Electrical System:

In order to develop a modular and easily debuggable electrical system, we started making design decisions as we planned the vehicle shell. Our primary focus when thinking about the electrical system was to ensure that it would be convenient to debug so that we could identify and fix problems early. To do this, we incrementally developed the electrical system, starting with LEDs to provide visual feedback on robot state. The LEDs let us know when the robot is powered on and when the computing suite is receiving 5V power. Eventually, the LEDs also told us the functional state of the computing suite, such as when the robot was physically e-stopped, software stopped, or running normally.

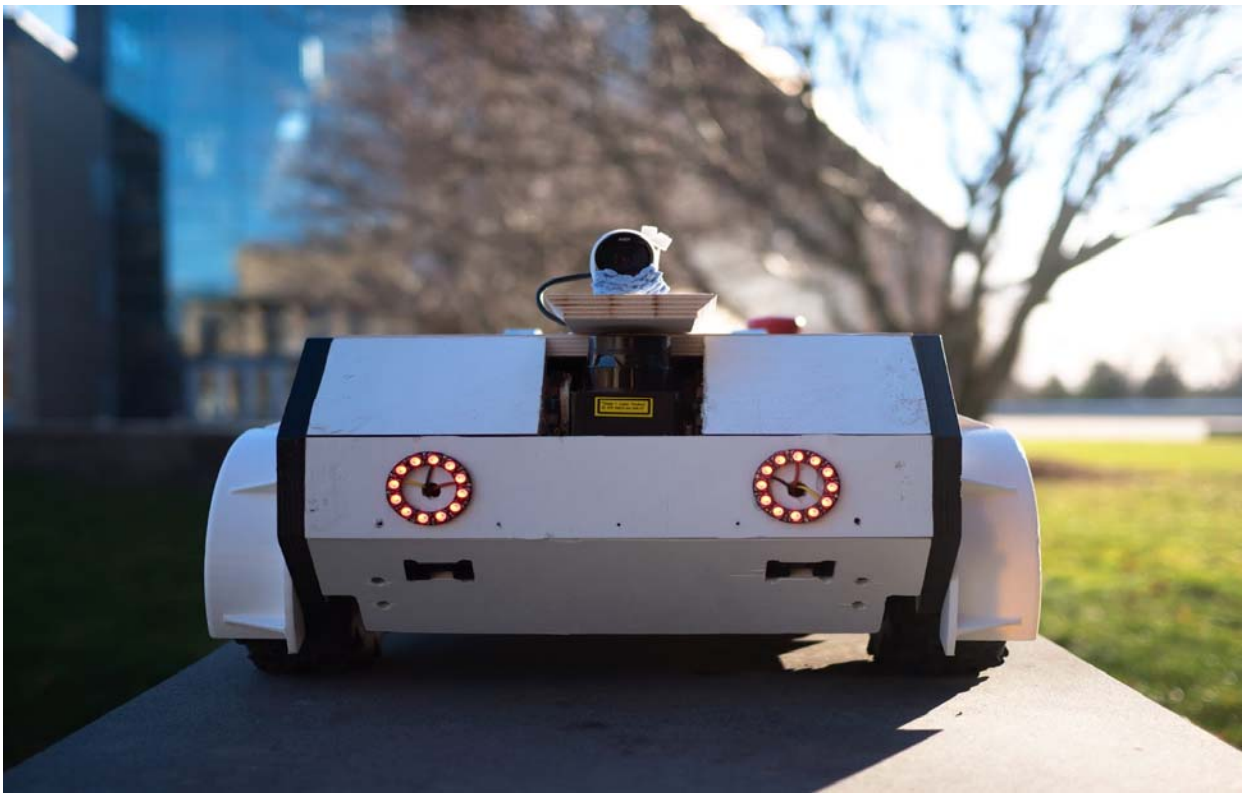

FIGURE 6 - FRONT OF ROBOT WITH SENSORS AND LIGHTS

With a method of visual feedback developed, we proceeded to wire system components together. In order to safely switch power to our robot on and off, our battery is wired to a switch via a single inline 25 amp fuse. The switch is in turn wired to the main fuse block. Since our fuse block lacks a mother ground, we ran ground from the battery to a terminal block and jumped ground to each terminal slot in order to provide a common ground for all components.

Given our focus on safety, an emergency stop button was wired before we added any more components to our electrical system. We wired one pole of the emergency stop button directly to the "+" terminal strip on our Roboclaw motor controller and another pole of the emergency stop button to a positive terminal on the fuse block. This allows us bypass any software controlled systems and cut power to motors on the robot.

To increase our awareness of the electrical state of the system, we wired indicator lights to our emergency stop button and to the main +12VDC power line such that the indicator lights would turn on as soon as the robot was powered on. The indicator light wired to the e-stop turns off when the e-stop is activated.

In order to switch power to our robot on and off safely, our battery is wired to a switch via a single inline 25 Amp fuse. The switch is in turn wired to the main fuse block. Since the fuse block we are using lacks a mother ground, we ran ground from the battery through the terminal and then jumped ground to each terminal slot in order to provide a common ground for all components.

In order to ensure that our robot is functioning properly, we wired indicator lights to emergency stop and to the +12VDC power line such that the indicator lights will turn on as soon as the switch is flipped and the robot is powered, but remain turned on when the e-stop is pressed.

Given our focus on safety, an emergency stop button was the very next component wired after ensuring that the robot was powered on. In order to wire the emergency stop such that it would immediately cut power to the motors while bypassing any software-controlled systems, we wired one pole of the emergency stop directly to the "+" terminal strip on the roboclaw motor controller and another pole of the emergency stop to a positive terminal on the fuse block. By wiring the emergency stop this way, we are bypassing the sensing and computing suite, enabling the emergency stop to cut power to the motors directly.

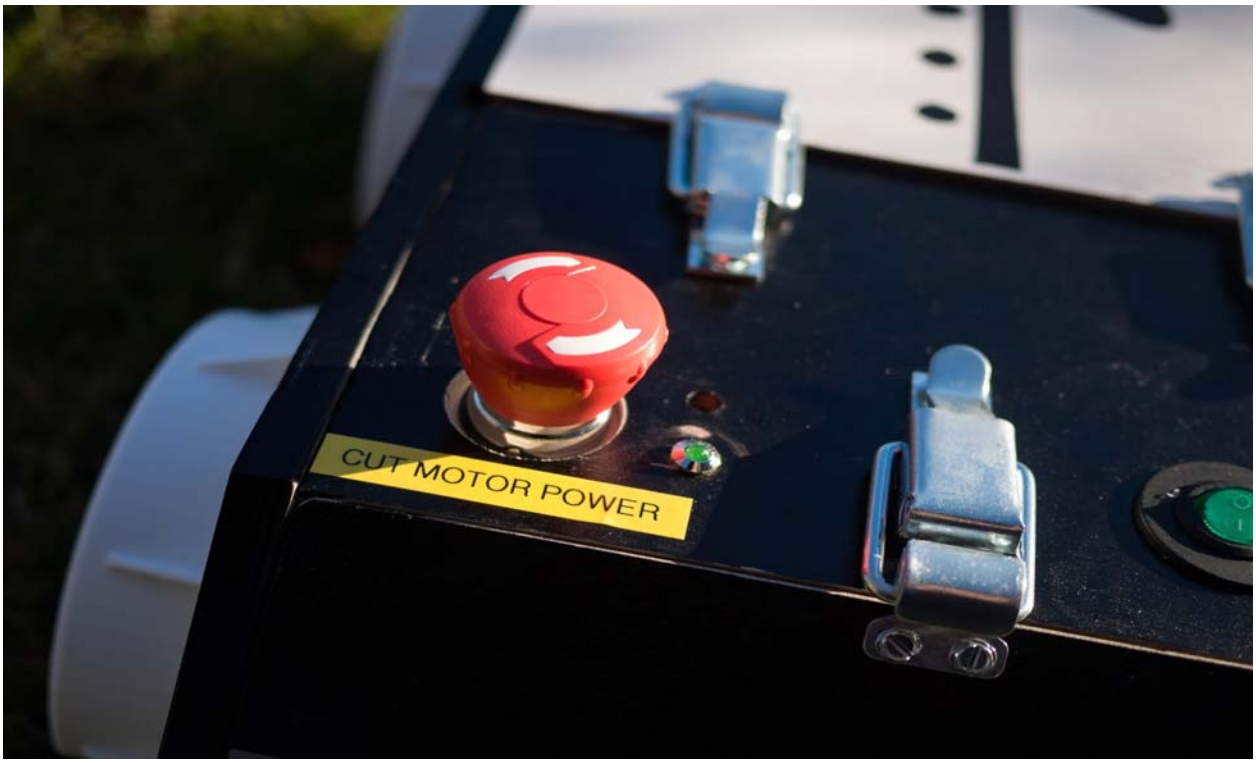

FIGURE 7 - EMERGENCY E-STOP ON THE BACK

With an emergency stop and indicator lights to let us know the current state of the system, we proceeded to wire up the motor controller and connect the drive motors in order to gain basic "hindbrain" level control over the robot.

With all of the basic power control components in place, we developed a system to interface between the sensors on the robot and the autonomy code we wrote to navigate around the Oval.

Sensing System:

Before running the code to receive sensor data and utilize it to autonomously navigate the Oval, we first had to power the sensing and computing suite and connect it to the power and actuation suite. In order to do so, we first supplied power to the Arduino, which will serve to control all of the actuation aspects of our robot autonomously as well as the behavior lights and infrared sensors. A 7-12 VDC barrel connector was wired into the Arduino Mega.

With power wired into the arduino, we then moved on to power the Odroid, which is used to control all of the higher level computing and autonomy functions of the robot, including the LIDAR, USB webcam, GPS, and INS/compass. However, the Odroid only takes 5VDC, which means it must be connected via a voltage regulator like the one used in our roadkill. This particular regulator takes in 10-30VDC and outputs 5VDC.

Although we've already powered the Odroid that's meant to control all of the sensing components, each of these components needs power as well. In order to do that, we wired power into the USB hub and connected the USB hub directly to the Odroid in order to allow the Odroid to receive sensor data.

Once the USB hub and the Odroid were connected, we were then able to connect all of the main flight sensors, specifically the LIDAR, webcam, and GPS/INS/Compass, to the USB hub via usb to usb connections.

In order to connect the sensing and computing suite to the actuation suite, we wired the Roboclaw to the Arduino such that the actuation of the robot can be controlled autonomously. Additionally, we wired the emergency stop from the power suite directly into the Arduino mega in order to have control over the emergency stop even when running the robot autonomously. With all of the sensors hooked into the computing suite and the computing suite connected to the power and actuation suite, we have a fully developed robot, ready to convert code to action.

Software System

The software system consists of the hindbrain, midbrain and forebrain. The hindbrain is the arduino controlling the motors, lights and IR sensors. The hindbrain reads commands from the midbrain, which consists of the odroid, camera and lidar. The midbrain is responsible for obstacle avoidance, as well as reading commands from the forebrain. The arbiter is also situated in the midbrain and is supposed to weight each input and decide on a final velocity output to give to the hindbrain. The odroid, GPS and compass makes up the forebrain, which determines a strategic overall plan for the robot.

Hindbrain

The hindbrain was the interface between the ROS ecosystem and the physical hardware and some of the sensors on the robot. The hardware included the motors and lights (and, optionally, the pan and tilt servos), and Sharp infrared sensor that would be used for stair detection. The hindbrain received messages from the arbiter and was responsible for powering the motors accordingly.

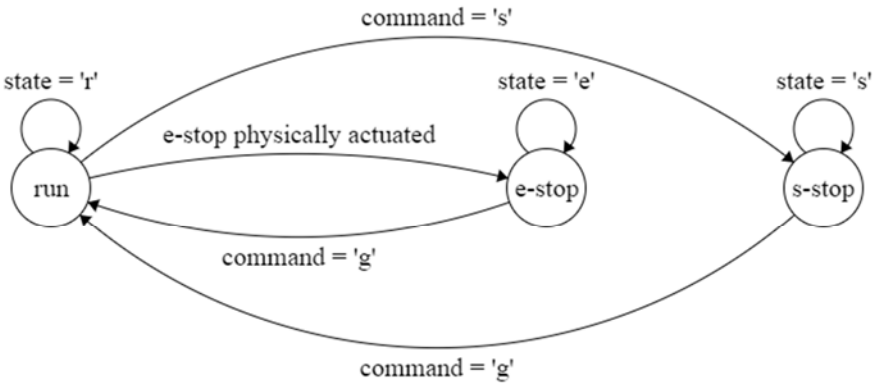

FIGURE 8- HINDBRAIN FSM DIAGRAM

Midbrain

In the midbrain, the Lidar is used mainly to wall follow and the camera is used mainly to avoid obstacles. That said, our camera was only set to detect objects that are bright orange, so it also took lidar distances when the curb or pillars are involved. We have a separate wall follow script that processes the data from the lidar, gives velocity output to wall follow, and also supplies minimum distance outputs to camera object detect. Although we planned to use an arbiter in the beginning, we ended up combining that arbiter into our camera obstacle avoidance script, which decided what to do using if-else statements and cases. The basic main logic is that if there are obstacles, avoid it. If not, wall follow.

Forebrain

In the forebrain, the GPS is used to direct the vehicle to GPS waypoints, accounting for the robot's initial heading and the direction of the waypoint in reference to the robot's current location. The GPS is also used to pinpoint the location of the robot, and determine when it has successfully navigated to a waypoint (as defined by as passing within 3 m of the obstacle). Unfortunately, while the GPS script was running accurately on collected data, we were unable to integrate it with the rest of our code via the arbiter, mainly due to a lack of time.

Individual Part: Midbrain Description

For the mechanical system I CADED a mount for the sonars (which we didn't use in the end), along with some other parts such as the LEDs and the microcontrollers. For the electrical system I crimped wire connections and soldered some components, as well as followed the Soildworks wiring tutorials. But I contributed the most to the software system, and specifically the midbrain, which I will describe below.

1. Working on Lidar data and initial wall following:

First, I worked with Sunny and Isaac to parse the raw lidar data from the ROS topic \scan. \scan.data contained 512 values, each corresponding to the distance at a certain angular direction during the Lidar scan.

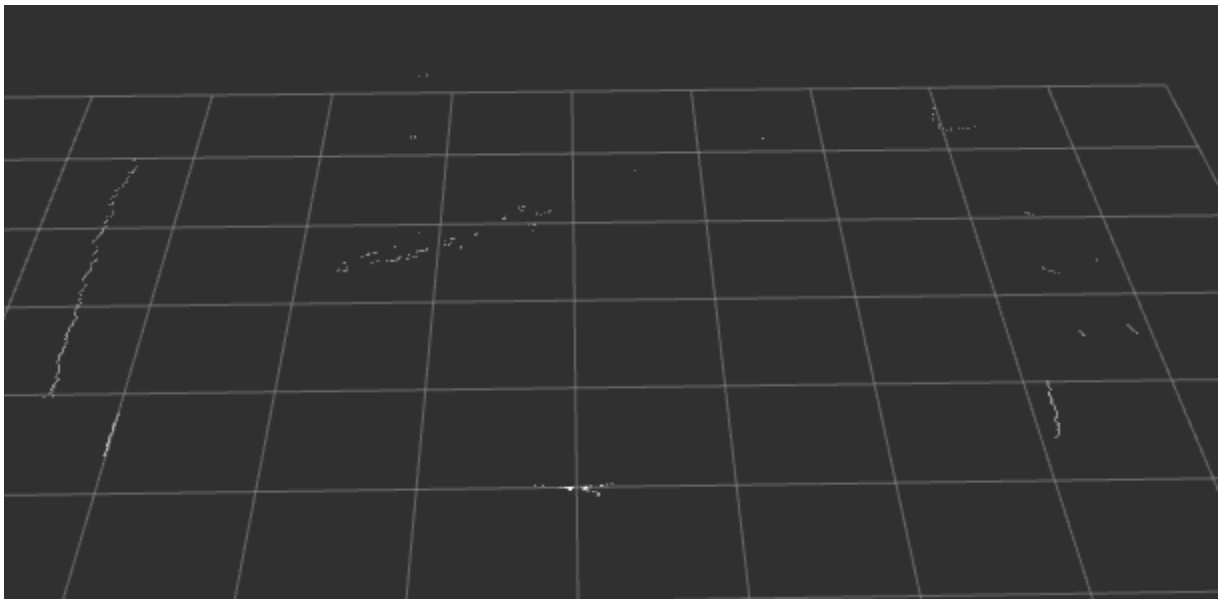

FIGURE 9- RAW LIDAR DATA: EACH POINT IS A DIFFERENT ANGLE

First, we wanted to avoid obstacles if we see a value below a minimum in a direction. As you can see in the code snippet below, I divided the 180 degrees angle range into 11 ranges, corresponding to the 11 elements multi array for turning that the arbiter takes in. Then, if any value in one of the ranges goes below my minimum threshold, I put a negative value in the array which means do not turn towards that range. I changed the linear speed array to do the following: if there are obstacles in front of us, we slow down. We also kept track of the overall minimal distance and its index to help us with wall following later on.

```
std_msgs::Int8MultiArray msgp;
std::vector<signed char> v(DEFAULT_SIZE*2,0); //index 0-11 is for linear velocity and index 12-21 are for turning. Initial values are all zeros
//ranges[0] correspond to 0 degrees (to the right), and ranges[512] corresponds to
//180 degrees (to the left)
for (int i = 0;i<msg.ranges.size();i++){
    if (msg.ranges[i] < min && msg.ranges[i]>0.15){
        min = msg.ranges[i];
        minIndex = i;
    }
    //Mark turning arrays:
    if (i%(int)(512.0/DEFAULT_SIZE) == 0) {
        marked = false;
    }if (msg.ranges[i]<2.0 && msg.ranges[i] >0.15 && !marked) {
        v[i / (512.0 / DEFAULT_SIZE) + DEFAULT_SIZE] = -MAX_turning; //Put negative values in the turning array at the obstacle directions
        marked = true;
    }
}
```

FIGURE 10- ORINIGAL CODE FOR OBSTACLE AVOIDANCE USING LIDAR

Below is the initial code I wrote for wall following. I borrowed the logic from the tutorial at https://syrotek.felk.cvut.cz/course/ROS_CPP_INTRO/exercise/ROS_CPP_WALLFOLLOWING.

Basically, it uses two controllers: one using the angle difference of the wall and one using the distance difference of the wall. Although the tutorial had a proportional and derivative control for the distance difference, to simplify I only used a proportional control for both controllers, as shown below. Then, to integrate with the 11 element array output, I wrote a map function (Figure 12) to convert the angular speed to an index in the array.

```
angleMin = (minIndex - size/2)*ANGLE_INCREMENT;
diffE = (min - wallDistance) -e;
e = min-wallDistance;
angularSpeed = direction*(P*e) + angleCoef*(angleMin - PI*(direction+2)/2); //P P controller
angularIndex = map(angularSpeed, DEFAULT_SIZE);

v[angularIndex] +=3; //angularIndex is the direction we should turn to in order to wall follow
```

FIGURE 11 - ORINIGAL CODE FOR WALL FOLLOWING

```
int map(double angularSpeed){ //maps the desired angular speed to a index in the turning array
    if (angularSpeed>0){  //turn left
        return (5-(angularSpeed/P) +DEFAULT_SIZE);
    }else if (angularSpeed == 0) return DEFAULT_SIZE+5;
    else {  //turn right
        return (5+(-angularSpeed)/P + DEFAULT_SIZE);
    }
}
```

FIGURE 12 - THE MAP FUNCTION IN WALL FOLLOWING

Setting the controller coefficients would require testing, but at this point Sunny took over the wall following code using the same basic logic and did most of the testing. However he decided (after talking with us) to output Twist messages of angular velocity rather than following the 11 element array output to the arbiter.

2. Working on Obstacle avoidance with camera
   Since Sunny took over the lidar wall following code, I started working with Issac to write the camera obstacle avoidance code.

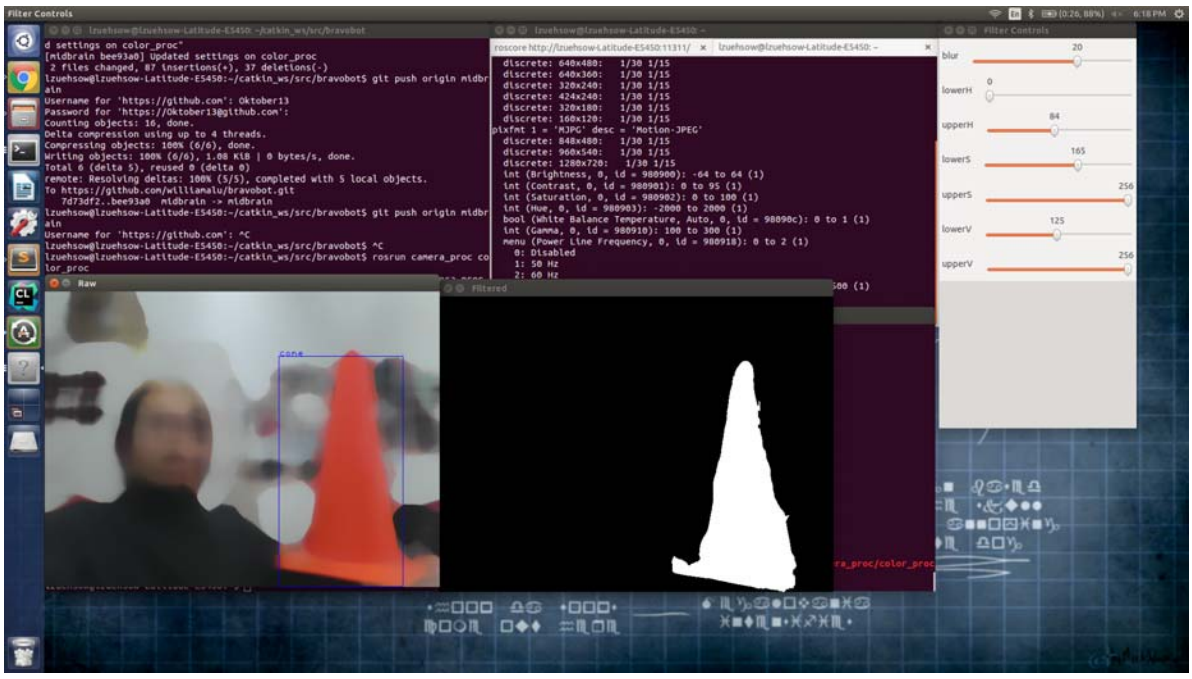

FIGURE 13 - CAMERA CONE DETECTION (BRIGHT ORANGE)

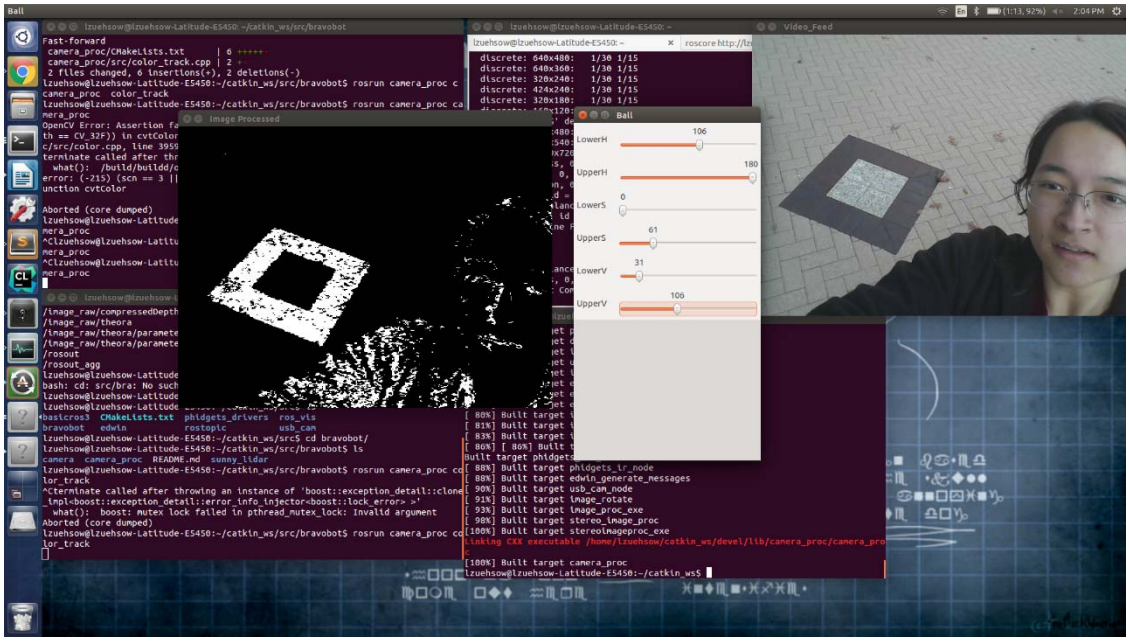

FIGURE 14 - PURPLE SQUARE DETECTION WITH CAMERA

Thanks to Isaac and Lydia, we were able to detect bright orange cones very well, as shown in Figure 13. However, originally we were going to follow the purple square in the center of the tracks, but as shown in Figure 14, the purple color was too similar to many other colors so following the squares wasn't really feasible.

The camera detected the cones as a box with x, y width and height. We decided to break down the image into left, center and right based on where the middle of these obstacle boxes are. If there are no obstacles in the left, we go left. Then, if there are no obstacles in the center, we go straight. At last, if there are both obstacles on the left and in the center, turn right. What if there are obstacles in all three directions? We look at the lidar data. If there is the curb on the left, turn right. If not, turn left. What if there are no obstacles in view? Don't publish anything.

The next step was to differentiate between close obstacles to avoid and farther away obstacles to go towards or ignore. We did that by setting a threshold for the height of the obstacle. If it's above a certain number, it is close enough and we start avoiding it. If there are obstacles far away, it is safe to head toward them. And I implemented that as calculating the average position of the objects far away and go there. The code is shown in the last section of how it worked in the race.

3. Integration with the arbiter

In the beginning, we used Rocco's arbiter that takes in two 11 elements arrays, one for linear speed and one for turning. An element in the middle would mean stop for the linear speed array and go straight for turning. Elements before the middle means either go backwards or turn left respectively, and elements after the middle means go forward or turn right. A negative value for an element means the behavior is not desirable and a positive value means the opposite.

For example, [-5,-4,-3,-1,0,0,3,3,3,0,0] for the linear speed array would mean to go forward at a slow to medium speed.

The arbiter would then take in inputs from all sensors and add them to generate a Twist message output of linear speed and angular speed. Below is Rocco's original code for it:

```
def run(self):
        vel_sum_array = np.zeros(ARRAY_SIZE)
        turn_sum_array = np.zeros(ARRAY_SIZE)
        #vel_sum_array = np.array([1., 2, 3, 4, 5, 6, 5, 5 ,5, 2])
        #turn_sum_array = np.array([0., 0, 10, 1, 2, 3, 3, 2, 1, 1])

        for i in range(len(INPUTS)):
                vel_sum_array += self.vel_array[i]
                turn_sum_array += self.turn_array[i]

        vel = vel_sum_array.argmax()
        turn = turn_sum_array.argmax()
        msg = Twist()
        msg.linear.x = 2*vel/(ARRAY_SIZE-1)-1
        msg.angular.z = 2*turn/(ARRAY_SIZE-1)-1
        self.cmd_vel_pub.publish(msg)
```

FIGURE 15- ORIGINAL ARBITER WITH 11-ELEMENT ARRAY INPUTS AND TWIST OUTPUT

But, since Sunny decided to output Twist messages instead of 11 element arrays for wall follow, I changed the arbiter to the following:

```
def run(self):
        msg = Twist()
        if (self.cmrx!= "N/A" and self.cmrz != "N/A"):
                msg.linear.x = self.cmrx
                msg.angular.z = self.cmrz
                self.cmrx = "N/A" #after reading the camera cmds, set it to null so if it dosn't publish on the next run, follow lidar cmds
                self.cmrz = "N/A"
        else:
                msg.linear.x = self.lidx
                msg.angular.z = self.lidz

        self.cmd_vel_pub.publish(msg)
```

FIGURE 16 - ARBITER THAT ACCEPTS TWIST MESSAGES INPUTS

The basic logic is, if the camera is not publishing, it means there are no cones around, so just follow the wall follower output from the Lidar. If there are camera obstacles, follow the camera command code to avoid them.

This logic has the flaw that the camera may not be publishing at the same speed as the arbiter is running.

Lastly, since for the race the only input to the arbiter were from the camera and Lidar, and there was a flaw in my arbiter logic, I decided to integrate the arbiter into the camera command script as a case. Specifically, if the camera sees no obstacles, follow the Lidar, as shown below:

```
if (far.size() == 0 && close.size() == 0){// if there's no obstacles, follow lidar
    msg.linear.x = lidlinx;
    msg.angular.z = lidAngz;
    std::cout << "I don't see obstacles, following Lidar" <<std::endl;
    pub.publish(msg);
      msg.linear.x = 0;
      std::cout << "No obstacles at all" << std::endl;
      if (savedHeading != heading && savedHeading != -100){
          headTo(savedHeading);
          std::cout << "Headed back to saved heading";
          savedHeading = -100;
      }
      else if (savedHeading == -100){
          msg.angular.z=0;
          msg.linear.x=.3;
          pub.publish(msg);
          std::cout << "Going straight because no obstacle and no saved heading" << std::endl;
      }
}else if (close.size()>0){
      savedHeading = heading;
      std::cout << "Saved a Heading" << std::cout;
    msg.linear.x = .3;
```

FIGURE 17- ARBITER IN CAMERA COMMAND (FOLLOW LIDAR IF THERE ARE NO OBSTACLES)

Besides the things above, I also helped Ragaani to write a code to head back to original setting after obstacle avoidance using IMU data, and helped write the logic in the hindbrain to convert Twist messages to motor speeds.

Summary of Race results

In the first race without obstacles, our robot went past only 2 columns because it veered too far to the left while driving, and collided with the left curb.

FIGURE 18- FIRST RACE RESULTS

In the second race with obstacles we went past 8 columns, successfully avoiding three orange cone obstacles. Once our robot successfully avoided the obstacles, however, it continued on in a straight line until it collided with the wall of the Academic Center, driving through a gap between two columns. The likely cause of this behavior is that our robot finished operating on the commands from our obstacle avoidance code, but wasn't close enough to the next set of obstacles to use them to navigate. Without integrated wall-following code, Bravobot was unable to adjust its heading, and kept going until it crashed.

FIGURE 19 - SECOND RACE RESULTS

In the third race we also went past 7 columns, but Bravobot did not make it around the O. Bravobot eventually veered too far to the left while driving, and collided with the left curb, likely for the same reasons elaborated on above- Once Bravobot finished avoiding its current obstacle, it could not set a new heading and crashed.

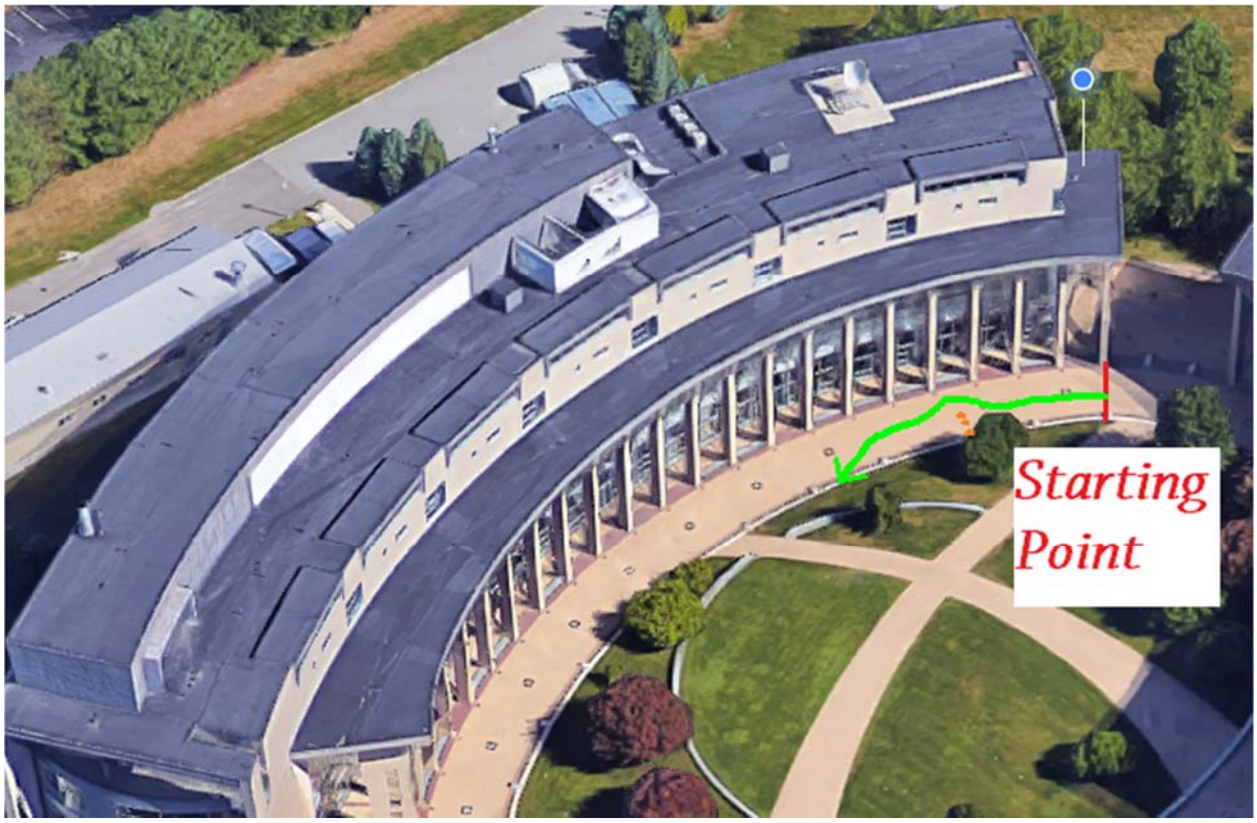

FIGURE 20- THIRD RACE RESULTS

We were 3rd in terms of the number of columns we made. We passed 2 columns in the first race, 8 columns in the second, and 7 in the third. Our accuracy improved significantly once obstacles were added to the course.

How well the midbrain worked in the race

How it worked in the race:

1. The decision to integrate the arbiter into the camera processing: Although it's against design practices, our decision to incorporate the arbiter into the camera obstacle detection would have worked if the wall following script worked. Given the time we had left (it was 12 am Thursday), I made the decision because wall following is a case among the many other cases that were present in the camera command script. If we had a separate arbiter, it would have to take in all the information the camera command script had through some additional topics but it would send out the same thing.

2. The fact that the wall following stopped working the day of the race was a big blow to the functioning of the arbiter. On the day before the race, it was working quite well. It was able to follow the curb with the blocks and only stopped working when there was the gap in the curb. Then Sunny tried to improve the wall following code to handle the gaps, but something went wrong in the process.

3. Since the wall following did not work, we have to change the code in the short time we had on race day to achieve the best outcome we can. This is where the else - if cases in our code really screwed us up. Because we added many cases and sub-cases to the same piece of code, it was difficult to follow and try to account for all the cases. See code in the figure below:

```
}else if (close.size()>0){
      savedHeading = heading;
      std::cout << "Saved a Heading" << std::cout;
    msg.linear.x = .3;
    if (left.size() == 0 && leftLidObst>1.5){ // If there's no close obstacle on the left, go a little left by default
        if (center.size() == 0){
            msg.angular.z = -0.05;
            std::cout << "I'm turning slight left" <<std::endl;
        }else{
            msg.angular.z = -.4;
            std::cout << "I'm turning left" <<std::endl;
        }
        pub.publish(msg);
    }else if (center.size() == 0){ // there might be obstacle on the right, but not in the middle, so we can go straight
        msg.angular.z = 0;
        pub.publish(msg);
        std::cout << "I'm going straight, but there might be obstacles beside me" << std::endl;
    }else if (right.size() == 0 && rightLidObst>2.0){// There are things on the left and in the center, go right
        msg.angular.z = .5;
        pub.publish(msg);
        std::cout << "I'm turning right" << std::endl;
    }else{ //if there's obstacles everywhere, look at lidar data
        if (leftLidObst < 2.0){
            msg.angular.z = 0.5;
            std::cout << "There are obstacles in front, a curb on the left, I'm turning right" << std::endl;
        }else if (rightLidObst<2.0){
            msg.angular.z = -0.5;
            std::cout << "There are obstacles in front, an obstacle on the right, I'm turning left" << std::endl;
        }else{
            msg.angular.z = 0;
            std::cout << "There are obstacles all around me, don't know what to do" << std::endl;
        }
        pub.publish(msg);
    }
    std::cout << "Object is close" << std::endl;
}else if (far.size() > 0){
    float avgx = 0;
    for (int i = 0; i<far.size(); i++){
        avgx+= far[i];
    }
    avgx/= far.size();
    msg.linear.x = .5;
    msg.angular.z = mapAngular(avgx);
    pub.publish(msg);
    std::cout << "Object is far away" <<std::endl;
}
```

FIGURE 21- CAMERA COMMAND LOGIC WITH MANY ELSE IF STATEMENTS BASED ON OBSTACLES AVOIDANCE

a. We first tried using the IMU code we wrote that will turn back to the heading before obstacles once there are no obstacles in view. This did not work as planned because we did not have time to test it and tweak parameters before the race. The code is partly shown below and partly in Figure 17 in the commented lines.

```
void headTo(float desired){
    float error = desired-heading;
    //if negative, drift left, else drift right
    while (heading!=desired){
        msg.angular.z=0.1*error;
    }
}
```

FIGURE 22 - HEAD BACK TO ORIGINAL HEADING WITH IMU DATA

b. So then we tried to use the object is far away code that I wrote to try to go towards objects that are far away. However, although my logic seems correct, I also haven't tested it so it did not work either.

c. Finally, we reverted to the simple logic of: If there are no obstacles, just go slightly left following the curvature of the O. However, it was difficult to tune that constant in a hurry since the curvature of the O changes, so our robot eventually turned too far left that it went into the curb.

4. Despite the setbacks, our cone avoidance worked really well as we were able to turn right of the three cones in a row perfectly. It worked well because we have tested it thoroughly before the race.

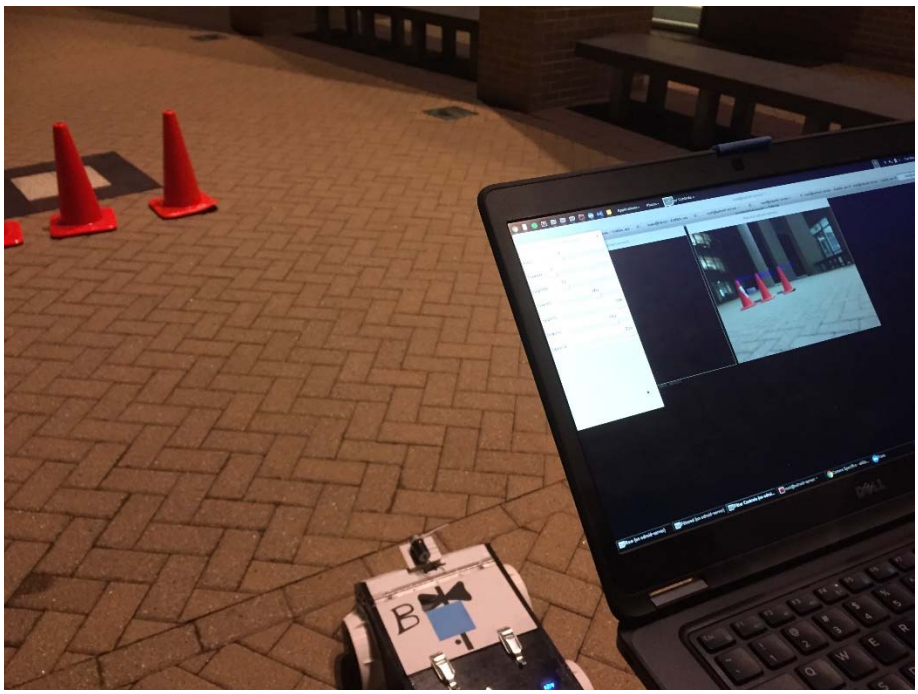

FIGURE 23- CONE DETECTION AND AVOIDANCE ON REAL ROBOT

FIGURE 24 - CONE AVOIDANCE TESTING

How to improve the midbrain:

1. Make wall following work. Talk to Sunny about what he tried to do and fix the bugs. Team Delta told me a good idea to only use a 10 degrees range on the left of the Lidar data to wall follow instead of the one third of left values that Sunny uses.
2. Use Lidar to avoid obstacles in the front range, so that our robot would not run into the curb. We can do this either using my original lidar obstacle avoidance code with the 11 elements array or by a case bases of left, middle and right.
3. In retrospect, doing the arbiter by case might not have been a good idea, and it would be better to use the 11 elements array that Rocco gave us because it is more generic and can handle every situation.
4. Integrate with the GPS and IMU data to find waypoints. Lydia wrote some awesome GPS waypoint finding code that could be integrated into the arbiter.

FIGURE 25- TESTING LATE THE NIGHT BEFORE IN THE COLD

Things I learnt:

1. How to efficiently write and test software with 4 people:
    a. I should probably have helped sunny with the wall following code and pair programmed rather than letting him do it all by himself. I didn't help him because his code seemed to be working very well in the beginning. But three of us worked on the camera obstacle avoidance where he just worked by himself on the lidar. I could also have used the lidar to do more obstacle avoidance.
    b. Write code and test it earlier. It was really not necessary to stay up till 3am, trying to code and test things for the first time. If we worked as efficient as on the last night for two meetings beforehand, we could have made the robot work before the race.
    c. Integrate with the arbiter earlier.
    d. Break up tasks better. Pair programming and collaboration in thinking about navigation strategy is very helpful, but if we don't have much time, we should only have 2 people doing the same thing and the others doing a different part that's equally important.
    e. Robots are fun. Bravobot is a gentleman.

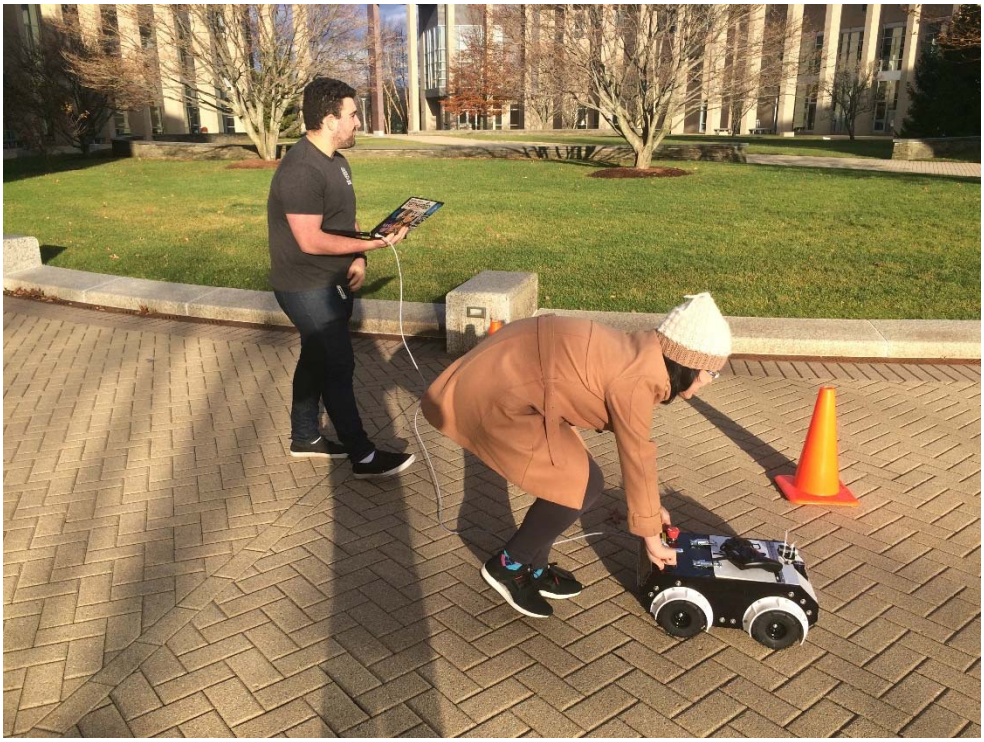

FIGURE 26- PUSHING BRAVOBOT TO GET A ROS .BAG FILE THE LAST WEEK WHEN THE MOTORS BROKE DOWN