**State Estimation and Motion Planning**

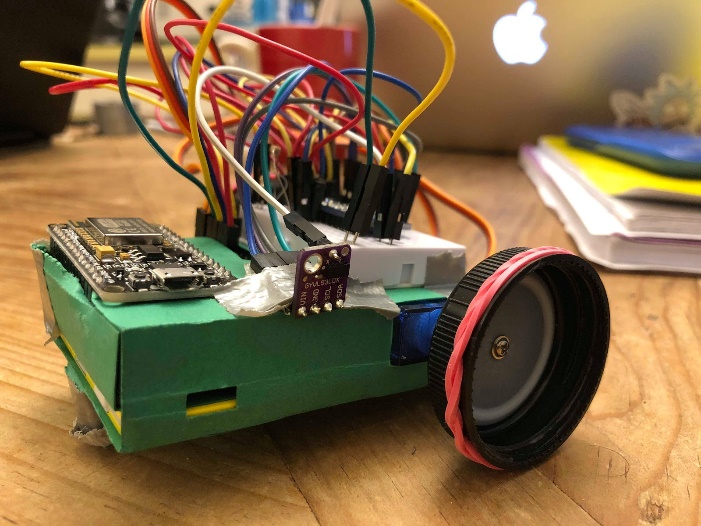
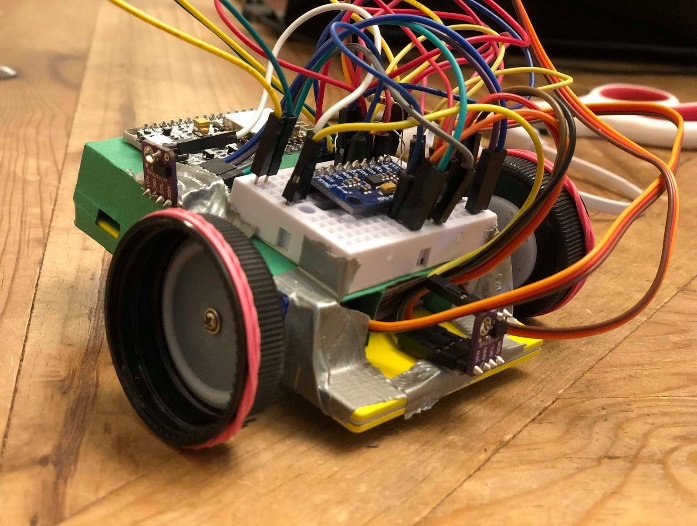
**Jared Leitner – 804436513**

**Ambarish Kowluri – 704423441**

**Mayur Bhandary – 904487050**

**Lab Assignment 3&4**

**Introduction**

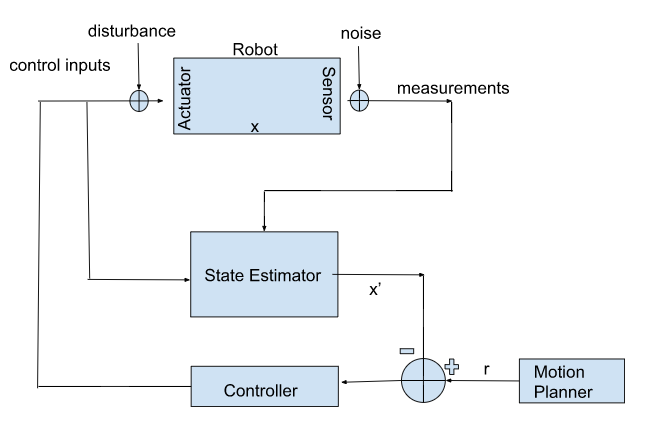
A robot can be defined as a virtual or physical system that can sense, plan, and act. The concept behind an autonomous two-wheel driving robot can be split into two tasks: estimating and evaluating the current state of the robot, and planning the motion or path for the robot. These mechanisms can be scoped through linear algebraic methods and the connection between software and hardware.

The robot above is our two-wheeled driving paperbot.

For the hardware, the bot utilizes 4 different components to account for the state estimation and motion planning tasks. These components include two range sensors (IRs), a magnetometer, and a microcontroller (ESP8266). The purpose for the ESP8266 microcontroller is to maintain the ability to communicate with the bot real-time and through WiFi. The IR sensors are meant to determine the distance between the bot and the obstacle in front of the sensor. In this lab, we place the sensors at the front and right sides of the bot for state measurements. Lastly, the magnetometer keeps track of the direction that the bot is facing. Raw values from the magnetometer are mapped to a unit circle so the direction of the paperbot is discernible.

For the software, we used the Arduino IDE to communicate sensor measurements from the ESP8266 microcontroller to Python using a websocket. To be consistent, we also made state estimation and motion planning calculations using Python. The reason for this coding structure is because performing mathematical matrix calculations is simpler through Python and maintains a pipeline to the Arduino IDE to retrieve real-time sensor measurements. In addition, we used the ESP8266 WiFi module, which hosts an on-board server that controls the bot’s motion. This allowed us to make parameter measurements such as velocity, angular velocity, and more complex parameters such as the disturbance and noise within our system.

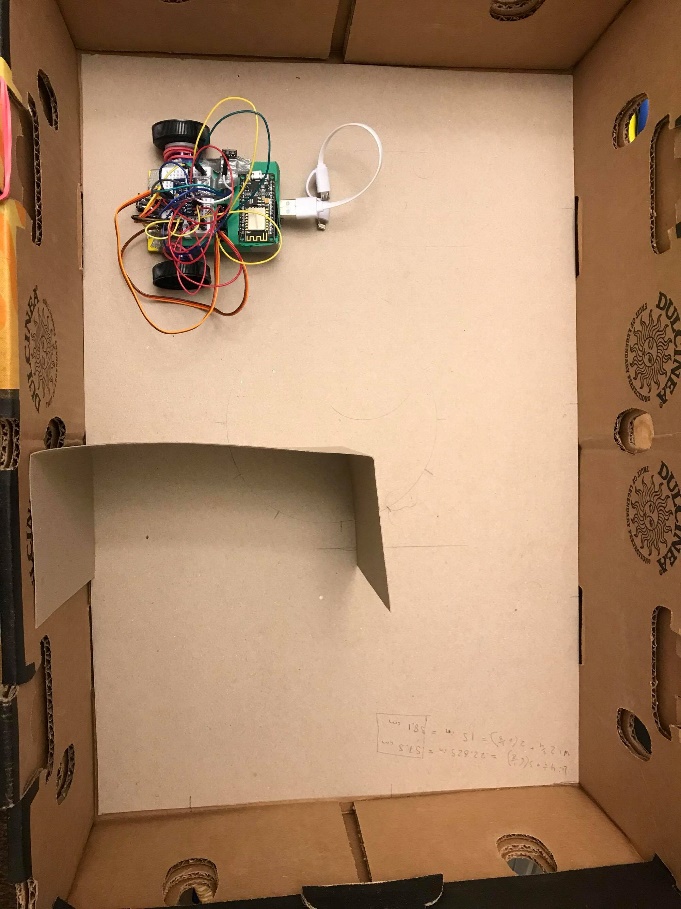
Now let us look at a more detailed outlook for this process. The system can be modeled by the following diagram:



The above control loop combines the state estimator and motion planner to derive an error and implement an appropriate control of the robot. In order to obtain the state estimator and motion planner we implement Python code in order to determine the next state of the robot and plan a path for the robot. This overarching idea leads us to the use of the Kalman Filter and Rapidly-Exploring Random Trees (RRTs) which help in the state estimation and motion planning, respectively. In addition, we examine the disturbance of the paperbot’s actuation as well as the noise in sensor measurements. These features also play a role in the overall control of the robot.

Further technical information and source code can be seen in the GitHub repository below:

<https://github.com/mayurbhandary/EE183DA-Lab3/tree/master>

****The environment in which the car will operate consists of a box with various obstacles that the car will have to avoid. The following photo shows the environment in which our robot will operate. As can be seen, there is one obstacle that the car will have to avoid.

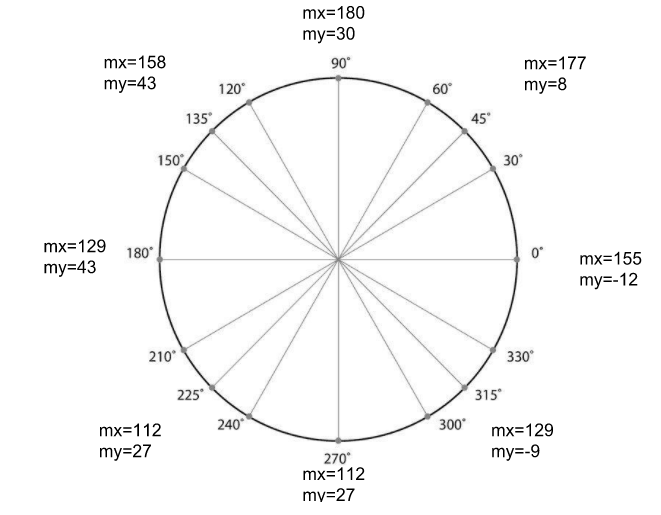
A functioning state estimator and motion planning algorithm must be constructed in order to navigate the bot from its initial state to a desired goal state. Our implementation of both of these blocks is described in detail in the following sections as well as an analysis of the success of each.

The next section describes the details of each sensor measurement, including a mapping from the magnetometers mx and my readings to a heading value as well as how we sent sensor measurements to Python in order to carry out the mathematical calculations.

**Sensor Measurements and Using Python**

Initially, we planned on computing the state estimation onboard the micro controller. However, we soon realized that the computational overhead in forming our state estimate would slow down the paperbot. For this reason, we chose to send our sensor measurements from the paperbot to a python script which would run on a remote machine. This was done by making use of the WebSocket Client in python found at this Github: <https://github.com/websocket-client/websocket-client>.

The Websocket Client allows us to receive information from the paperbot directly and process the information separately. The paperbot’s motion was controlled using the same procedure as in the paperbot.ino file given to us in this lab. Therefore, we had two lines of communication with the paperbot. The first connection is between the python script and the paperbot which allows us to see the raw sensor readings through a computer terminal. The payload received by the python script is as follows: [range sensor front, range sensor right, magnetometer x, magnetometer y, input]. The second connection is between the web browser and paperbot which allows us to control the motion of the paperbot.

The range sensor measurements were straightforward to obtain using the example code provided. The magnetometer data was not so simple. In order to find the heading of our paperbot, we needed to experimentally create a mapping between the sensor measurements and the heading of the paperbot. This was not a simple task. First we had to notice a trend in the mx and my data as the paperbot was rotated through 360 degrees. These data are plotted in the figure shown below. From the figure, we were able to create a linear mapping from my values to angles. In some cases, the my values did not map to distinct angles. For these situations, the mx values were used as tie breakers.

**State Estimation**

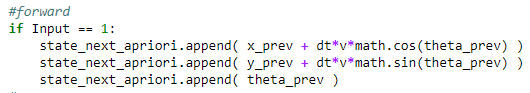
Now that we were able to receive the data from the sensors, state estimation using Kalman filtering can be carried out. State estimation is required due to the fact that there is no way of measuring the state of the robot directly. In our case, the state of the robot consists of an (x,y) position and offset angle Ɵ. We began by analyzing the a priori state estimate.

A PRIORI STATE ESTIMATE

The a priori state estimate can be formulated as follows:

The A matrix in our case is the identity matrix and the B matrix varies depending on the control input U. In our case, this control input takes on values from 1 to 5 depending on the desired motion. A control input of 1 indicates to drive forward, 2 to reverse, 3 to turn left, 4 to turn right, and 5 to remain.

Simplifying the control input take on one of these 5 values allowed us to construct a function in Python that calculated the apriori state estimate based on the current state and control input. A screenshot of the code used to calculate the a priori state estimate when then control input indicated move forward is displayed below:



As can be seen, the change in position in the x and y direction are added onto the previous x and y values in order to determine the a priori x and y estimate. The a priori theta value is not altered in this case, as the robot is only driving forward. It is important to note that both the tangential and angular velocity at which the robot moves and rotates was experimentally determined and is documented in the source code. The a priori state estimates for the other input cases can also be seen in the source code.

This a priori state estimator was tested by commanding the car, via the wifi connection, to move and turn in various directions. We serially output the estimated state values while moving the robot around to see whether or not the estimated values made sense.

A demonstration of this test can be found in the previously mentioned Github repository.

A PRIORI SENSOR ESTIMATE

The a priori sensor estimate can be formulated as follows:

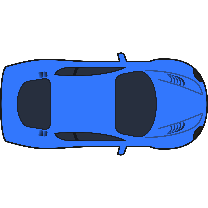
In our case, this sensor estimate was determined only using the a priori state estimate. A function was constructed in Python to carry out this calculation. The following diagram shows the set up for determining the a priori sensor estimate.

**y = BOX\_WIDTH**

**y = 0**

**x = 0**

**x = BOX\_LENGTH**



The input to the sensor estimation function is the a priori state estimate, which consists of a positional (x,y) coordinate and an offset Ɵ from the line y = 0. Given these parameters, we can construct equations for the lines that go through the front and right IR sensors (shown in red). Using these equations, we determine the intersection points of the left, right, top, and bottom wall (red dots). The distance from the a priori positional state to the intersection points can then be determined. From these distances, we want to choose the correct distance measured by the front and right sensors.

The following process is the same for the front and right sensor measurements:

1. Determine the direction vector of the sensor using the a priori theta estimate
2. Determine the four wall intersection vectors by subtracting the a priori position from the intersection points
3. Take the dot product of the wall intersection vectors and the direction vector
4. Discard the negative dot products because they indicate the direction vector and corresponding wall intersection vectors are in opposite directions
5. Choose the distance associated with the smallest positive dot product and assign that to the estimated sensor distance value

This process is use to determine the values read in by the front and right IR sensor. The estimated theta value measured by the magnetometer is simply the a priori theta value.

The sensor estimates were tested by commanding the car to move around, obtaining an a priori state estimate and serially outputting the corresponding sensor estimates. A demonstration of this test can be found in the previously mentioned Github repository.

Although this function produces meaningful sensor estimates, one down side to this approach is that it is difficult to derive the C matrix necessary for calculating the Kalman gain.

CO-VARIANCE MATRICES

In order to find the noise covariance matrix, R, 120 samples from the sensors were taken with the paperbot in stationary mode. The variance of each sensor measurement was taken and placed in the diagonal elements of the matrix. All other elements were set to zero. This means that we assumed that there was no relationship in the variances between sensors. This was an appropriate assumption because we did not expect there to be a correlation between either of the range sensors or the magnetometer.  The diagonal elements of our noise covariance matrix were the following: [.02478, 0.7965, 0.0057].

For the initial Process Covariance Matrix, a 3x3 diagonal matrix with reasonable initial values was chosen. We chose 0.002 to be the process variance for all three entries.

KALMAN GAIN & A POSTERIORI STATE ESTIMATE

As stated in class, the formulas for the Kalman Gain and the a posteriori state estimate are the following:

Unfortunately, our mapping from apriori state estimate to apriori sensor estimate was a nonlinear function. This means that in order to find a C matrix, we need to linearize the system and come up with an approximate C matrix. We were unable to formulate this analysis and were therefore unable to come up with a Kalman Gain. However, as stated by the professor, the kalman gain is not the only option. We could use any value to adjust our state estimate based on the error of the sensor estimate. We proceeded with finding an a posteriori state estimate with an element-wise division of the process covariance matrix and the noise covariance matrix. The results of using this as the Kalman gain were uncompelling and are not presented. In fact, the apriori state estimate was a better measure of the state than the a posteriori state estimate.

**Analysis & Results**

Our state estimator can successfully determine the a priori state and sensor estimates using the methods previously described. These estimates are required in order to determine the Kalman gain and the a posteriori state estimate.The covariance matrices for sensor noise and measurement uncertainty, which are also necessary for calculating the Kalman gain, were correctly derived as explained above.

The final component of the Kalman gain is the C matrix, or the matrix that maps from the a priori state estimate to the a priori sensor estimate. We were unable to derive this matrix from our sensor estimation function.

Our a priori state and sensor estimates are accurate when there are no obstacles in the box. However, when an obstacle is present it is not guaranteed that the sensor estimate is correct. This is because our algorithm only checks the distances from the car to each of the walls. If an obstacle was in between the car and one of the walls, our function would not pick that up and return the distance from the car to the wall instead of the correct distance from the car to the obstacle.

**Motion Planning**

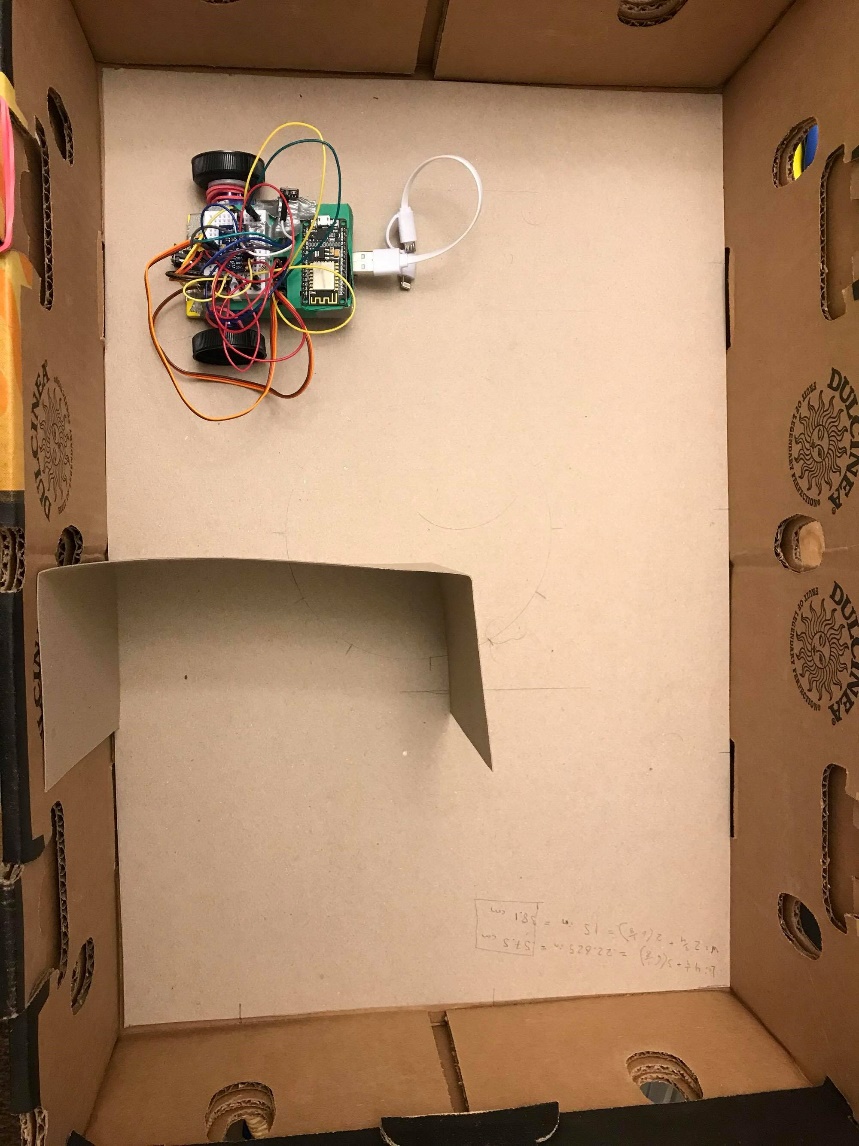
Motion planning is required for non-holonomic systems in which the number of input controls is less than the robot’s degrees of freedom. Our car has a three degree of freedom state and only two control inputs (one for each motor), meaning the system is non-holonomic. Using the Rapidly Exploring Random Trees (RRT) algorithm, a trajectory plan can be formed to avoid walls and any obstacles with in the robot’s environment.

The scope of our motion planning algorithm is to use RRT to construct a list of vectors that describes the path needed to get from the initial state to the goal state. From this list of vectors, a string of inputs consisting of values 0 through 5 will be determined. The numbers 0 through 5 refer to different instructions including drive forward and reverse, as described in the state estimation section.  A predefined obstacle space will also be fed to our motion planner in order to test its ability to construct a path that avoids the obstacle. This planning is carried out offline, or before the car is given any control inputs.

The following picture is an example of a motion plan that the RRT algorithm would construct that avoids our obstacle. The red vectors and points refer to the edges and nodes along the desired path from initial to goal state. The green vectors and points are other edges and nodes stored in the tree but not a part of the trajectory plan.

**Goal State**

**Initial State**



In our code, the RRT algorithm randomly selects a point within the dimensions of the box. This random point is compared to the nearest point in our family of nodes. A vector is generated between this random point and the nearest point and the step-size magnitude of this vector will generate the next node in our node family. But before adding this node, we must check for obstacles.

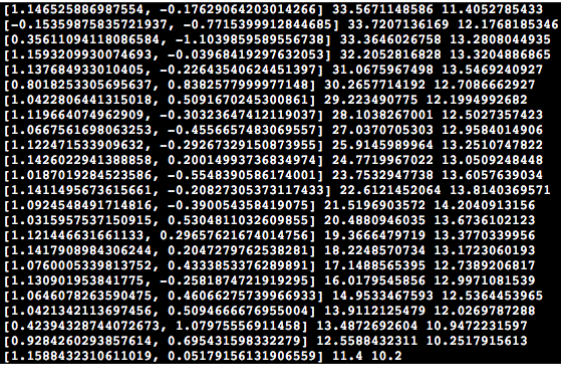
The obstacle space was constructed by creating a list of (x,y) coordinates. As can be seen from the above picture, our obstacle has a small width. We assumed a larger width when constructing the obstacle space in order to account for the width of the actual car.

The RRT algorithm simulates this obstacle by mathematically estimating a zone around the obstacle. The step-size magnitude of the vector retrieved before will determine whether any of its points hit the obstacle. If not, then the node is finally added to the node family. A path is formulated between the new node and the nearest node which undermines the objective behind RRT. Once the new node is within a small radius of the desired state, we now determine the path to get to that state. We iterate through the list of paths (edges) recorded and check to see what path gets us from the goal state to the previous state. The paths are traced with a starting point and vector thus subtracting the goal state with the vector will result in the previous node. Repeatedly performing this will allow us to reach the initial state and we know the path to get to the desired position.

The output of our RRT algorithm can be seen in the next section.

**Analysis and Results**

Our motion planner is able to construct a list of edges that describe the path of motion required to reach the goal state. From this list, another list of control inputs is determined. For example, we analyzed the path for an initial position of (11.4, 10.2) and desired position around (36.5, 11.4). We desired to create this path with an accuracy of within 2 cm of the desired state. An example of the series of paths is shown below:



The numbers outside the brackets are the current state (x,y) of the robot. The numbers within the brackets is the direction the robot is facing in terms of a vector. As can be seen, the path is traced back to the initial state. The goal state achieved from this path is (34.7, 11.2). These edges describe changes only in the positional state of the robot (x,y) and do not directly contain information regarding the change in theta. The change in theta required to move between two nodes was determined geometrically.

We were unable to translate this path into control inputs. However, the approach taken is to take the vectors and produce directional angles with respect to each current state. This approach was tedious, and this caused many problems in allowing the robot to move autonomously. Yet, we tested out this path list and manually controlled the robot to see whether the list would lead to the desired state. We were unable to establish a controller that would interpret these paths, but manually utilizing this to identify whether the path is well-defined showed the true accuracy of achieving the goal state. With that being said, we mainly could have improved the performance of the robot by taking into account the directional angles for each path with respect to the robot’s current state even though it may be tedious.

**Conclusion**

Overall, parts of our state estimator and motion planner work correctly, however, there are other parts required for each block to fully function that we were not able to complete.

For our state estimator, the a priori state and sensor estimates can be correctly determined, however, the a posteriori state estimate using the Kalman gain was not able to be determined.

For our RRT motion planning algorithm, the path required to move from the initial state to the final state could be determined, however, the corresponding control inputs were not able to be found.