The Gauntlet

Thomas Jagielski, Jasmine Kamdar

Quantitative Engineering Analysis 1 - Olin College of Engineering - May 2019

1 Introduction

The goal of this project was to investigate a robot's ability to navigate around a region using data from LIDAR scans. Specifically, the robot had to navigate from one spot to another, while avoiding obstacles. The robot had a LIDAR to detect objects and collect data about the course. K-means squared was used to evaluate the data, and determine the location of obstacles. This project entailed creating a contour plot and gradient ascent path for the robot to follow.

1.1 Challenge

We attempted to traverse the Gauntlet's pen without knowing the location of the obstacles. This corresponds to the Level II Challenge which states, "You are given the coordinates of the BoB, but you must use the LIDAR to detect and avoid obstacles." A diagram of the Gauntlet is shown in Figure 1,

1.2 Assumptions

- We knew the location of walls
- We knew the location and radius of the bucket (goal)
- There were only two obstacles in the 'pen'
- We started at position (0,0) in the frame of the pen with an initial heading of 0°

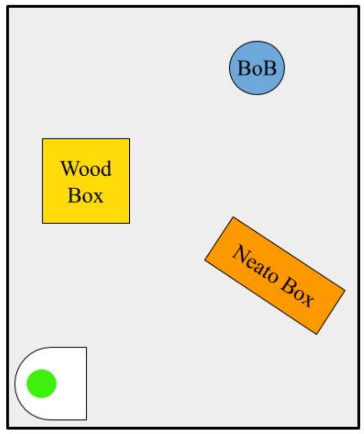

Figure 1: Overview of the Course

1.3 Approach

When first attempting this challenge, we used the RANSAC algorithm for obstacle detection. However, the algorithm yielded lines of best fit that crossed the pen, namely the gap between the obstacles. We then removed the data-points that represented the wall using x and y bounds for where we knew the wall to be. Similarly, we were unsuccessful when isolating the obstacle data-points, as the lines of best fit still connected across the gap.

It was only when we restricted the length of line as being less than 0.7 meters when we began to isolate the walls of obstacles.

When shifting to the second challenge we were curious about better ways to detect where the obstacles were. To do this, we implemented k-means clustering. We were able to represent each obstacle as a single data point extracted from a cluster of LIDAR data.

For the implementation onto the NEATO, we defined a sink at each of the points that were the cluster centers (obstacles) and a source at the known goal location.

2 Methodology

2.1 Preprocessing Data

Once the NEATO has entered the pen, data would have been from LIDAR scans of the course. The first part of evaluating the data is cleaning it and converting the coordinates from polar to Cartesian. The data provided is a radius and a theta value. When the radius is 0, there is no object in range at that angle, and the data is irrelevant for our purpose. So, those coordinates are removed from the data set. In order to convert the radius and theta to x and y-coordinates, we used the pol2cart function on MATLAB, which uses the following equations.

$$x = r * \cos(\theta); \tag{1}$$

$$y = r * \sin(\theta) \tag{2}$$

In order to understand the data received from the LIDAR we need to convert the LIDAR's coordinate system to match the Gauntlet's coordinate system. We know the center of the robot's coordinate system with respect to the Gauntlet is an angle of ϕ. The NEATO's coordinate system can also be offset from the Gauntlet's coordinate system in the x direction (xN) and in the y direction (yN). We also know that the LIDAR's coordinate system with respect to the center of the NEATO's is offset in the x direction by the distance (d) between the LIDAR's center and the center of the NEATO. The equation for converting the LIDAR's Cartesian coordinates to the center of the NEATO's Cartesian coordinates is shown in Equation 3. Equation 4 shows the conversion to the Gauntlet's coordinate system.

$$NC = \begin{bmatrix} 1 & 0 & d \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r * cos(\theta) \\ r * sin(\theta) \\ 1 \end{bmatrix} \tag{3}$$

$$GC = \begin{bmatrix} 1 & 0 & -xN \\ 0 & 1 & -yN \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\phi) & sin(\phi) & 0 \\ -sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} * NC \tag{4}$$

Another set of data to be removed is the group of data points representing the target. Since we want all the data points to represent obstacles, we would want the goal to not be considered an obstacle.

The last set of data we excluded before analysis were the walls. We knew the boundaries of the walls, so we simply excluded any points past the known x and y bounds. We did this because the walls created too many data points that distracted from the main obstacles in the course.

2.2 Source and Sink Placement

We computed the location of the obstacles by using the k-means clustering algorithm. K-means clustering is an algorithm that generates a "cluster center" by selecting random points across a dataset. The number of points corresponds to the number of clusters represented by the set, which is an input to the algorithm. Then each point is assigned to a cluster based off the closest "cluster center" represented in Euclidean distance. Once assigned, the new cluster center is computed as the mean of all the data-points in the cluster. This process is repeated until all the data-points are assigned a cluster. Then the variance of the clusters is computed, and the k-means clustering algorithm is repeated until the variance for all the clusters is a minimum.

Knowing there were only two obstacles in the gauntlet, we used two clusters in the k-means clustering algorithm. Furthermore, we used the MATLAB 'kmeans' function.

At each of the cluster centers, we placed a sink. We then scaled the sinks by 1.2 in order to make the obstacles large enough for the NEATO to avoid.

As we knew the location of the goal, we placed a source at this location. However, to make the magnitude of the source larger, we included a second source within 0.01 meters from the initial goal location. Furthermore, we scaled the magnitude of these two sources by a factor of three. The final equation of the sources and sinks is shown in Equation 5.

$$
\begin{aligned}
f(x,y) &= 1.3 * \left[\log\sqrt{(x-0.1388)^2+(y-1.3588)^2}\right] \\
&+ 1.3 * \left[\log\sqrt{(x-0.8103)^2+(y-0.8296)^2}\right] \\
&- 3 * \left[\log\sqrt{(x-0.9)^2+(y-1.75)^2}\right. \\
&+ \left.\log\sqrt{(x-0.91)^2+(y-1.74)^2}\right]
\end{aligned} \tag{5}
$$

2.3 NEATO Implementation

In order to have the NEATO drive to the goal using the expression for the location of the sources and sinks we can use a gradient ascent algorithm to plan the path.

Gradient ascent is an algorithm in which we can find a local maximum of a function. To do this, the algorithm will follow the steepest path, which is in the direction of the gradient vector. This vector magnitude is scaled by λ. Thus, the new position can be derived by Equation 6. $\vec{r_i}$ is the current position vector given as $\begin{bmatrix} x_i \\ y_i \end{bmatrix}$, $\vec{r_{i-1}}$ is the previous position vector given as $\begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}$, λ is the scaling factor for the gradient vector, and $\nabla f(x,y)$ is the gradient of the function evaluated at the current position.

$$\vec{r_i} = \vec{r_{i-1}} + \lambda * \nabla f(x,y) \tag{6}$$

For each step, λ will change by a factor of δ. Which can be formalized in Equation 7 where λ_i is the current λ for a given position, δ is the factor that will scale λ from step to step, and λ_{i-1} is the previous λ value.

$$\lambda_i = \delta * \lambda_{i-1} \tag{7}$$

A greater δ value will create a larger scale factor so that if the gradient vectors are decreasing it won't take too many small step sizes. However, there needs to be a balance so that the step sizes aren't too big that the gradient plot is inaccurate or that the algorithm overshoots.

We pre-computed the gradient ascent path by finding the x and y positions at each step, and then passed it into a function in which the NEATO was sent commands to drive to each of the positions that were yielded by the gradient ascent algorithm.

We broke the NEATO's program into two steps - turning to the new heading and driving forward. We broke each of these steps into individual functions.

For the turning function, the left and right wheel velocities were computed separately. The left wheel velocity is given by Equation 9.

$$V_L = V - \omega * \frac{d}{2} \tag{8}$$

where V is the linear velocity, ω is the angular velocity, and d is the wheel distance. The right wheel velocity is given in Equation 10.

$$V_R = V + \omega * \frac{d}{2} \tag{9}$$

where all the variables are the same as the V_L equation.

The next function called was the drive forward function. This function was given to us on the first day of class.

3 Experimental Results

After scanning the data, a contour plot is generated, along with a gradient path for the NEATO to follow. Figure 2 shows the contour plot and gradient path. It also displays the gradient vectors at various locations. The blue regions show the sinks that represent the locations of the obstacles and the yellow region shows the source that represents the target location.

After running the NEATO in the Gauntlet and collecting the wheel encoder data, we were able to find the experimental position. Figure 3 shows the experimental path compared to the theoretical path. The total time the NEATO took for the computations and driving the path was 42 seconds.

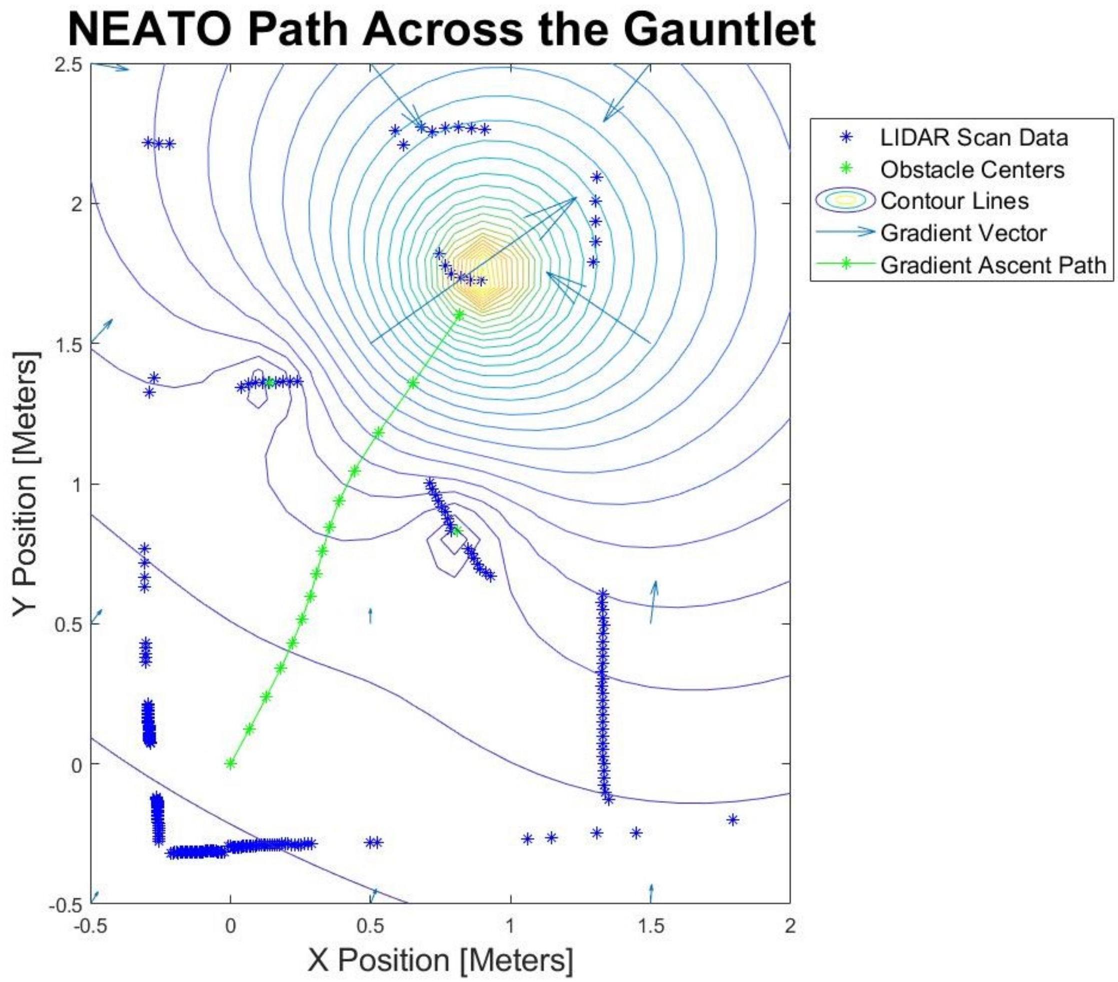

Figure 2: Contour plot, gradient path and gradient vectors

As we can see there is quite a bit of error in the expected and experimental path. However, the error seems to be the most in the middle of the path, around 0.5 meters in the x-position. Both the beginning and ending of the paths seem to have little error. This error is visualized below in Figure 4.

As there were many more points in the experimental path, we found the closest x values across the experimental and theoretical paths and computed the difference in the y value. We chose this as the error metric because both of the paths were moving to the right - and only had an increase in

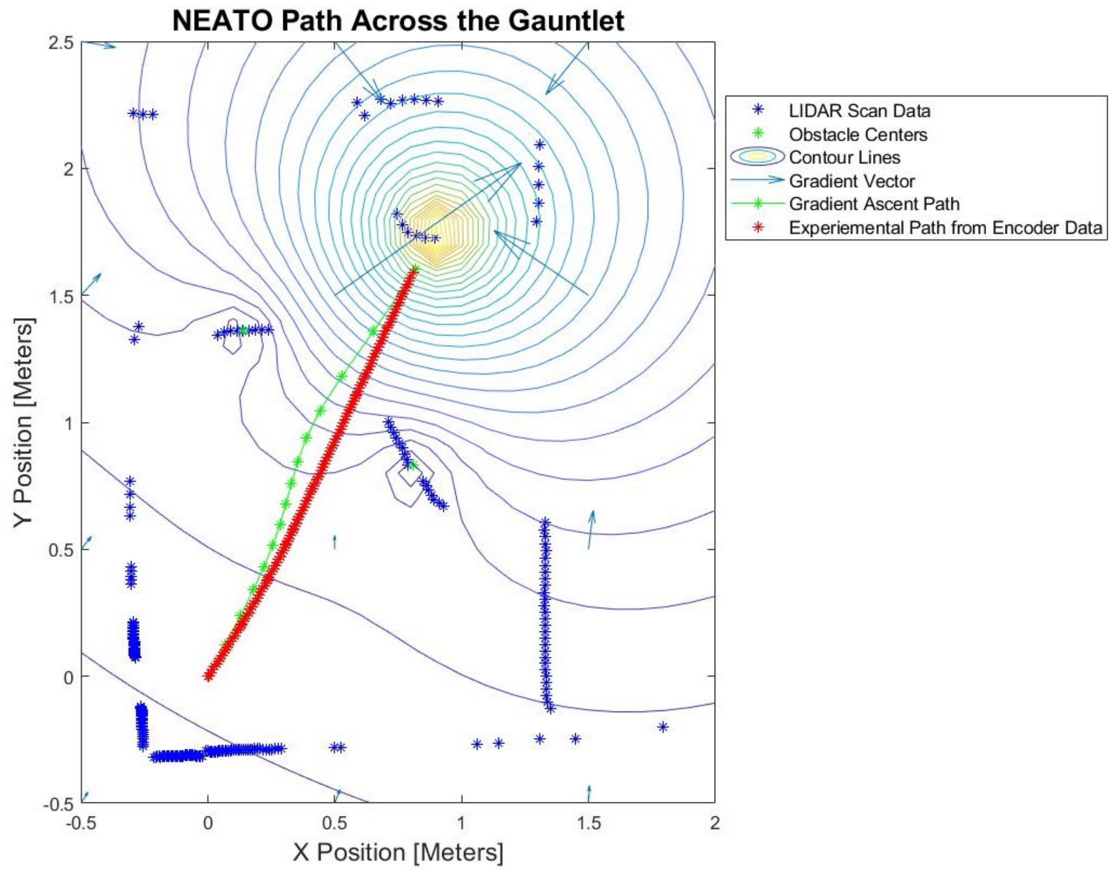

Figure 3: The planned path for the NEATO is shown in green, and the experimental path is shown in red.

x. With this, we found the direction which had the most error to be the vertical direction; and thus, it would fully encapsulate how well the NEATO performed.

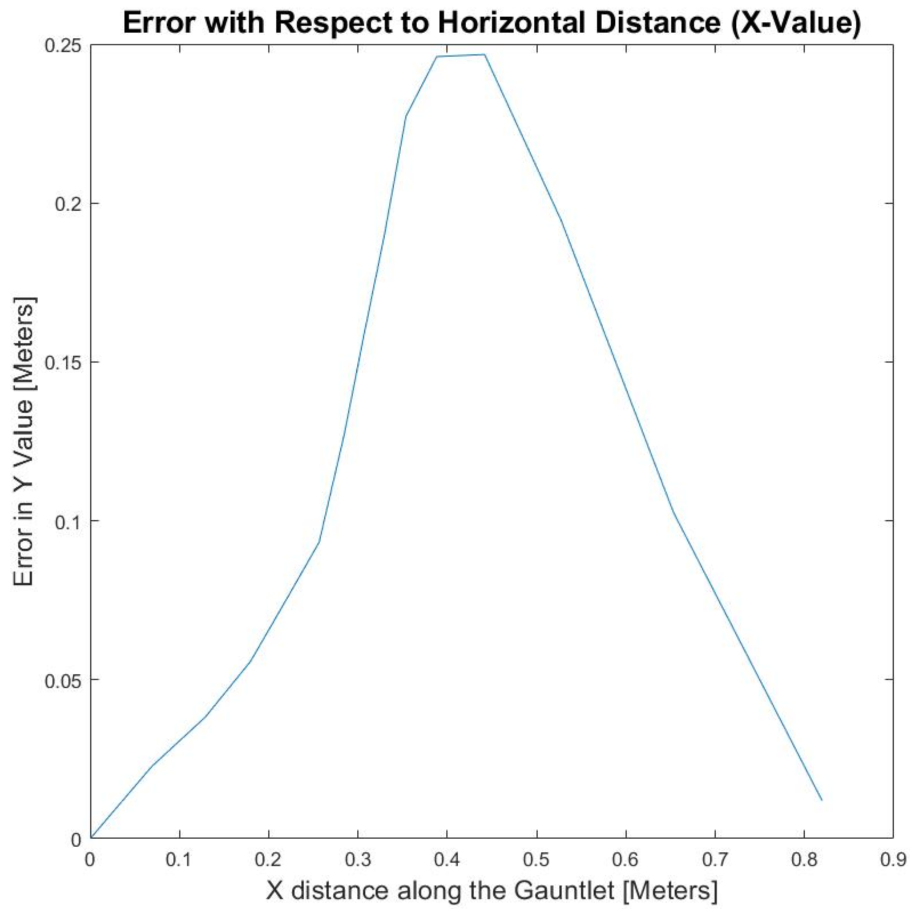

Figure 4: Vertical error for specific x distances along the path.

4 Discussion

The first failed result was when RANSAC was implemented and the best fit lines were created from the wall data points to the obstacles. To combat this, we removed the wall data. However, that still did not solve the problem because lines were being formed between obstacles. We tried adjust thresholds from data points and the lines to how many data points had to be within this threshold.

When implementing a completely different data organization method (k-means clustering) we found a much greater success. Still with the k-means method, there were variables we had to change around. We needed to adjust how many clusters we wanted to use. We changed around the number of clusters to see if more clusters would give us a better understanding of the size of the obstacle. However, when we did this, the contour plot started to get messy with many sinks. We were able to conclude that two clusters for two obstacles would do the job.

We believe the error source has come from a wheel distance that was too small compared to the actual wheel distance of the NEATO. We believe this because the parts of the curve that have the most error is when the the function had the most curvature. Namely, the NEATO did not turn as much as was anticipated; however, it still ended in the same position, which indicates to us that there was a consistent lack of turn between the paths.

5 Conclusion and Future Work

From this project, we determined that RANSAC can be useful for creating best fit lines for one cluster of data. However, when there are multiple, RANSAC will create lines through different clusters of data, which is not what one wants when organizing clusters of data. On the other hand k-means divides up clusters fairly well. One problem with k-means is that you have to set how many clusters there will be in the data set. So, if we wanted the robot to sense an unknown number of obstacles we would have to add a different function to create clusters.

One future step would be to define the walls as walls rather than remove them from the data set. A possible solution to do this would be to sweep horizontal and vertical lines that fit with a certain amount of data points that match the walls. Another future implementation would be identifying the circle base of the bucket, such that we would not have to exclude the bucket data by knowing its position. Instead, before using k-means, we would identify the radius of the bucket and exclude all of the data that have this radius to call that the source.

6 Video

A link to our NEATO completing the Gauntlet Challenge is provided below:
Gauntlet Level Two - Jasmine Kamdar and Thomas Jagielski
or,
https://www.youtube.com/watch?v=BYIJMM5$_{H}1s$

7 Code

7.1 Level Two Script

```
% Initialize ROS for the robot
rosshutdown
rosinit('10.0.75.2',11311, 'NodeHost','10.0.75.1')
sub = rossubscriber('/stable_scan');

% Collect data at the pen origin
scan_message = receive(sub);
r = scan_message.Ranges(1:end-1);
theta = [0:359]';

% Put r and theta as objects in a "data" structure
data.r = r;
data.theta = theta;

% Set the goal position coordinates
goal = [0.9,1.75];
% Set the radius of the goal bucket
radius = 0.1778;
% Load the data collected from the scan to plot data in retrospect
%load('Challenge2 Scan.mat')

% Call the mapToGlobal function to put the data in the frame of the ganulet
% pen
[x,y] = mapToGlobal(0.1016,0,0,0,data);
```

```
% Plot the data to ensure it looks correct
plot(x,y,'b*')

% Delete the walls data
a = 1;
for k=1:length(x)
    % Ensure the data is not past the known position of the walls
    if y(k) < 2 && y(k) > 0.1 && x(k) > -0.25 && x(k) < 1.25
        y_new(a) = y(k);
        x_new(a) = x(k);
        a = a + 1;
    end
end
clear x y
b=1;
for m=1:length(x_new)
    % Delete the data for the goal based on position and radius of the
    % bucket
    circdistance=sqrt((goal(1)-x_new(m))^2+(goal(2)-y_new(m))^2);
    if circdistance > radius
        y(b) = y_new(m);
        x(b) = x_new(m);
        b = b + 1;
    end
end

% Use k-means clutering algorithm to detect the two obstacles.
[IDX, C] = kmeans([x;y]',2)
hold on
% Plot the position of the center of the cluters
plot(C(:,1),C(:,2),'g*')
```

```
% This code is adopted from Day 7's "ScalarFieldGradient.m"

% Initialize the ranges and point spacings
[X,Y] = meshgrid([-0.5:0.1:2],[-0.5:0.1:2.5]);
[X1,Y1] = meshgrid([-0.5:1:2],[-0.5:1:2.5]);
% Define the source and sinks based on the position of the cluster centers
syms x y
sink1 = log(sqrt((x-C(1,1))^2+(y-C(1,2))^2));
sink2 = log(sqrt((x-C(2,1))^2+(y-C(2,2))^2));
source=log(sqrt((x-0.9)^2+(y-1.75)^2))+log(sqrt((x-0.91)^2+(y-1.74)^2));
f=1.3*sink1 + 1.3*sink2 - 3*source;
% Find the symbolic gradient of the function
g = gradient(f,[x,y]);

% Plot the function's contour
contour(X,Y,subs(f,[x,y],{X,Y}),30)

% Make a vector field to represent the gradient at specific points in the
% curve
G1 = subs(g(1),[x,y],{X1,Y1});
G2 = subs(g(2),[x,y],{X1,Y1});
quiver(X1,Y1,G1,G2)
axis equal

syms x y;
% Run the gradient descent algorithm to find the expected path of the NEATO
r = gradientDescentXY(f,0.1315,0.86,0,0,goal(1),goal(2));
plot(r(1,:),r(2,:),'*-g')
title('NEATO Path Across the Gauntlet')
xlabel('X Position [Meters]')
ylabel('Y Position [Meters]')
% hold off
```

```
% Drive the NEATO in the path of gradient ascent to the goal
neatoGradient(r,0)

%% Error Metric
% Load the encoder data
encoder_data = load('Challenge2.mat');
% Find Wheel Velocity
diffEncoder = diff(encoder_data.dataset(:,1:3));
V_l_encoder = diffEncoder(:,2)./diffEncoder(:,1);
V_r_encoder = diffEncoder(:,3)./diffEncoder(:,1);
time_encoder = encoder_data.dataset(:,1);

% Initialize the data to after start
V_l_encoder = V_l_encoder(17:211);
V_r_encoder = V_r_encoder(17:211);
time_encoder = time_encoder(17:211)-time_encoder(17);

% Find the linear speed
encoder_linear_speed = (V_l_encoder + V_r_encoder)/2;

% Find angular velocity
d = 0.27;%wheel distance [m] - changes for every robot
omega_encoder = (V_r_encoder - V_l_encoder)./d; %find the angular speed of the robot

% Set the initial Position
pos_x = 0;
pos_y = 0;
theta = 0;

% Initialize matricies for the experiemental position (x,y) and the heading
% angle
exp_pos = zeros(length(V_l_encoder),2);
theta_mat = zeros(length(V_l_encoder),1);

```

```matlab
% Find the change in time for each step
dt = diff(time_encoder);

% Compute the position and heading given by the encoder data
for p=1:length(V_l_encoder)-1
    if p > 1
        exp_pos(p,1) = exp_pos(p-1,1)+(encoder_linear_speed(p,:)*dt(p,:)*cos(theta));%the x-coordinate
        exp_pos(p,2) = exp_pos(p-1,2)+(encoder_linear_speed(p,:)*dt(p,:)*sin(theta));%the y-coordinate
        theta = theta + omega_encoder(p,:)*dt(p,:);%the heading
        theta_mat(p) = theta; %save the angles for various time steps
    else
        exp_pos(p,1) = pos_x + (encoder_linear_speed(p,:)*dt(p,:)*cos(theta));%the x-coordinate
        exp_pos(p,2) = pos_y + (encoder_linear_speed(p,:)*dt(p,:)*sin(theta));%the y-coordinate
        theta = theta + omega_encoder(p,:)*dt(p,:);%the heading
        theta_mat(p) = theta; %save the angles for various time steps
    end
end

% Plot the experimental data
plot(exp_pos(:,1),exp_pos(:,2),'r*')
legend('LIDAR Scan Data','Obstacle Centers','Contour Lines','Gradient Vector','Gradient Ascent Path','Experiemental Path from Encoder Data')
hold off

% Compute the vertical error for each of the points in the theoretical path
```

```
for z=1:length(r)
    [Y,I]=min(abs(r(1,z)-exp_pos(:,1)));
    yerror(z)=abs(r(2,z)-exp_pos(I,2));
end

% Create a figure with for the metric over the distance of the Gauntlet
figure;
plot(r(1,:),yerror)
title('Error with Respect to Horizontal Distance (X-Value)')
ylabel('Error in Y Value [Meters]')
xlabel('X distance along the Gauntlet [Meters]')
```

7.2 Map to Global Script

```
function [r_G_x,r_G_y] = mapToGlobal(d,x0,y0,heading,data)
    % Function to translate data in the frame of the LIDAR to the global
    % frame of the Gauntlet pen

    syms x_N y_N phi theta r;
    r_N = [r*cosd(theta)-d;r*sind(theta)]; % Map the LIDAR data to the frame of the NEATO
    r_G = [x_N-d*cosd(phi)+r*cosd(theta+phi);
           y_N-d*sind(phi)+r*sind(theta+phi)]; % Map the NEATO data to the global frame

    a = 1; % Initialize 'a' as the index value
    % Initialize r and theta clean matricies
    r_clean = zeros(length(nonzeros(data.r)),1);
    theta_clean = zeros(length(nonzeros(data.r)),1);
    % Clean the data for non-zero values of r
    for k=1:length(data.r)
        if data.r(k) ~= 0
            r_clean(a) = data.r(k);
            theta_clean(a) = data.theta(k);
```

```
                a = a + 1;
            end
        end

        length(nonzeros(data.r)) == length(r_clean); % Ensure the length of the
        % clean data is the same length as the non-zeros data
        r_N_x = zeros(length(nonzeros(r_clean)),1);
        r_N_y = zeros(length(nonzeros(r_clean)),1);

        for k=1:length(r_clean)
            % translate to the NEATO frame by substituting in correct values
            r_N_x(k) = subs(r_N(1,1),[theta,r],[theta_clean(k),r_clean(k)]);
            r_N_y(k) = subs(r_N(2,1),[theta,r],[theta_clean(k),r_clean(k)]);
        end

        % Convert the "symbolic" matrix to matrix of doubles
        r_N_x = double(r_N_x);
        r_N_y = double(r_N_y);

        r_G_x = zeros(size(r_N_x));
        r_G_y = zeros(size(r_N_y));
        for k=1:length(r_clean)
            % translate to the global frame by substituting in correct values
            r_G_x(k) = subs(r_G(1,1),[x_N,y_N,phi,theta,r],[x0,y0,heading,theta_clean(k),r_clean(k)]);
            r_G_y(k) = subs(r_G(2,1),[x_N,y_N,phi,theta,r],[x0,y0,heading,theta_clean(k),r_clean(k)]);
        end

```

```
    % Convert the "symbolic" matrix to matrix of doubles
    r_G_x = double(r_G_x);
    r_G_y = double(r_G_y);

%     Code to plot the NEATO and Global Frames

%     figure;
%     plot(r_N_x,r_N_y,'ks')
%     title('Frame of Neato Data')
%     xlabel('X-Distance [Meters]')
%     ylabel('Y-Distance [Meters]')

%     figure;
%     plot(r_G_x,r_G_y,'ks')
%     title('Global Frame Data')
%     xlabel('X-Distance [Meters]')
%     ylabel('Y-Distance [Meters]')

end
```

7.3 Gradient Ascent

```
%% Gradient Descent Function
function r = gradientDescentXY(f, lambda, delta, x0,y0, xf, yf)
    % Function to compute the gradient ascent/descent path and stop when within a
    % threshold distance from the goal
    syms x y;
    % Find the symbolic gradient of the input function
    % NOTE: use -gradient for gradient descent
    grad_syms = gradient(f);
    % Initialize the starting parameters
    lambda(1) = lambda;
    r(:,1) = [x0;y0];
    a=2;
    % Set the threshold distance from the goal
```

```
    rthresh = 0.1778;
    % Compute the initial distance from the goal
    threshdist= sqrt((x0-xf)^2+(y0-yf)^2);
    while threshdist > rthresh
        % Find the new position vector
        r(:,a) = r(:,a-1)+lambda(a-1)*subs(grad_syms,[x
            ,y],[r(1,a-1),r(2,a-1)])
        % Compute the distance away from the point
        threshdist=sqrt((r(1,a)-xf)^2+(r(2,a)-yf)^2)
        % Compute the new lambda value
        lambda(a) = delta * lambda(a-1);
        % Increment 'a'
        a = a+1;
    end
end
```

7.4 NEATO Drive Gradient Ascent

```
function heading = neatoGradient(f,heading_init)
    % Funtion to drive the NEATO in the path of
        gradient ascent
    % Set the wheel distance
    d = 0.27;
    % Set the linear speed to zero so the NEATO will
        turn in place
    V = 0;
    % Set omega
    omega = 0.27;
    syms x y;
    % Use the gradient ascent positions to drive to.
    % If the positions are computed before this
        function is called, set r=f
    % in order to save computation time.
    %r = gradientDescent(f,0.15,0.86,0,0);
    r=f;
    % Find the headings at each position
    heading = [heading_init atan(diff(r(2,:))./diff(r
        (1,:))) + pi*double(diff(r(1,:))<0)];
```

```

    for k=2:length(heading)
        % Compute the change in heading from one position to the next
        change_heading = heading(k) - heading(k-1);
        % Find the left and right wheel velocities in order to turn the
        % NEATO
        Vl(k) = V - omega * (d/2);
        Vr(k) = V + omega * (d/2);
        % Function to turn the NEATO to a new heading
        turnNeato(Vl(k),Vr(k),change_heading/omega)
        % Pause for data collection
        pause(0.01)
        % driveforward was a scipt distrubuted in the beginning of class
        %driveforward(norm(r(:,k)-r(:,k-1))*0.3048,0.1) % Convert from feet to meters by using 0.3048
        driveforward(norm(r(:,k)-r(:,k-1)),0.1)
        % Pause for data collection
        pause(0.01)
    end
end
```

7.5 Turn the NEATO Script

```
function turnNeato(Vl,Vr,time)
    % turnNeato is a function that will publish wheel speeds to turn the
    % NEATO for a given amount of time

    % Initialize ROS to publish wheel velocities to the NEATO
    pubvel = rospublisher('/raw_vel')

    % Create the message to send over ROS
    message = rosmessage(pubvel);
```

```
    % Use tic and toc for the timing of the turn
    tic

    % Set the right and left wheel velocities
    message.Data = [Vl, Vr];

    % Publish the left and right wheel velocities to the NEATO
    send(pubvel, message);
    while 1
        pause(0.01)
        if toc > time %  If the time since the start is greater than the time for the turn

            message.Data = [0,0]; % Set wheel velocities to zero if we have turned for long enough
            send(pubvel, message); % Send new wheel velocities
            break % Leave this loop once we have reached the stopping time
        end
    end
end
```

7.6 RANSAC Algorithm

```
function [min_max,P,best_A,best_B,best_percent,outside] = RANSAC(global_x,global_y,trials,threshold)
    % Function that implements the RANSAC algorithm described in Day 6

    % Concatenate the global x and y coordinates
    P = [global_x,global_y];
    P = P';

    % Initialize the best A point, B point, and percent
```

```
    best_A = [0,0];
    best_B = [0,0];
    best_percent = 0;

    % Initialize the loop interation
    loop_iteration = 1;
    while loop_iteration < trials
        m = 1;
        z = 1;
        % Increment the loop iteration
        loop_iteration = loop_iteration + 1;
        % Find two random indicies for the points
        point_index = randi([1, length(P)],2,1);

        % Define point 'A' and 'B'
        A = P(:,point_index(1));
        B = P(:,point_index(2));

        % Define the vectors used to compute the distance between a point
        % and line
        That = [B(1) - A(1);B(2) - A(2);0]./vecnorm([B(1) - A(1);B(2) - A(2);0]);
        Khat = [0;0;1];
        Nhat = cross(That,Khat);

        % Initialize a counter for points that are in
        in = zeros(1,length(P));
        for a = 1:length(P)
            % Define a r vector between any (x,y) coordinate point in the
            % matrix of all points and point A
            R = [P(1,a)-A(1,1);P(2,a)-A(2,1);0];
            % Compute the distance between the line and any given point
            distance = abs(dot(Nhat,R));
            % Check if the distance is within the threshold
```

```
            if distance < d
                in(a) = 1;
                inside(1,z) = P(1,a);
                inside(2,z) = P(2,a);
                z = z + 1;
            else
                outside(1,m) = P(1,a);
                outside(2,m) = P(2,a);
                m = m + 1;
            end
        end

        % Compute the percentage of the points that were within the bounds
        percent_in = mean(in);

        % If this was the best percentage, save all of the values and
        % define the bounds for the line
        if percent_in > best_percent
            insidex = inside(1,:);
            insidey = inside(2,:);
            best_A = A;
            best_B = B;
            best_percent = percent_in;
            x_max = max(insidex);
            x_min = min(insidex);
            y_max = max(insidey);
            y_min = min(insidey);
            min_max = [x_min,x_max;y_min,y_max];
        end

        % If this percentage was above the threshold percent, save all of
        % the values, define the bounds, and end the loop
        if percent_in > threshold
            best_A = A;
```

```
                best_B = B;
                best_percent = percent_in;
                x_max = max(inside(1,:));
                x_min = min(inside(1,:));
                y_max = max(inside(2,:));
                y_min = min(inside(2,:));
                min_max = [x_min,x_max;y_min,y_max];
                break
            end
        end
end
```