MTE 544 - Autonomous Mobile Robotics

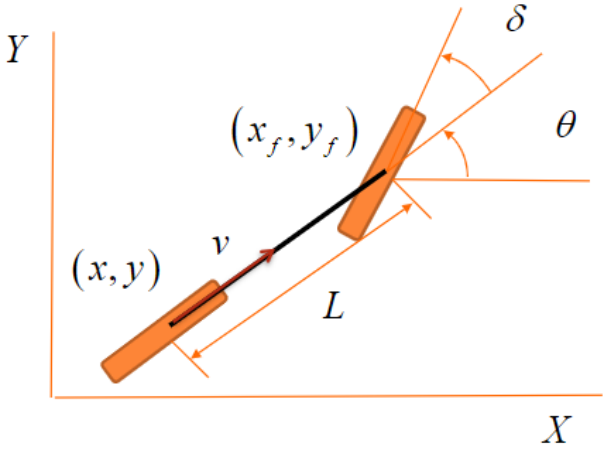
Assignment 2

By: Lalit Lal (20572296), Michal Kaca (20564560)
Date: 29/11/2018

1)

The following motion model is associated with the Ackermann bicycle model. This can be directly taken out of the lecture slides for verification. The bicycle model is one of the simplest 2-wheel models in the world. Essentially, the x position updates itself by taking the previous x position and adding it to the distance traveled in the previous time step (taking time, velocity, and previous orientation as the determining factors). The same is done for the y position, with the exception of using sine instead of cosine (high school geometry). The orientation is calculated by taking the previous orientation and adding it to the change in orientation calculated between the current and previous steps by taking into account the distance between the centers of the two wheels, the linear velocity, steering angle, and change in time. See the complete equation below.

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} + v_t \cos\theta_{t-1} dt \\ y_{t-1} + v_t \sin\theta_{t-1} dt \\ \theta_{t-1} + \frac{v_t \tan\delta_t}{L} dt \end{bmatrix}$$



R indicates the error function with the associated additive Gaussian disturbance of deviation 0.02m for both the y and x coordinates, and 1 degree for the orientation.
The inputs are: velocity (v), and steering angle (δ).

$$\boldsymbol{E_R} = \begin{matrix} \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} \end{matrix}, \quad \boldsymbol{\lambda_R} = \begin{matrix} \mathbf{0.0003046} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0.0004} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0.0004} \end{matrix}, \quad \boldsymbol{R} = \begin{matrix} \mathbf{0.02^2} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0.02^2} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{(\frac{3.1415}{180})^2} \end{matrix}$$

The randomized Gaussian noise (error) matrix ($\boldsymbol{\varepsilon}$) can be calculated with the equation below:

$$\boldsymbol{\varepsilon} = \boldsymbol{E_R} * \boldsymbol{\lambda_R}^{0.5} * \boldsymbol{randn(n, 1)}$$

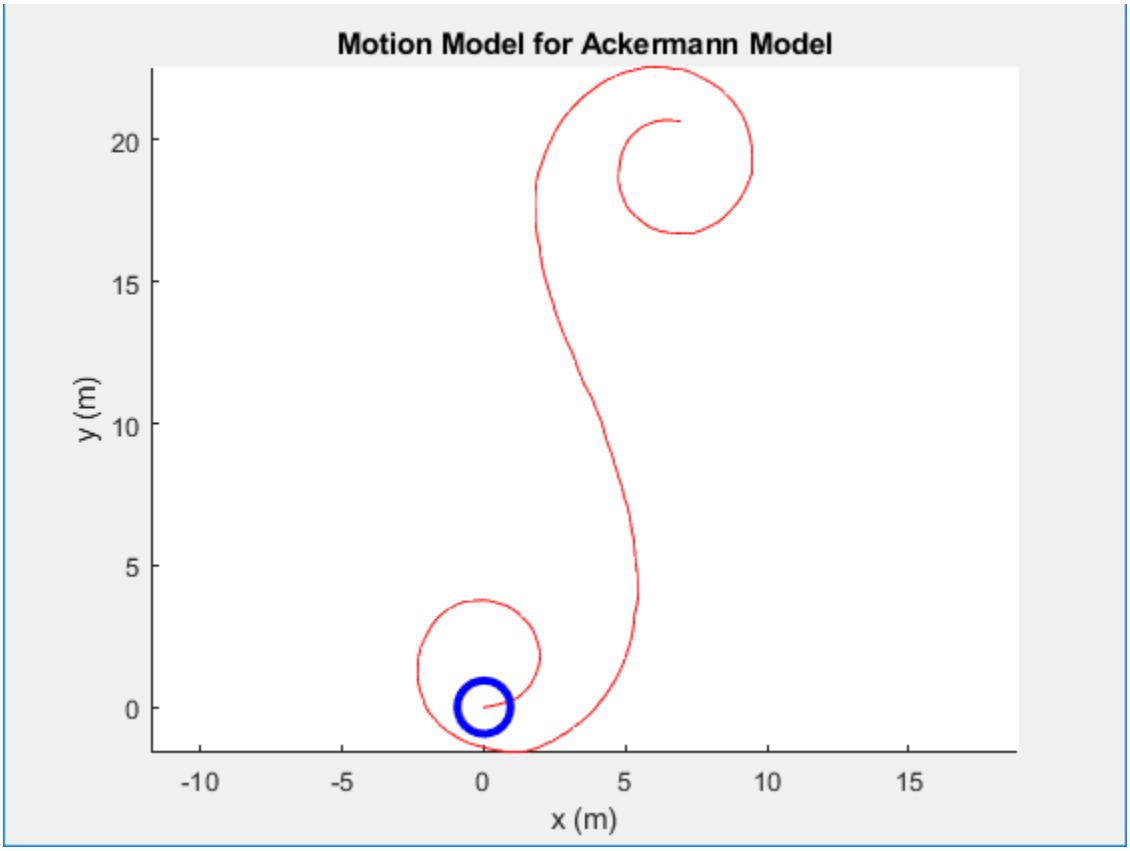
where n = 3

Upon updating the position at each time step, the error matrix is simply added to the motion model equation to add noise to the model.

The steering angle limitation is specified to be between +30 and -30 degrees, hence a saturated block can be placed in the code. See the snippet below.

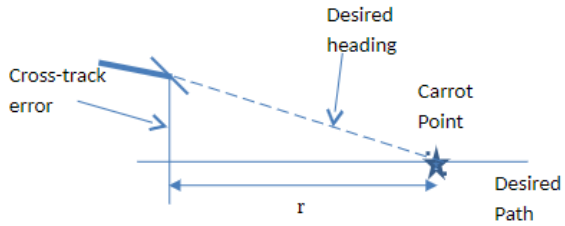
```
steerAngle = 10 - i/10; % input control in degrees
if steerAngle > 30
    steerAngle = 30;
elseif steerAngle < -30
        steerAngle = -30;
end
```

The parameters used for the simulated motion model are:
L = 0.30 m
v = 3.0 m/s
Simulation time = 20 seconds
Steering angle = δ = (10 - t) degrees, where t is in seconds. This indicates that δ will begin at 10 degrees, and end at -10 degrees.



As seen by the plot above, the motion model behaves exactly as expected: Halfway through the time step, the steering angle transitions from negative (counterclockwise) to a positive (clockwise) steering direction. The final position of the robot is at approximately (x, y) = (5, 20).

2)



The carrot (follower) algorithm is a method used for following a given path. The discretized given path can be seen as a vector of (x, y) points that the robot is to follow (as accurately as possible). The lookahead distance, r, is set by the user which determines how far ahead (in the vector of goal points) will the robot look before choosing a sole target (carrot point). The steering angle is set with the equation below:

$\delta = C*Eo$

Where C is the controller for the corrected steering angle, and Eo is the angle between the current robot orientation and the vector pointing from the robot to the target points (aka the desired path).

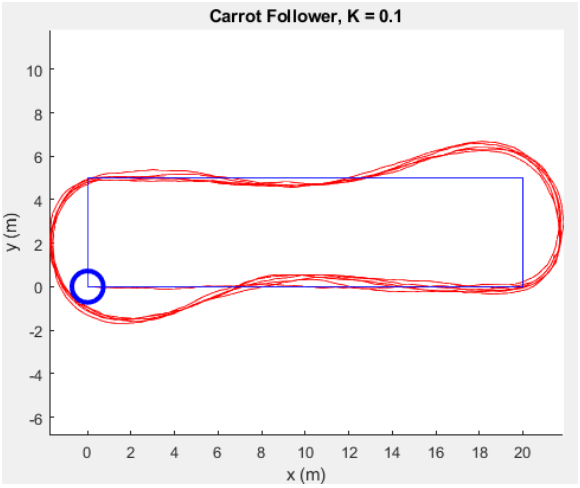
In other words, δ sets the robot in the path of the desired heading.

The process is repeated until the entire path has been traversed.

The group used a proportional controller with a gain of 1 for best results, thus setting:

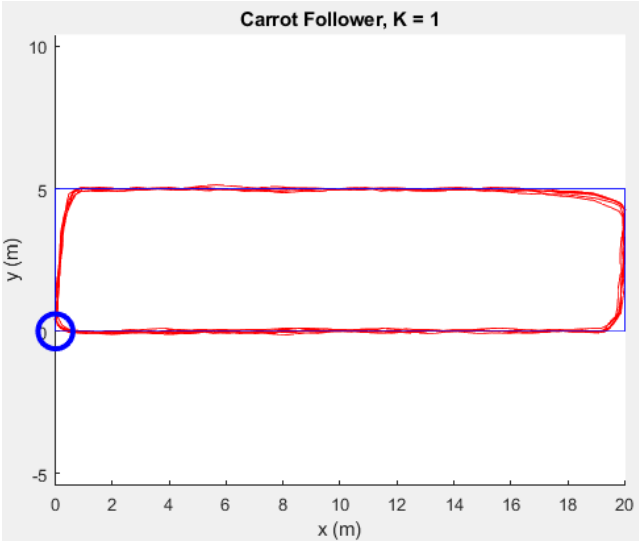
$\delta = Kp*Eo = Eo$

See the plots below for demonstrating the effect of Kp for Kp = 0.1 and Kp = 10.



A small Kp causes the robot to miss its carrot point (since it doesn't turn sharply enough), and a large Kp causes fluctuations in the path. One use for a lower Kp is to prevent cutting corners, and in contrast, a use for a higher Kp is to more decrease the standard deviation from the path.

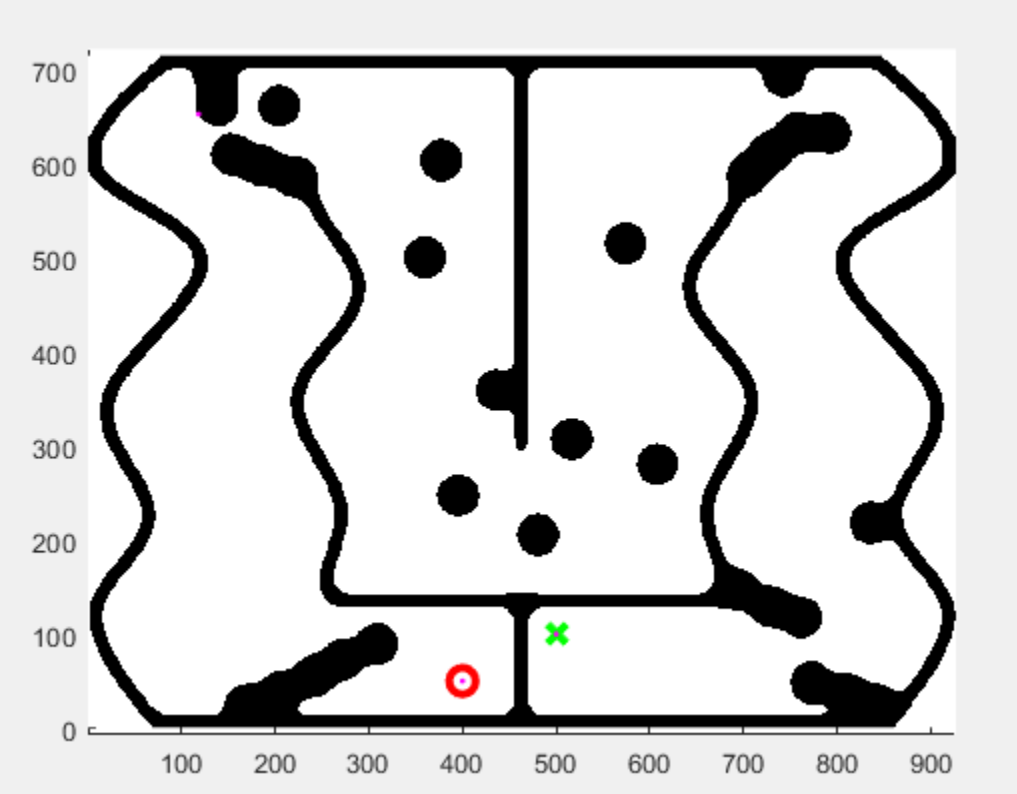
For the deal simulation, the robot tracks a 20 by 5 meter rectangle (vertices: [0,0 0,20 20,5 5,0], with its starting point and orientation pose being (x, y, theta) = [0 0 0]. The constant velocity of the robot is 3 m/s. The robot's global direction of travel is counterclockwise.



The selected value for r was simply the next index. In the case that the next point was in the reverse direction, the selected value for r became the previous index, thus reversing the loop. The group used the approach of setting r to an index lookahead point, rather than a distance lookahead point.

It can be seen in the plot above that the robot cuts corners in order to prevent overshooting. This is done using an angle saturator that prevents the robot from having a steering angle being greater than 30 degrees. Additionally, an angle wrapper is used that prevents Eo from leaving the bounds of -3.14 and 3.14 radians. This is an excellent algorithm that is quick and easy to implement. However, limitations for this exist, and are discussed later on in the report.

3) and 4)



The given map (see above) is the IGVC auto-navigation course. The goal of this question is for our defined robot to traverse the given map using a planning strategy (to set the path), and a carrot planner (to follow the set path) with the conjunction of our initially defined motion model.

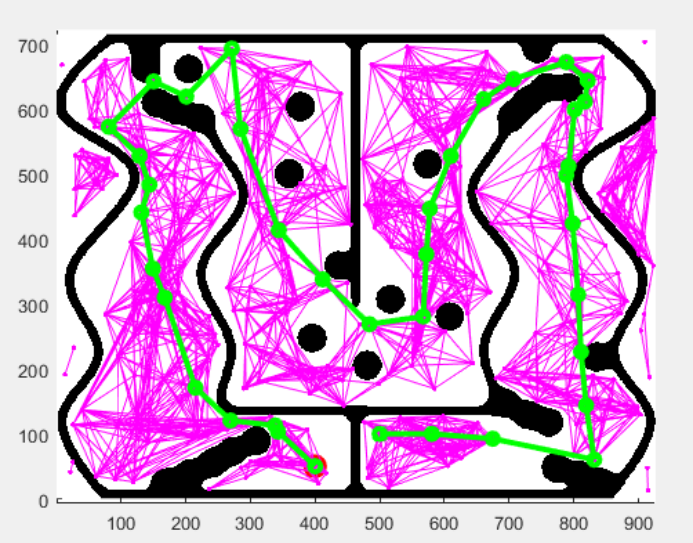
The path planner used by the group is the probabilistic roadmap (PRM) algorithm. The logic overview of the group's implemented planner is as follows:

1. Set initial and final points; add them to milestones array
2. Create random set of points (within map boundaries)
3. For each new point, check if newly created point is in free space; aka if it is a point that can be occupied by the robot
    a. If so, add to milestones array
    b. Else, ignore
4. Connect all milestones with a vector
5. Using the Bresenham algorithm, if the vector passes through an occupied space, discard connection, otherwise create edge (connection)
6. Check if a shortest path can be found between start and end point
    a. If so, draw path, collect points, and end algorithm
    b. Else, add more random points, and repeat loop

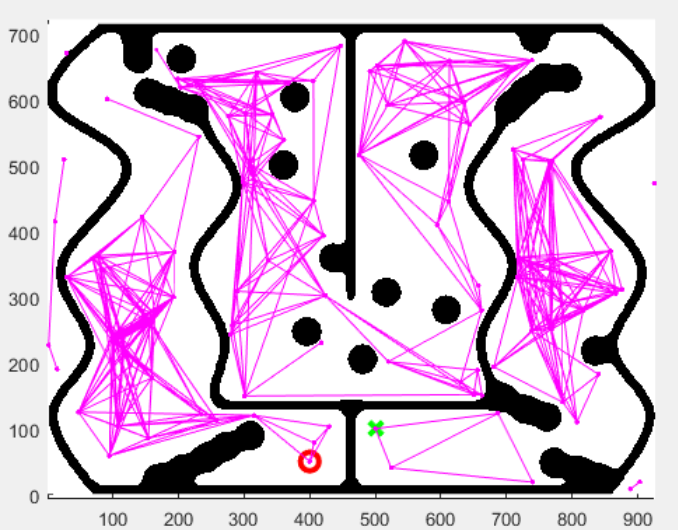
The group used PRM because of its certainty of finding a path (assuming one exists and given enough time to run the algorithm), as well as its ease of application.

For the simulation, the only constraints are for the robot to start at location [40 5 3.14159], and to end at location [50 10 X], where X is any orientation. The start and end

point are shown in the simulation plot with a red circle and a green x respectively. It can be seen by the image below that a path was successfully found (green points with line connecting them). It is important to note that although this is the short path found, it is not necessarily the most efficient path found. In other words, if the algorithm finds a complete path from point A (start) to point B (end), then it will terminate the algorithm. An approach of allowing the algorithm to run for an additional N cycles could be used in order to increase the probability of finding a more efficient complete path. However, even this workaround does not guarantee that the optimal path will be chosen.

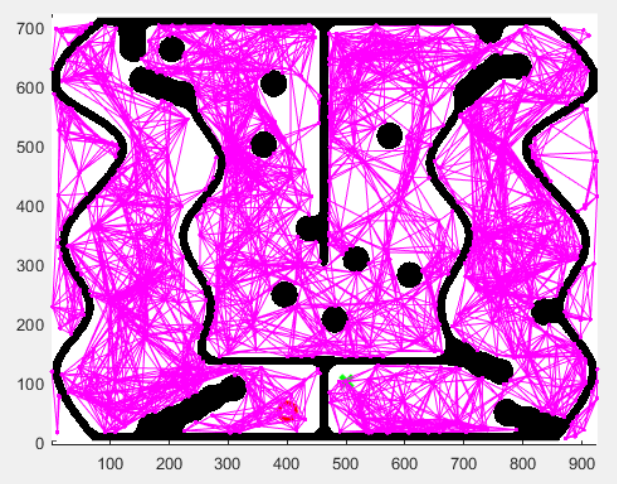


The algorithm can be seen in progress in the image below. Several milestones have been created and edges have been created. However, many sections of the map still need to be connected, hence many more random sample points are necessary.

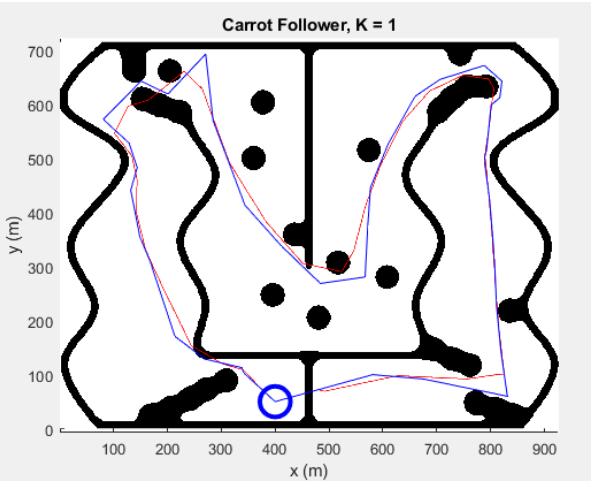


In cases where the algorithm does not choose favorable points, it may take longer than anticipated to run, thus not generating a path within a given time constraint. For example, in the image below, the algorithm ran out of time before successfully completing the path from the

start to the finish point. This time constraint is one of the limitations of the PRM algorithm in the real world, where time is almost always a constraint.



Applying the carrot follower onto the calculated PRM, one obtains the image below, where the blue lines represent the connected PRM target points, and the red is the actual robot path. An important limitation is observed in the plots below. By overlaying the map as the background of the plot, one can see that the robot actually hits some of the obstacles where the space is extremely narrow. This can be avoided in the future by rewriting the carrot algorithm so that the lookahead distance is increased and by applying a conditional statement to the PRM that algorithm that waits for enough points to be present in the shortest connected path before ending the program. This observable problem is the most significant issue when using the carrot path following algorithm in tight spaces.



Some assumptions that have been made for navigating the robot through the IGVC auto-navigation course include:

1. The path is navigable by a robot of a given footprint (outer dimensions). In other words, the robot can physically get from point A (start) to point B (end) without altering the map or the robots physical structure.
2. The given map is accurate and fixed for the duration of the test. If the map was to change at any point, then the PRM algorithm would need to be rerun and the carrot points would need to be changed accordingly.
3. There are no moving objects in the navigation course, other than the robot itself. If a moving object was incorporated into the course, then a sensing system would need to be applied with a minimalistic version of object detection (and planning a path around the moving obstacle).
4. The noise is minimal. This can be mitigated by using more reliable PD/PID controllers to assuage the cumulative erroneous noise.

Appendix:

Appendix A: Q1 MATLAB code

```
clc
close all
clear all

l = 0.3; % m
v = 3.0; % m/s
dt = 0.1; % s
x = [0.0 0.0 0.0]'; %init state
R = diag([0.02^2 0.02^2 (pi/180)^2]);
[RE, Re] = eig(R);

% angle limit = +- 0.523599 radians

n=200; % # of samples ...20 seconds at 0.1s freq
for i=2:n
    % Disturbance
    E = RE*sqrt(Re)*randn(3,1);
    % Dynamics
    steerAngle = 10 - i/10; % input control in degrees
    if steerAngle > 30
        steerAngle = 30;
    elseif steerAngle < -30
            steerAngle = -30;
    end
    x(:,i) = x(:,i-1) + [dt*(v)*cos(x(3,i-1)) ; dt*(v)*sin(x(3,i-1))  ;
dt*(1/l)*(v)*tand(steerAngle)] + E;
   end

figure(1); clf; hold on;
plot(x(1,:), x(2,:), 'Color', 'r');
plot(x(1), x(2), 'bo', 'MarkerSize',20, 'LineWidth', 3)
%plot(x(1,:), x(2,:), 'Color', 'r');
title('Motion Model for Ackermann Model')
xlabel('x (m)');
ylabel('y (m)');
axis equal
```

Appendix B: Q2 MATLAB code (modified to follow PRM path)

```
clc
close all
clear all
%% Ackerman Bicycle Motion Model
r = 0.25;
l = 0.3;

L = 0.3;
MAX_ANGLE = 30*pi/180;

dt = 0.1; % Timestep

% init states for Q3
x = [400 50 2]';
x1 = [400 50 2]';

% init states for Q2
% x = [ 0.0 0.0 0.0]';
% x1 = [ 0.0 0.0 0.0]';

R = diag([0.02^2 0.02^2 (1*pi/180)^2]);
[RE, Re] = eig(R);


vt = 3; %3m/s
n=200; % Samples

% Q2 rectangle discretized path
%points =
[[0,0];[2,0];[4,0];[6,0];[8,0];[10,0];[12,0];[14,0];[16,0];[18,0];[20,0];[20,
1];[20,3];[20,5];[18,5];[16,5];
%[14,5];[12,5];[10,5];[8,5];[6,5];[4,5];[2,5];[0,5];[0,3];[0,1]]

% PRM path
points =
[[400,50];[342,103];[338,113];[269,129];[214,171];[168,310];[149,355];
[132,442];[144,484];[129,529];[81,573];[151,644];[201,620];[271,694];[285,570
];
[344,414];[412,338];[484,269];[567,281];[572,377];[577,447];[610,528];[660,61
6];
[706,647];[788,673];[821,644];[817,613];[801,600];[792,513];[788,499];[798,42
5];
[807,315];[811,226];[819,144];[831,60];[675,92];[581,100];]

%points =
[[0,0];[5,0];[10,0];[15,0];[20,0];[20,5];[15,5];[10,5];[5,5];[0,5]];
%points = [[0,0];[20,0];[20,5];[0,5]]

%while loop control variables
endLoop = 0;
halfWayCheck = 0;
%conservative thresholding
```

```
upperXLimit = 22;
upperYLimit = 6;
lowerXLimit = 18;
lowerYLimit = 4;

state = 0;
tolerance = 0.5;
%steer_ang = 0;
route_done = 0;
lookup = 1;
i = 2;
x_next = 50; % init random shit
init_dir = 'forward';
init_loop = 1;
kp = 1;
while route_done == 0
    %Update positions x, y of robot
    x(1,i) = x(1,i-1) + vt * cos(x(3,i-1)) * dt;
    x(2,i) = x(2,i-1) + vt * sin(x(3,i-1)) * dt;

    %determine which point on path is closest
    distances = sqrt((points(:,1) - x(1,i)).^2 + (points(:,2) - x(2,i)).^2);
    [M,I] = min(distances);

%This part should check if we need to keep increasing out step, but don't
worry
%        if M < 0
%            I = I + lookup;
%            if I > length(points)
%                I = 1;
%            elseif I < 1
%                I = length(points);
%            end
%            M = distances(I);
%        end

    %get index of new carrot points
    next_ind = I + lookup;
    prev_ind = I - lookup;
    %min_dist_x = points(I,1);
    %min_dist_y = points(I,2);

    % DESIGN POINT: CHOOSE PREVIOUS STATE OF CURRENT (i)
    x0 = x(1,i-1);
    y0 = x(2,i-1);
    theta0 = x(3,i-1);

    % loop around path points
    if (next_ind > length(points))
        next_ind = 1;
    end
    if prev_ind < 1
        prev_ind = length(points);
    end
```

```
    % determine x,y values of lookahead points (both ahead and before)
    % before not used
    x_next = points(next_ind,1);
    y_next = points(next_ind,2);

    x_prev = points(prev_ind,1);
    y_prev = points(prev_ind,2);

    % when first moving, check direction of path you're going (fwd or bwd)
    if init_loop == 1

        if abs(atan2(y_next-y0,x_next-x0)-theta0) < abs(atan2(y_prev-
y0,x_prev-x0)-theta0)
            init_dir = 'forward';
        else
            init_dir = 'backward';
        end
        init_loop = 0;

    end

    %depending on if you were initially moving fwd of bwd, moving in that
    %direction along path
     if strcmp(init_dir, 'forward')
         theta_err = atan2(y_next-y0,x_next-x0)-theta0;
         %theta_err = atan((y_next-y0)/(x_next-x0));
     else
         theta_err = atan2(y_prev-y0,x_prev-x0)-theta0;
         %theta_err = atan((y_prev-y0)/(x_prev-x0));
     end

    if theta_err > pi || theta0 > pi || theta_err < -pi

        %disp(theta0);
        theta_err = (wrapToPi(theta_err));
    end

    %P control
    %theta_err = min(abs(atan2(y_next-y0,x_next-x0)-theta0),
abs(atan2(y_prev-y0,x_prev-x0)-theta0));
    theta_err = theta_err * kp;
    steer_ang = theta_err;

    %angle saturation due to bike angle
    if steer_ang < -MAX_ANGLE
        steer_ang = -MAX_ANGLE;
    elseif steer_ang > MAX_ANGLE
        steer_ang = MAX_ANGLE;
    end


    %update heading of robot with new steering angle
    x(3,i) = (x(3,i-1) + vt * tan(steer_ang)/L * dt);
```

```
    %plot lookahead points (should be rectangle)
    path(1,i) = points(next_ind, 1);
    path(2,i) = points(next_ind,2);

    %guassian noise
    x(:,i) = x(:,i) + RE*sqrt(Re)*randn(3,1);
    i = i + 1;
    if(i == 9005)
        disp(i);
        route_done=1;
    end
end

%read map and store it into occ grid (nodes)
I = imread('IGVCmap.jpg');
I = imgaussfilt(I,9);
%I = im2bw(I,0.0001);
map = im2bw(I, 0.7); % Convert to 0-1 image
map = 1-flipud(map)'; % Convert to 0 free, 1 occupied and flip.

% Plot
figure(1); clf; hold on;
colormap('gray'); % added colormap
imagesc(1-map');  % added colormap
plot(x(1,:), x(2,:), 'Color', 'r');
plot(path(1,2:end),path(2,2:end), 'Color', 'b');  % Removed first param for
0,0
%plot (x1(1,:), x1(2,:), 'Color', 'b');
plot( x(1,1), x(2,1), 'bo', 'MarkerSize',20, 'LineWidth', 3)
%plot( x1(1), x1(2), 'bo', 'MarkerSize',20, 'LineWidth', 3)
%plot( [x0(1) x1(1)], [x0(2) x1(2)],'b')
%plot( x(1,:),x(2,:), 'm.', 'MarkerSize', 3)
title('Carrot Follower, K = 1')
xlabel('x (m)');
ylabel('y (m)');
axis equal
hold on
%rectangle('Position',[0,0, 20,5],'Curvature',[0,0],...
         %'LineWidth',2,'LineStyle','--');
```

Appendix C: Q3 MATLAB code

```
%% Probabilistic Road Map example
clear all;
clc;
close all;


%% Problem parameters
tic;

%time limit
timeLimit = 900;

%read map and store it into occ grid (nodes)
I = imread('IGVCmap.jpg');
I = imgaussfilt(I,9);
%I = im2bw(I,0.0001);
map = im2bw(I, 0.7); % Convert to 0-1 image
map = 1-flipud(map)'; % Convert to 0 free, 1 occupied and flip.
[M,N]= size(map); % Map size


% Robot start position
dxy = 0.1;
startpos = [40/dxy 5/dxy];

% Target location
searchgoal = [50/dxy 10/dxy];

% Set up the map
xMax = [M N]; % State bounds
xMin = [0 0];
xR = xMax-xMin;

% Set up the goals
x0 = startpos;
xF = searchgoal;

%  % Set up the obstacles
% rand('state', 1);
% nO = 30; % number of obstacles
% nE = 4; % number of edges per obstacle (not changeable).
% minLen.a = 1; % Obstacle size bounds
% maxLen.a = 4;
% minLen.b = 1;
% maxLen.b = 6;

obstBuffer = 1; % Buffer space around obstacles
maxCount = 1000; % Iterations to search for obstacle locations

env = map;

%% Multi-query PRM, created until solution found
```

```
tic;
done = 0;
milestones = [x0; xF];
e = zeros(2,2);
% Number of edges to add
p = 12;
t = 0;
tm = 0;
te = 0;
ts = 0;
ec = 0
figure(1); clf; hold on;
colormap('gray');
imagesc(1-map');
plot(startpos(1), startpos(2), 'ro', 'MarkerSize',10, 'LineWidth', 3);
plot(searchgoal(1), searchgoal(2), 'gx', 'MarkerSize',10, 'LineWidth', 3 );
axis equal
hold on;

while ((~done) && (t < timeLimit))
    %disp('Enter Loop');
    t=t+1;
    % Get new milestone
    newstone = 0;
    t0 = cputime;
    while (~newstone)
        %disp('Getting milestone');

        if(length(milestones) < 400) %sample all first
            sample = [(xR(1)-1)*rand(1,1)+xMin(1)+1 (xR(2)-
1)*rand(1,1)+xMin(2)+1];
            keep = map(int16(sample(1)),int16(sample(2)))
        else
            sample_q = [randsample(M,1) randsample(N,1)];
            while(sample_q(1) > M || sample_q(2) > N)
                sample_q = [randsample(M,1) randsample(N,1)];
            end

            sample_qnot = [2.*randn(1,1)+sample_q(1)
2.*randn(1,1)+sample_q(2)];
            while((sample_qnot(1) > M || sample_qnot(2) > N) ||
sample_qnot(1)<1 || sample_qnot(2)<1)
                sample_qnot = [2.*randn(1,1)+sample_q(1)
2.*randn(1,1)+sample_q(2)];
            end
            if (map(int16(sample_q(1)),int16(sample_q(2))) &&
~map(int16(sample_qnot(1)),int16(sample_qnot(2))))
                keep = 0;
                sample = sample_qnot;
            elseif (map(int16(sample_qnot(1)),int16(sample_qnot(2))) &&
~map(int16(sample_q(1)),int16(sample_q(2))))
                keep = 0;
                sample = sample_q;
            else
                keep = 1;
            end
```

```
        end

        if (~keep)
            %disp('Got milestone');
            milestones = [milestones; sample];
            newstone = 1;
            figure(1); hold on;
            plot(milestones(:,1),milestones(:,2),'m.');
        end
    end
    %disp('Got Milestone');
    t1 = cputime;
    tm = tm+t1-t0;
    % Attempt to add closest p edges
    t2 = cputime;
    cur = length(milestones(:,1));
    for i = 1:cur-1
        d(i) = norm(milestones(cur,:)-milestones(i,:));
    end
    [d2,ind] = sort(d); %stores sorted array with their index order (closest
edges)
    % Check for edge collisions (no need to check if entire edge is
    % contained in obstacles as both endpoints are in free space)
    for j=1:min(p,length(ind))
        ec = ec + 1;
        x1 = milestones(cur,1);
        y1 = milestones(cur,2);
        x2 = milestones(ind(j),1);
        y2 = milestones(ind(j),2);
        points_2 = [x1,y1;x2,y2];
        [vec_x vec_y] = bresenham(x1,y1,x2,y2);
        ray_points = [vec_x vec_y];
        collision = false;
        for k=1:length(vec_x)
            if map(vec_x(k),vec_y(k))
                collision = true;
            end
        end

        %if (~CheckCollision(milestones(cur,:),milestones(ind(j),:),
obsEdges))
        %(pdist(points_2,'euclidean') < 500) // this checks if points
        %within certain distance
        if (~collision)
            %disp('NO COLLISION');
            e(ind(j),cur) = 1;
            e(cur,ind(j)) = 1;
            plot([milestones(ind(j),1)
milestones(cur,1)],[milestones(ind(j),2) milestones(cur,2)],'m');
        else
            %disp('Collision');
            e(ind(j),cur) = 0;
            e(cur,ind(j)) = 0;
        end
    end
```

```
    t3 = cputime;
    te = te + t3 - t2;

    % Check if a path from start to end is found
    t4 = cputime;
    [sp, sd] = shortestpath(milestones, e, 1, 2);
    path_points = [];
    if (sd>0 && length(sd) > 45)
        done = 1;
        for i=1:length(sp)-1
        plot(milestones(sp(i:i+1),1),milestones(sp(i:i+1),2), 'go-',
'LineWidth',3);
        path_points(i:i+1,:) = [milestones(sp(i:i+1),1)
milestones(sp(i:i+1),2)]
        end
    end
    t5 = cputime;
    ts = ts + t5 - t4;
end
tm
te
ts
ec

% figure(1);clf;hold on;
% axis equal
disp('Time to find shortest path');
toc;

if(length(sp) ~= 0)
    test_xx = M544_A2(path_points);
else
    disp('no path');
end
```