Homework 2

Andy Cheng & Holly Schlichting

1) Moving the tool center point from point A to point B while avoiding body collisions is the goal of industrial robot trajectory planning. Motion planning and, erroneously, path planning are two terms used to describe trajectory planning. The difference between trajectory and path planning is that trajectory planning is time-parameterized. The term "trajectory planning" refers to the process of determining how to move depending on velocity, time, and kinematics.

1.1 Dividing 2 seconds into centi-seconds (cs):

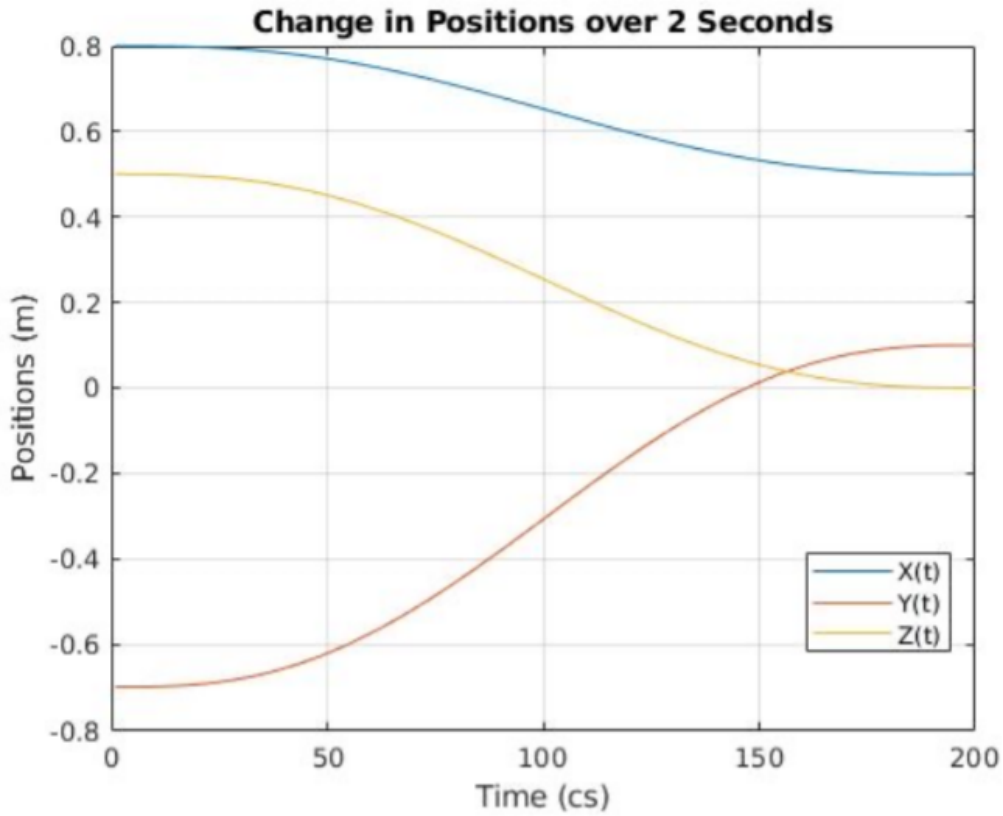
```
>> tt = 0:0.01:2

tt =

  Columns 1 through 9

         0    0.0100    0.0200    0.0300    0.0400    0.0500    0.0600    0.0700    0.0800

  Columns 10 through 18

    0.0900    0.1000    0.1100    0.1200    0.1300    0.1400    0.1500    0.1600    0.1700

  Columns 19 through 27

    0.1800    0.1900    0.2000    0.2100    0.2200    0.2300    0.2400    0.2500    0.2600

  Columns 28 through 36
```

Figure: 1.1

Now we can graph the X(t), Y(t), and Z(t) motion with the tpoly planners over 2 seconds using the “mtraj” function:

```
>> s = mtraj(@tpoly, [0.8000 -0.7000 0.5000], [0.5000 0.1000 0], tt)

s =

    0.8000   -0.7000    0.5000
    0.8000   -0.7000    0.5000
    0.8000   -0.7000    0.5000
    0.8000   -0.7000    0.5000
    0.8000   -0.6999    0.5000
    0.8000   -0.6999    0.4999
    0.7999   -0.6998    0.4999
    0.7999   -0.6997    0.4998
    0.7998   -0.6995    0.4997
    0.7997   -0.6993    0.4996
    0.7997   -0.6991    0.4994
    0.7995   -0.6988    0.4992
    0.7994   -0.6984    0.4990
    0.7993   -0.6980    0.4988
    0.7991   -0.6975    0.4985
    0.7989   -0.6970    0.4981
    0.7986   -0.6964    0.4977
    0.7984   -0.6957    0.4973
    0.7981   -0.6949    0.4968
    0.7978   -0.6941    0.4963
    0.7974   -0.6932    0.4957
    0.7971   -0.6921    0.4951
    0.7966   -0.6910    0.4944
    0.7962   -0.6898    0.4936
    0.7957   -0.6885    0.4928
    0.7952   -0.6872    0.4920
    0.7946   -0.6857    0.4910
    0.7940   -0.6841    0.4901
    0.7934   -0.6824    0.4890
    0.7927   -0.6806    0.4879
```

The graph obtained is shown here:



Since only the homogeneous transformation matrices are given, the roll, pitch and yaw angles are to be determined using the same matrices but without the displacement part:

```
>> Arpy = [0.9256 -0.2427 0.2904 0; 0.2863 0.9508 -0.1181 0; -0.2474 0.1925 0.9496 0; 0 0 0 1]

Arpy =

    0.9256   -0.2427    0.2904         0
    0.2863    0.9508   -0.1181         0
   -0.2474    0.1925    0.9496         0
```

```
>> Brpy = [0.9890 -0.0998 0.1092 0; 0.0992 0.9950 0.0110 0; -0.1098 0 0.9940 0; 0 0 0 1]

Brpy =

    0.9890   -0.0998    0.1092         0
    0.0992    0.9950    0.0110         0
   -0.1098         0    0.9940         0
         0         0         0    1.0000
```

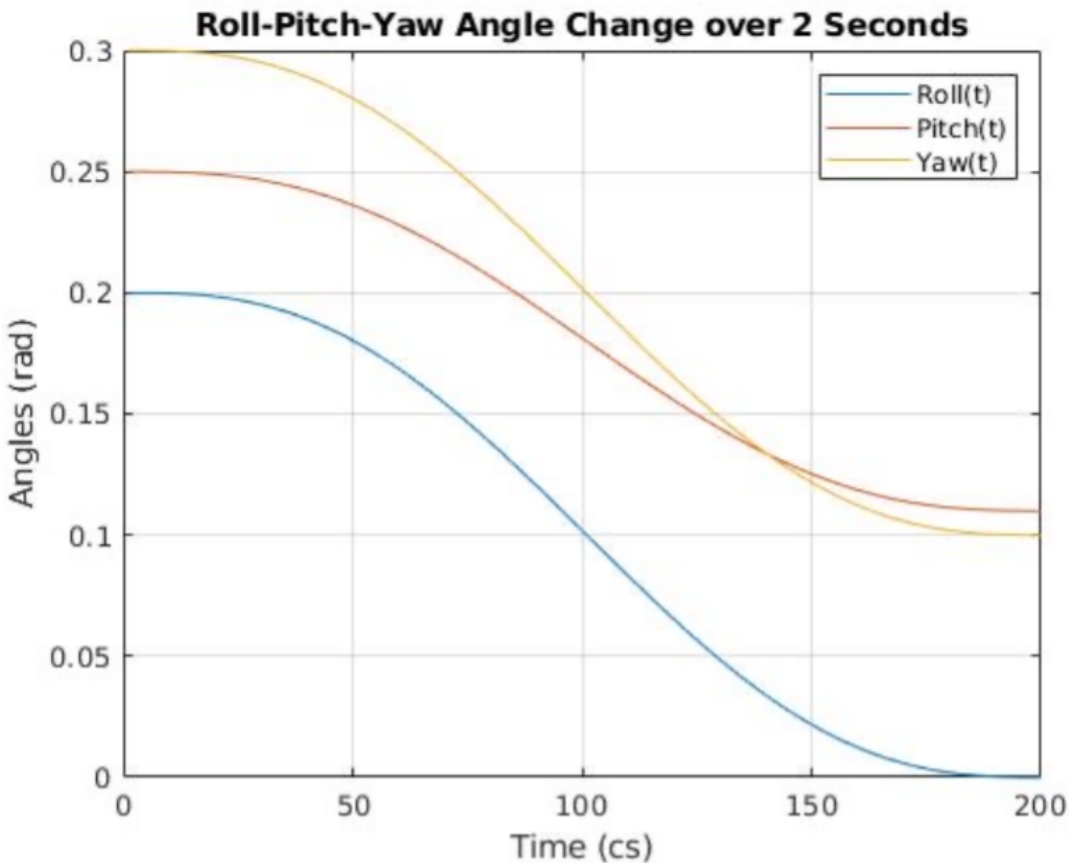
The RTB comes with the transformation to roll-pitch-yaw angles converter, so the matrices here can be converted to their respective roll-pitch-yaw angles:

```
>> ARPY = tr2rpy(Arpy)

ARPY =

    0.2000    0.2500    0.3000
>> BRPY = tr2rpy(Brpy)

BRPY =

         0    0.1100    0.1000
```

Now with the RPY angles from both frames, we can get the trajectories with the same function:

```
>> rpy = mtraj(@tpoly, ARPY, BRPY, tt)

rpy =

    0.2000    0.2500    0.3000
    0.2000    0.2500    0.3000
    0.2000    0.2500    0.3000
    0.2000    0.2500    0.3000
    0.2000    0.2500    0.3000
    0.2000    0.2500    0.2999
    0.2000    0.2500    0.2999
    0.1999    0.2499    0.2999
    0.1999    0.2499    0.2999
    0.1998    0.2499    0.2998
    0.1998    0.2498    0.2997
    0.1997    0.2498    0.2997
    0.1996    0.2497    0.2996
    0.1995    0.2496    0.2995
    0.1994    0.2496    0.2994
    0.1993    0.2495    0.2992
    0.1991    0.2494    0.2991
```

Here are the roll-pitch-yaw trajectories over the course of two seconds:



1.2

In this problem the goal is to plot and plan the mobile robot in world coordinate frames from a stopping point to various different points. Since only the maximum linear speed and acceleration time were given, we first list all the waypoints for x and y:

A = (0,0)

B = (2, 4)

C = (7, 3)

D = (10, 3.5)

The maximum linear velocity is 10 yards/s, and the linear acceleration time from 0 to max speed is 2 seconds.

Then we list all the waypoints for theta:

A = 0

B = pi/4

C = pi/2

D = pi

Since the maximum angular velocity and acceleration time were not given, we have to make assumptions about those so that we can use the mstraj function. Therefore, the maximum angular velocity is assumed to be pi/8, and the angular acceleration time is assumed to be 1 second. With the above assumptions and waypoints, we're now able to implement them in Matlab:

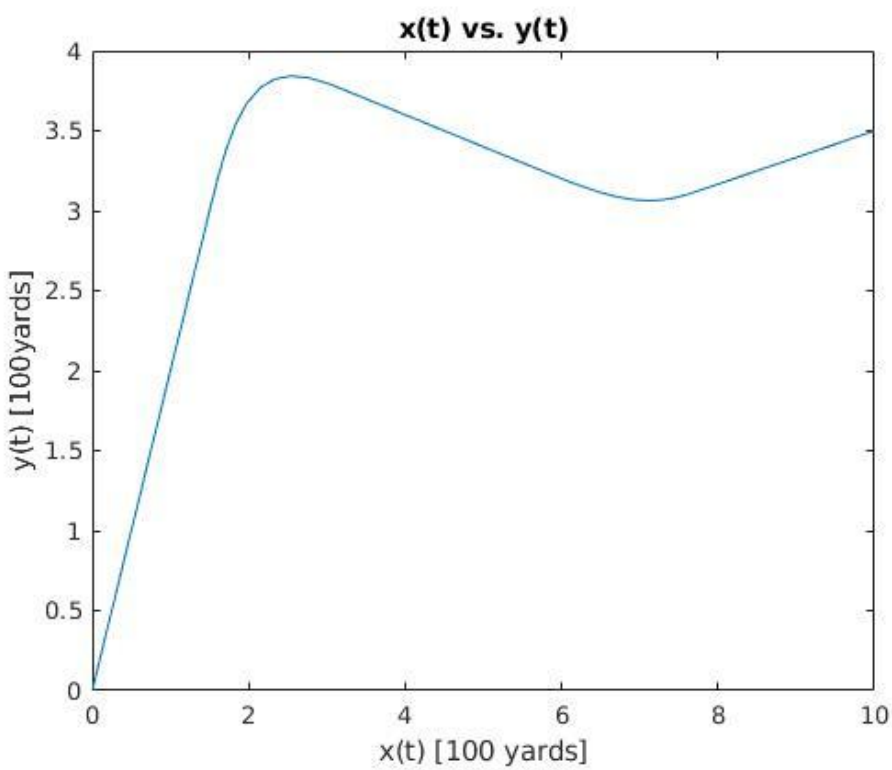
Editor - /home/wen-chung/MATLAB-Drive/Robotics_Application/HW2_1_2.m

+3 | HW2_4_3_Show_Voronoi_80x80.m | HW2_3_1_Bug2_60x60Debugged.m | HW2_1_2.m

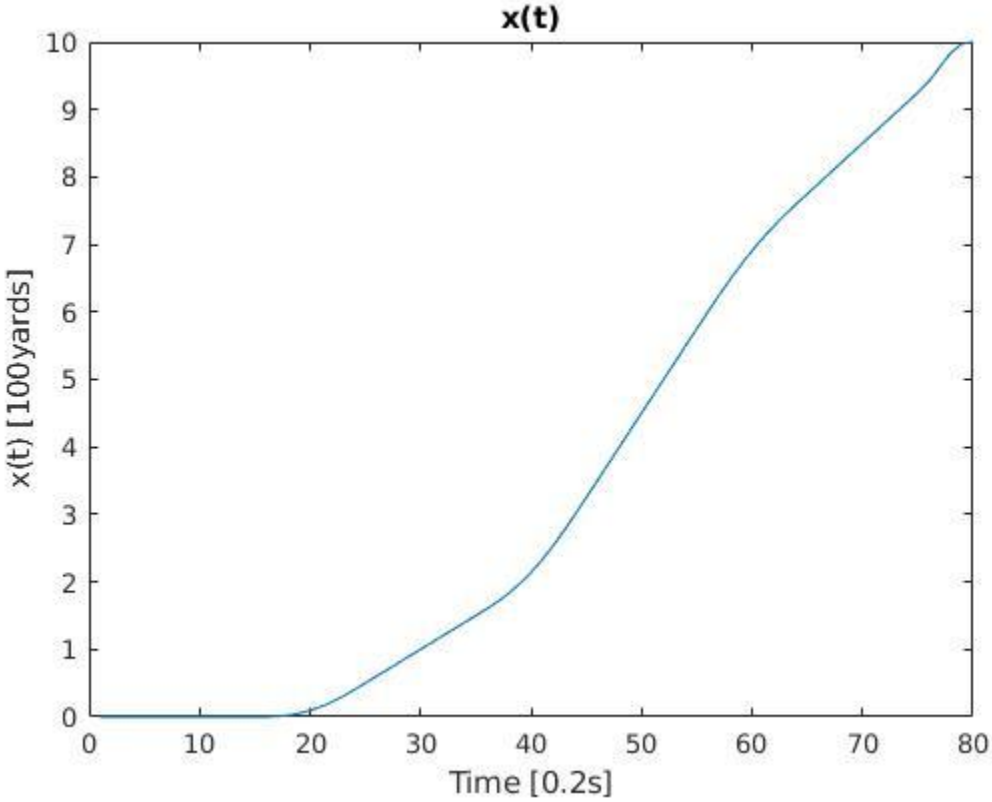
```
% all the via points in x y coordinates
via = [0 0; 2 4; 7 3; 10 3.5]';

% all the via points for theta
via_theta = [0; pi/4; pi/2; pi]';

% plot x y trajectories using given constraints
q0 = mstraj(via(:,[1 2 3 4])', [10,10], [], via(:,1)', 0.2, 2);

% plot theta trajectory using assumed constraints
q0_theta = mstraj(via_theta(:,[1 2 3 4])', [pi/8], [], via_theta(:,1)', 0.2, 1);

% plot x vs. y
figure()
plot(q0(:,1), q0(:,2));

% plot x vs. time
figure()
plot(q0(:,1));

% plot y vs. time
figure()
plot(q0(:,2));

% plot theta vs. time
figure()
plot(q0_theta(:,1));
```

Figure: 1.2 Code

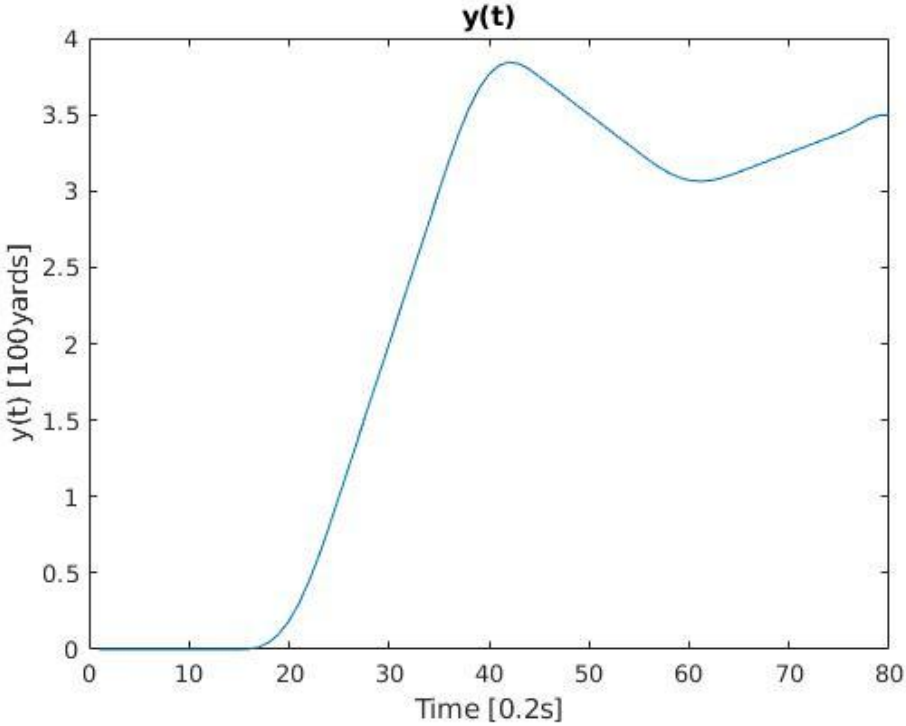
And the resulting x y plot is as follows:



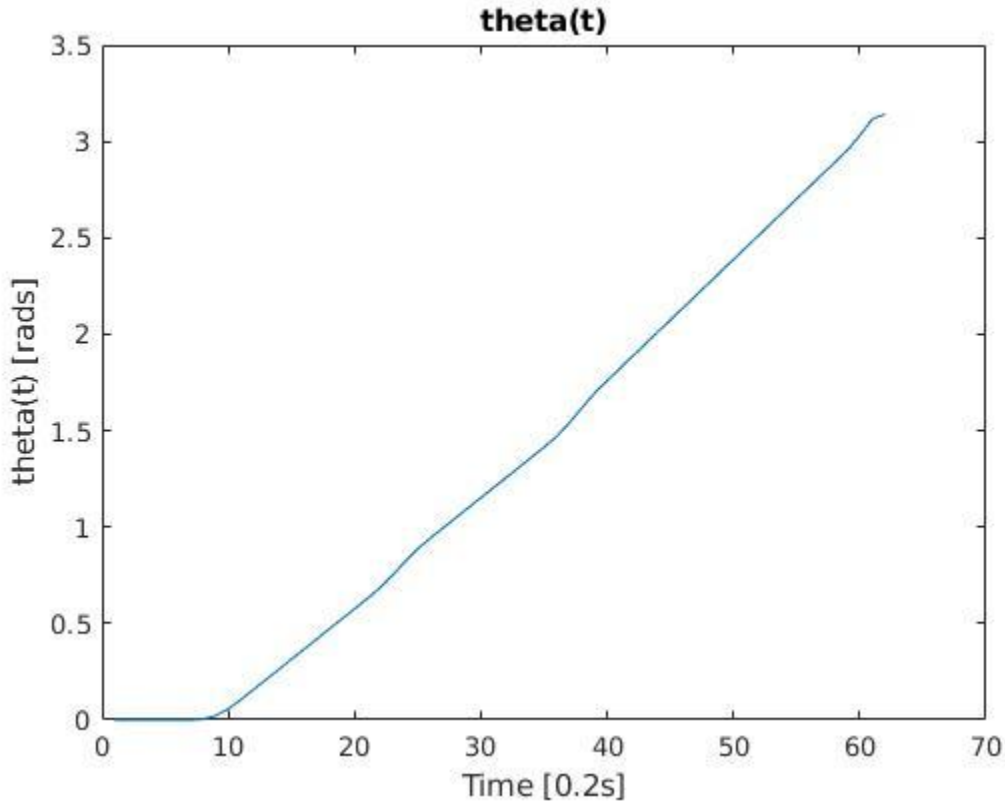
X(t) versus time plot:



Y(t) versus time plot:

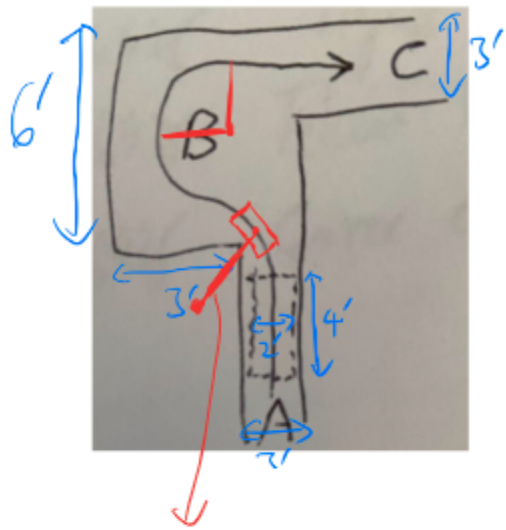


Theta(t) versus time plot:



2.1

Given the physical constraints from the problem, the radii of the curvatures and gamma were calculated:



$$R = 1' + \varepsilon + \frac{3'}{2}$$

$1'$ → ½ the width of robot

ε → gap between robot and corner

$\frac{3'}{2}$ → the distance between center of robot and center of rotation outside

$$= 1' + 1' + \frac{3'}{2}$$

$$= \left(2\tfrac{3}{2}\right)'$$

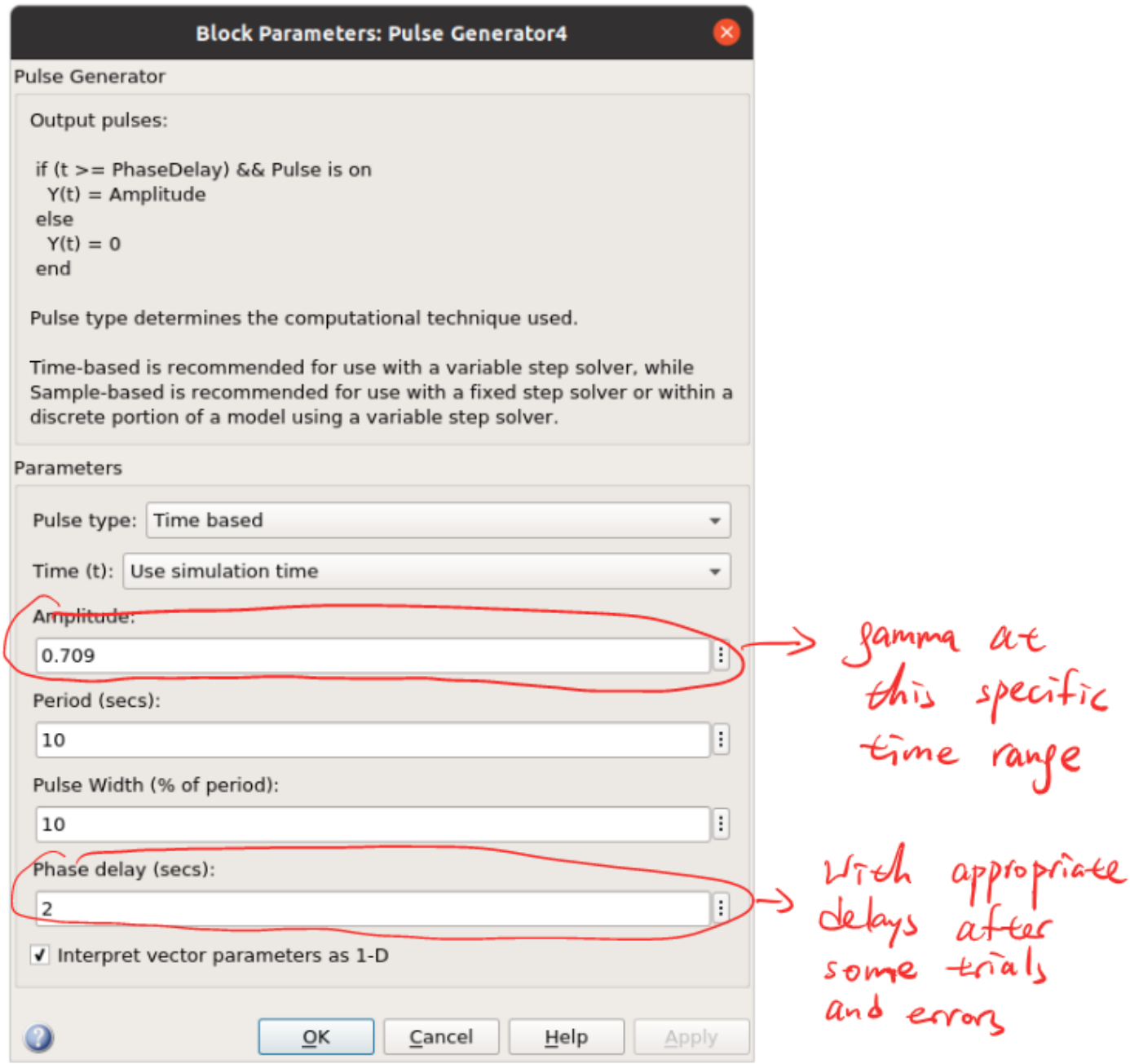
$L = 3'$ → the distance between back wheel axel and front wheel axel

$$\gamma = \tan^{-1}\left(\frac{L}{R}\right)$$

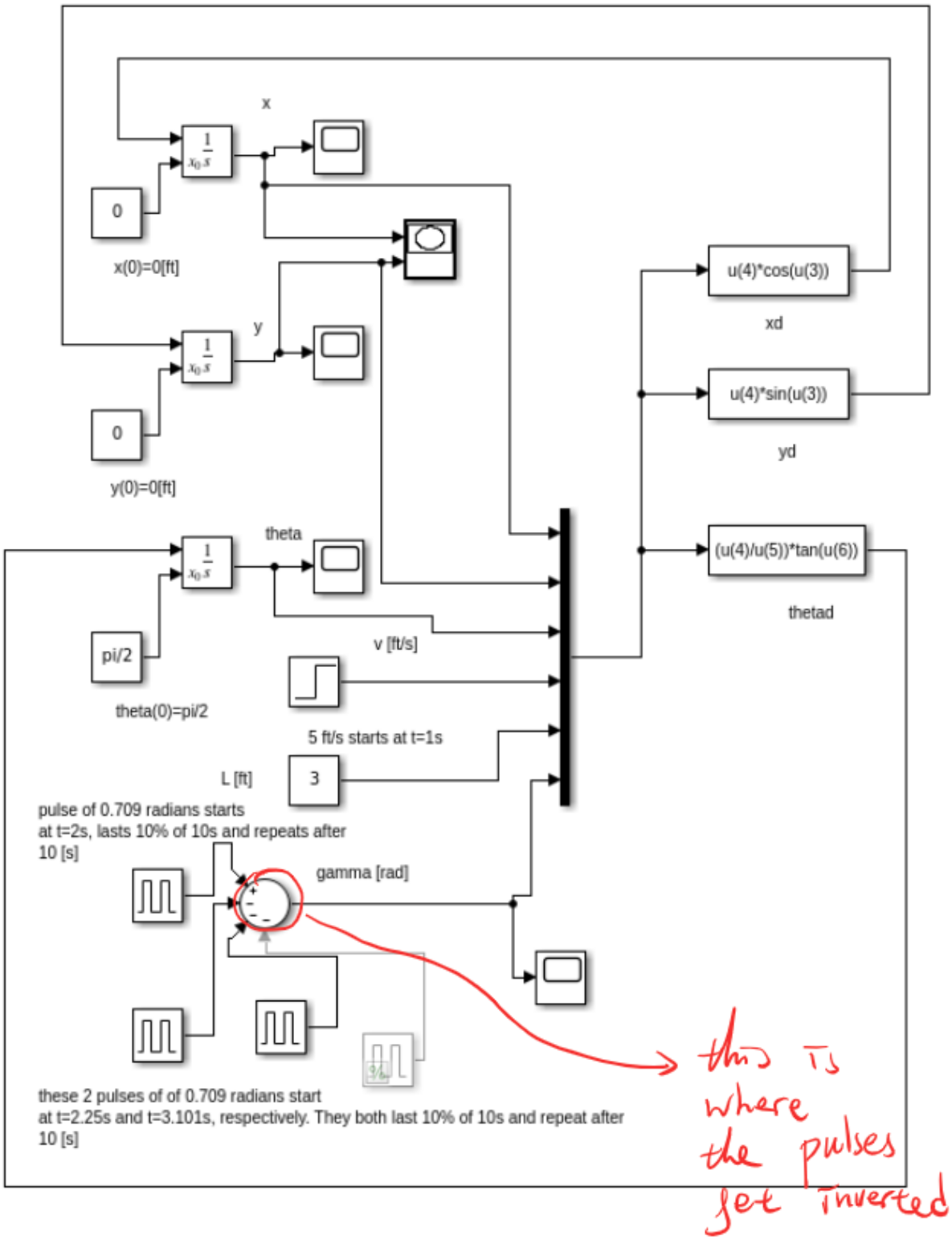
$$= \tan^{-1}\left(\frac{3'}{2\frac{3}{2}'}\right)$$

$$= \tan^{-1}\left(\frac{3}{\frac{7}{2}}\right) = \tan^{-1}\left(\frac{6}{7}\right) \approx 0.709$$

As can be seen above, the gap between the robot and the wall at the turning corner is assumed to be 1 foot. We now can use the above calculated values to replace the pulse values used in the bicycle simulink from the lecture:



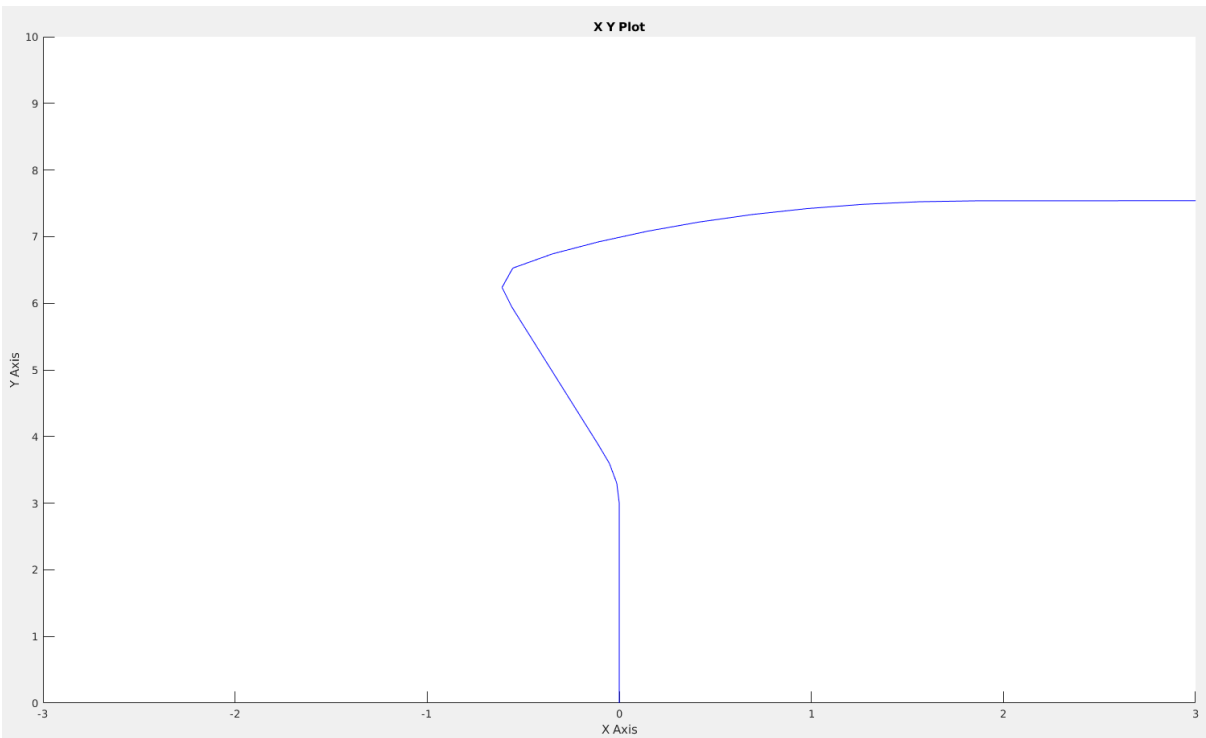
With some additional pulses and appropriate delays determined by some trails and errors, we now have the updated bicycle simulink with the appropriate initial conditions and pulses:

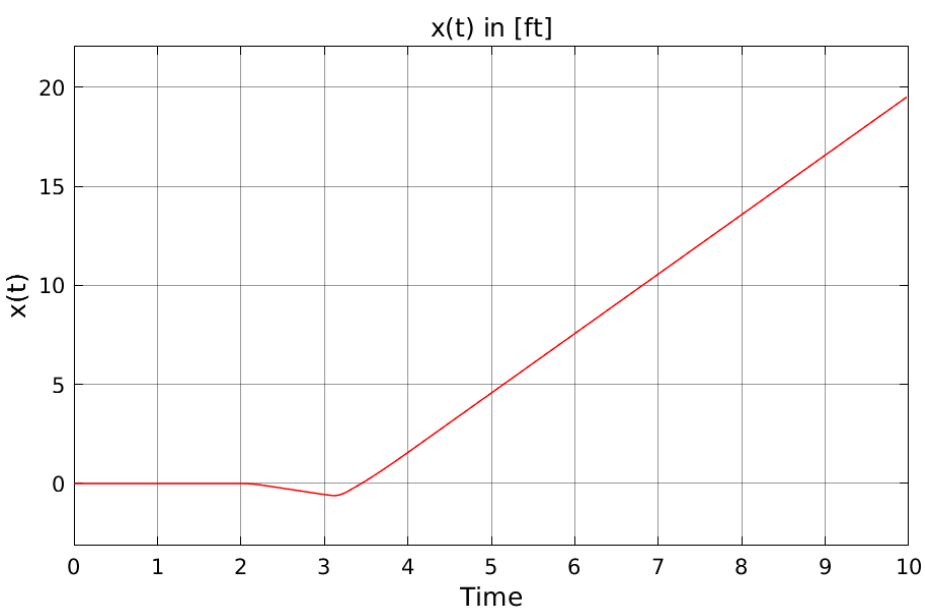


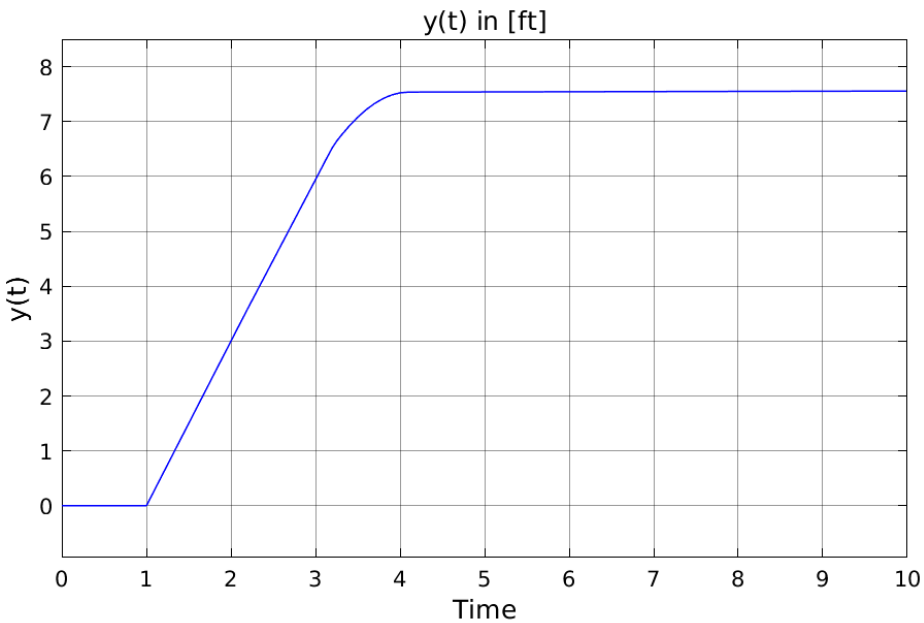
As can be seen above the initial conditions physical constraints are as follows:

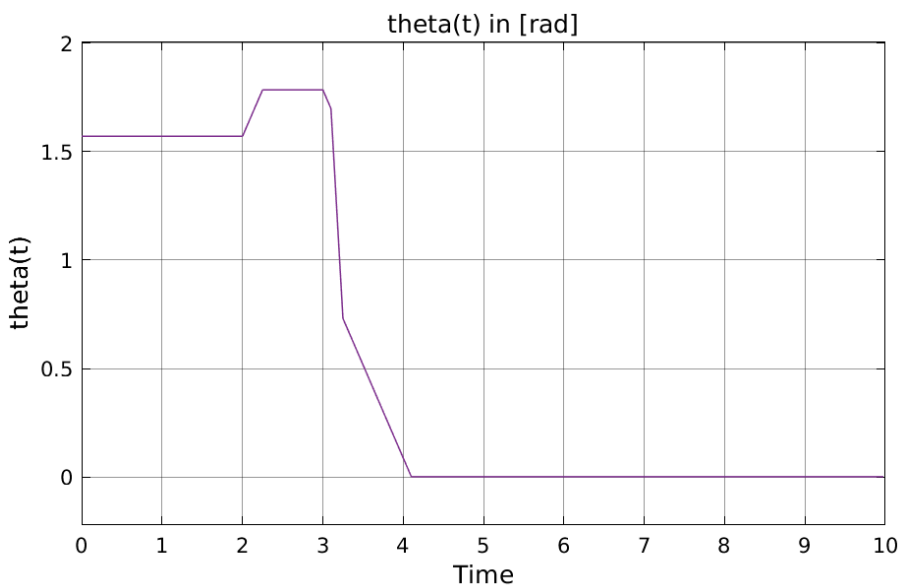
- x(0) = 0 ft
- y(0) = 0 ft
- theta(0) = pi/2 (so that the robot can go in the positive y direction initially)
- L = 3 ft (given)

The following is the final trajectory of the robot:

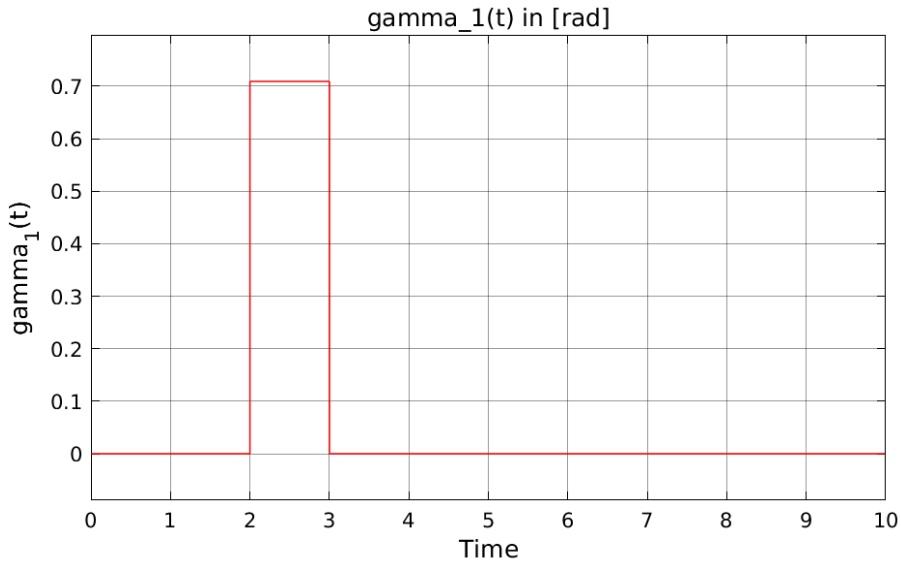


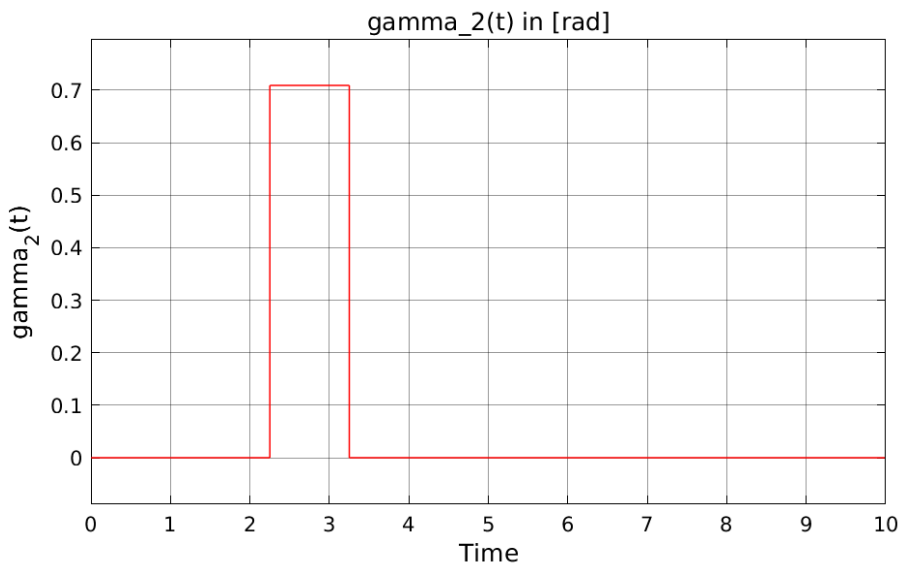


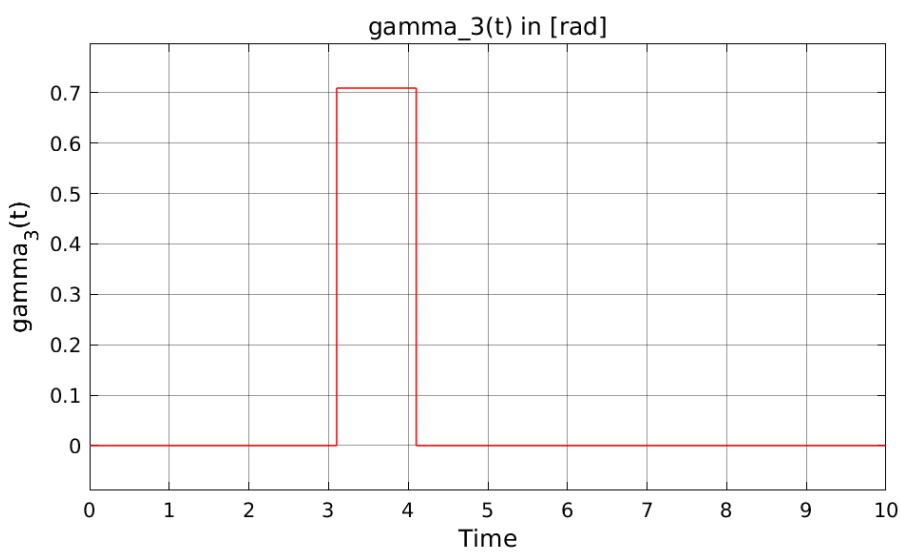




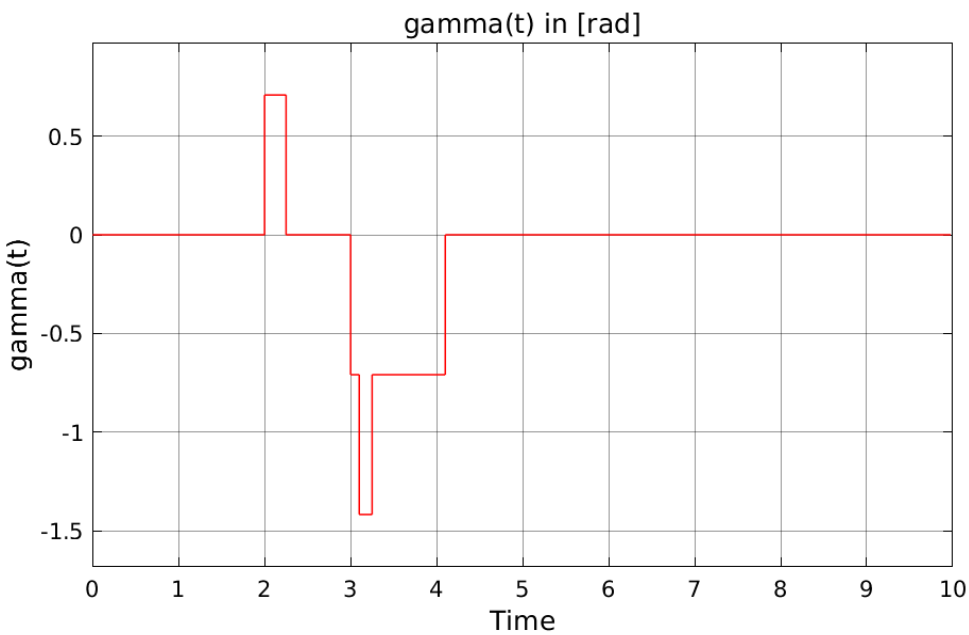
Note there's a delay of 2 seconds in the turning to show that the robot was traveling in the first hallway for a while before entering the big room. This is shown in the pulses:


gamma_1(t) in [rad]
gamma_1(t)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5 6 7 8 9 10
Time


gamma_2(t) in [rad]
gamma_2(t)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5 6 7 8 9 10
Time

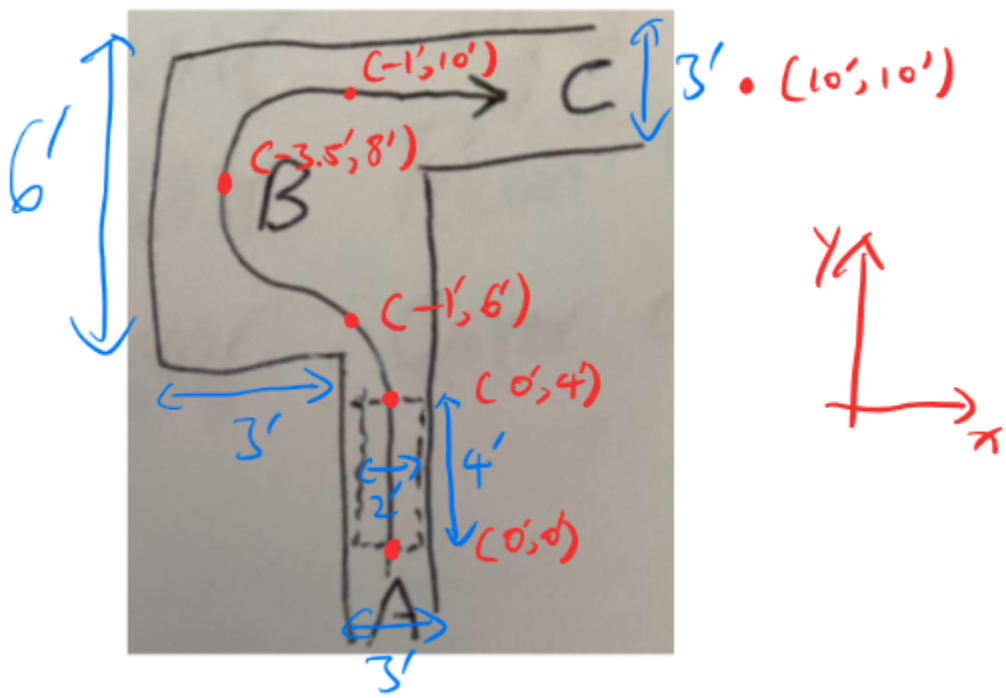

gamma_3(t) in [rad]
gamma_3(t)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5 6 7 8 9 10
Time

Note the last 2 pulses will be inverted so that the bicycle can turn back to the right. And the following is the resulting gamma over the duration of this simulation:

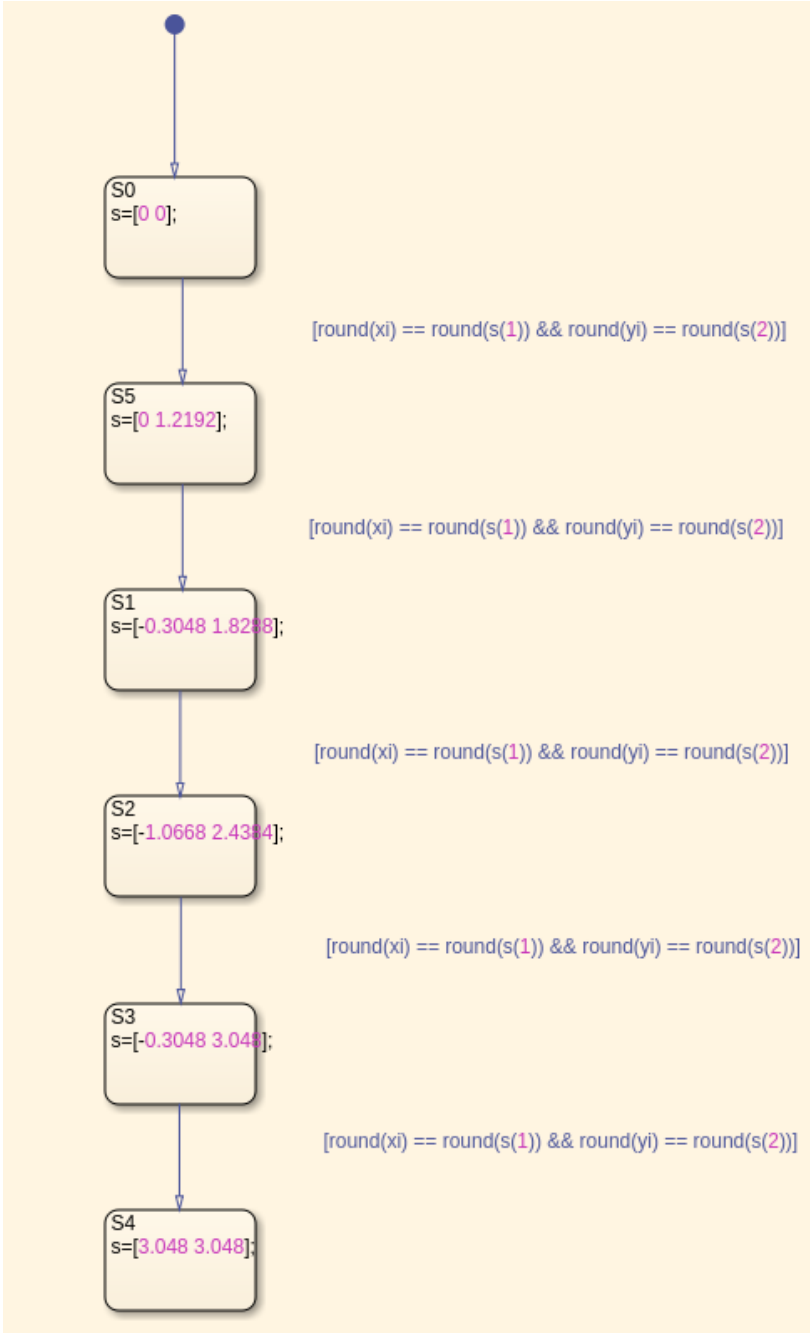


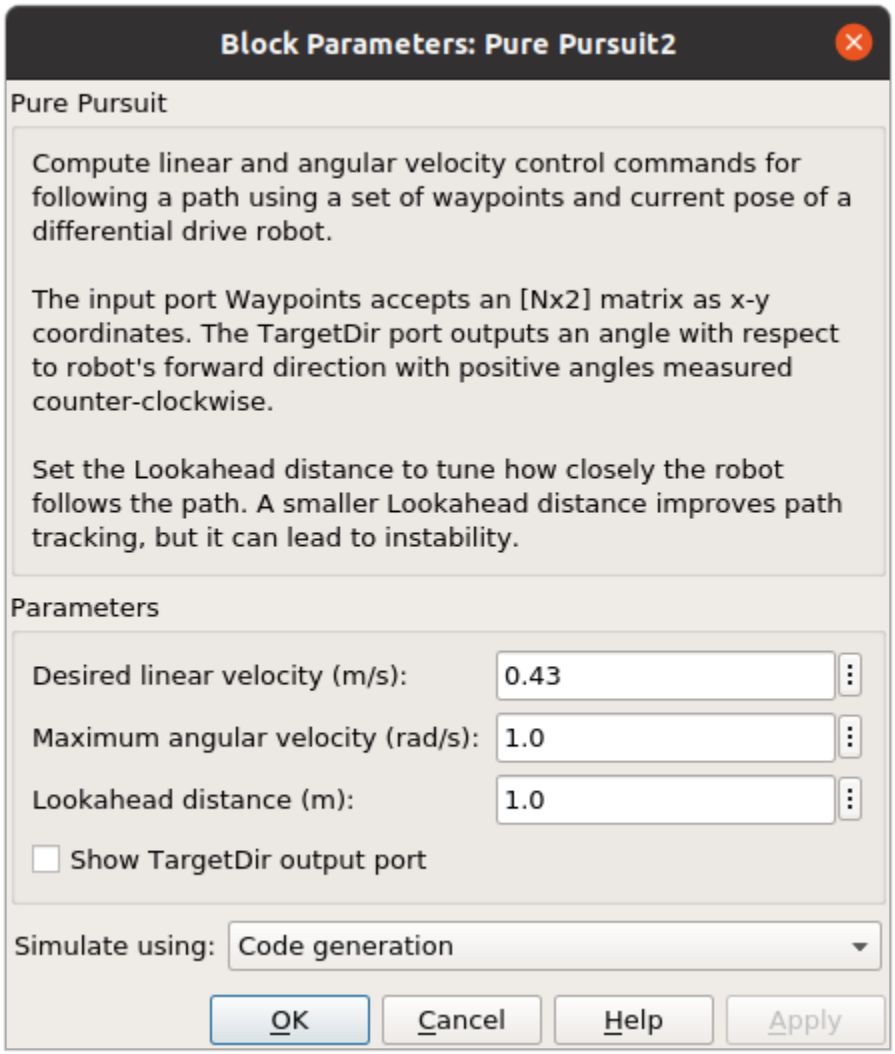
2.2

Since the RTB comes with the differential drive simulink, we can just use it along with the pure pursuit block that comes with the Robotics System Toolbox from Matlab. The pure pursuit block takes custom waypoints as inputs and produces linear and angular velocities for those specific waypoints given the desired linear velocity and maximum angular velocities. Given the scenario from the problem, below the red points represent the defined waypoints, in the unit of feet:

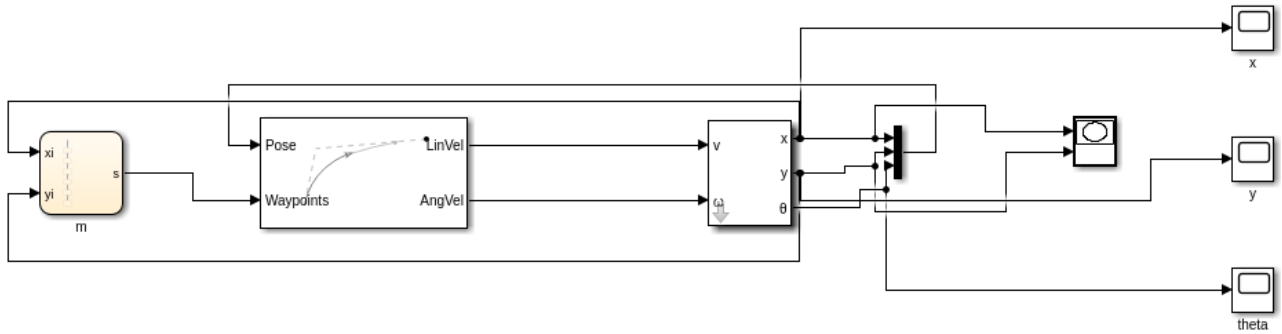


Below are the defined waypoints, desired linear velocity, and maximum angular velocities. Note the waypoints have been converted to meters since the pure pursuit block only works with meters:


S0
s=[0 0];
[round(xi) == round(s(1)) && round(yi) == round(s(2))]
S5
s=[0 1.2192];
[round(xi) == round(s(1)) && round(yi) == round(s(2))]
S1
s=[-0.3048 1.8288];
[round(xi) == round(s(1)) && round(yi) == round(s(2))]
S2
s=[-1.0668 2.4384];
[round(xi) == round(s(1)) && round(yi) == round(s(2))]
S3
s=[-0.3048 3.048];
[round(xi) == round(s(1)) && round(yi) == round(s(2))]
S4
s=[3.048 3.048];

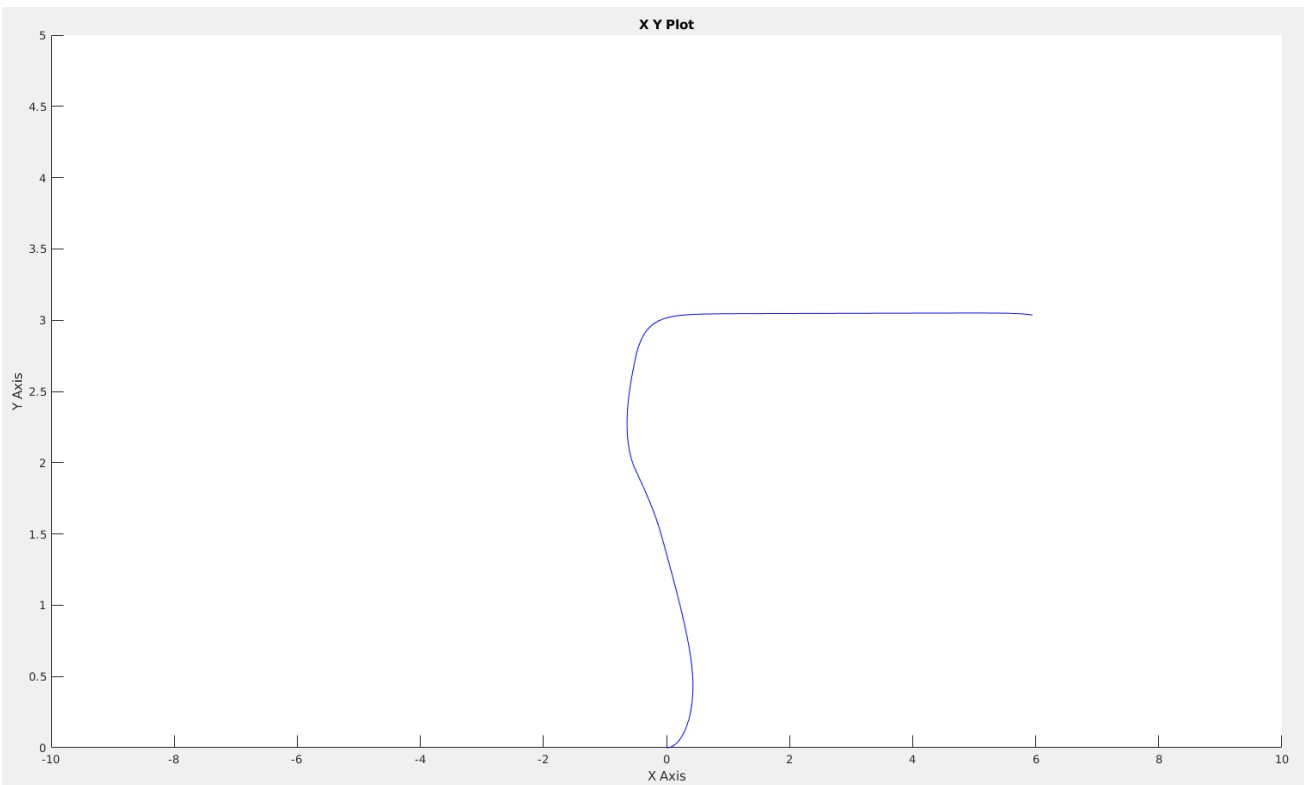


The if statements between the states are for transitioning among the states. Below is the overall simulink for this differential drive vehicle:

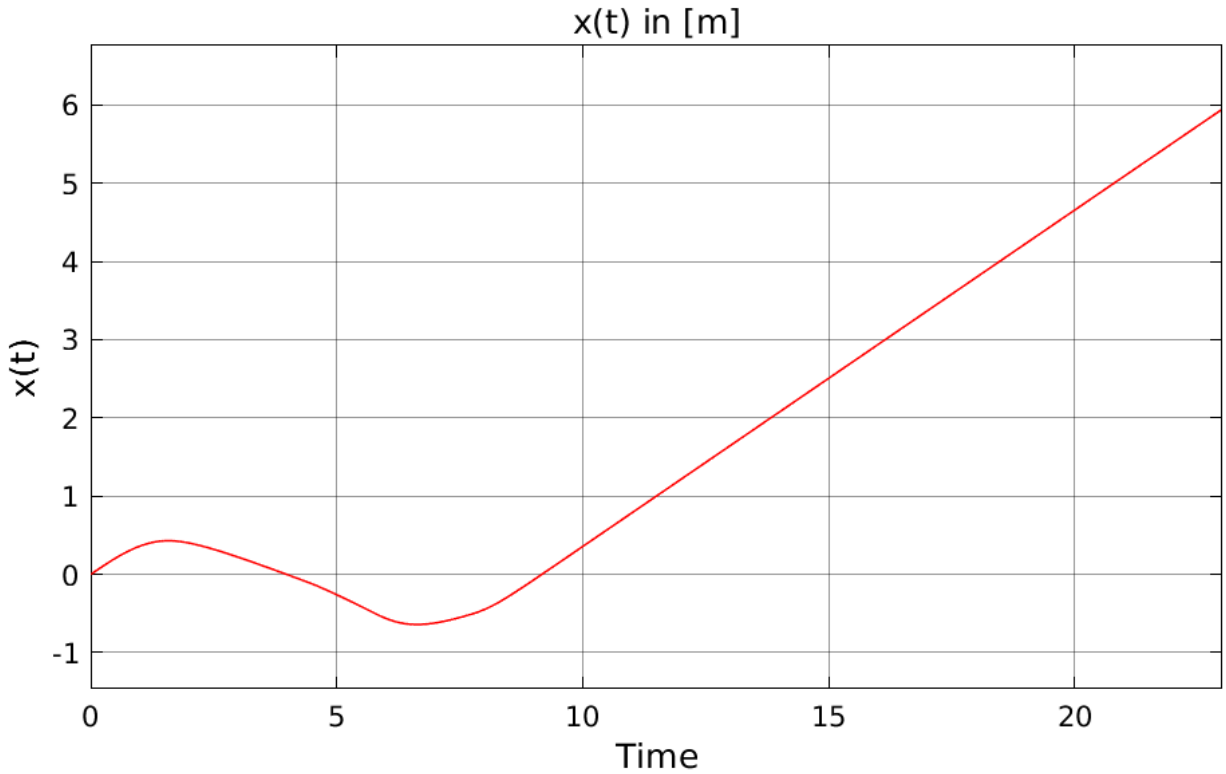


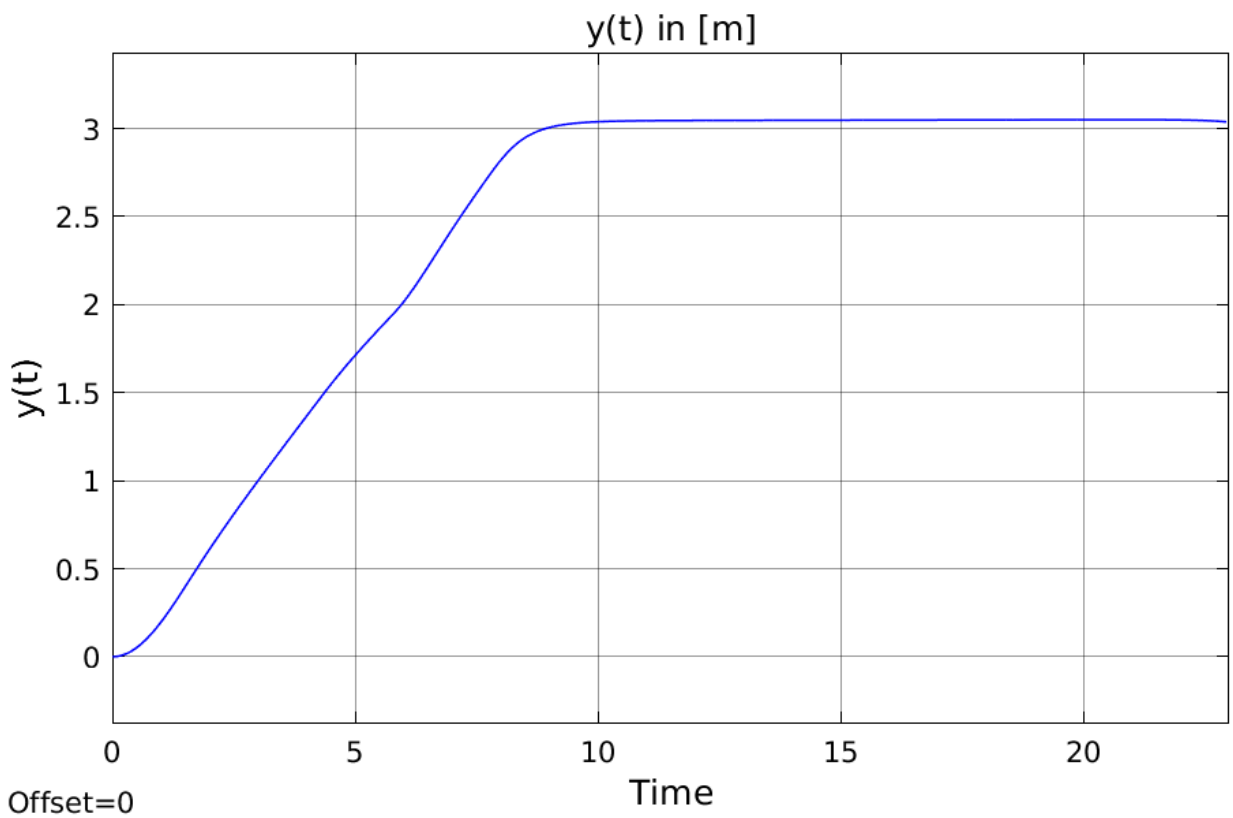
[1]

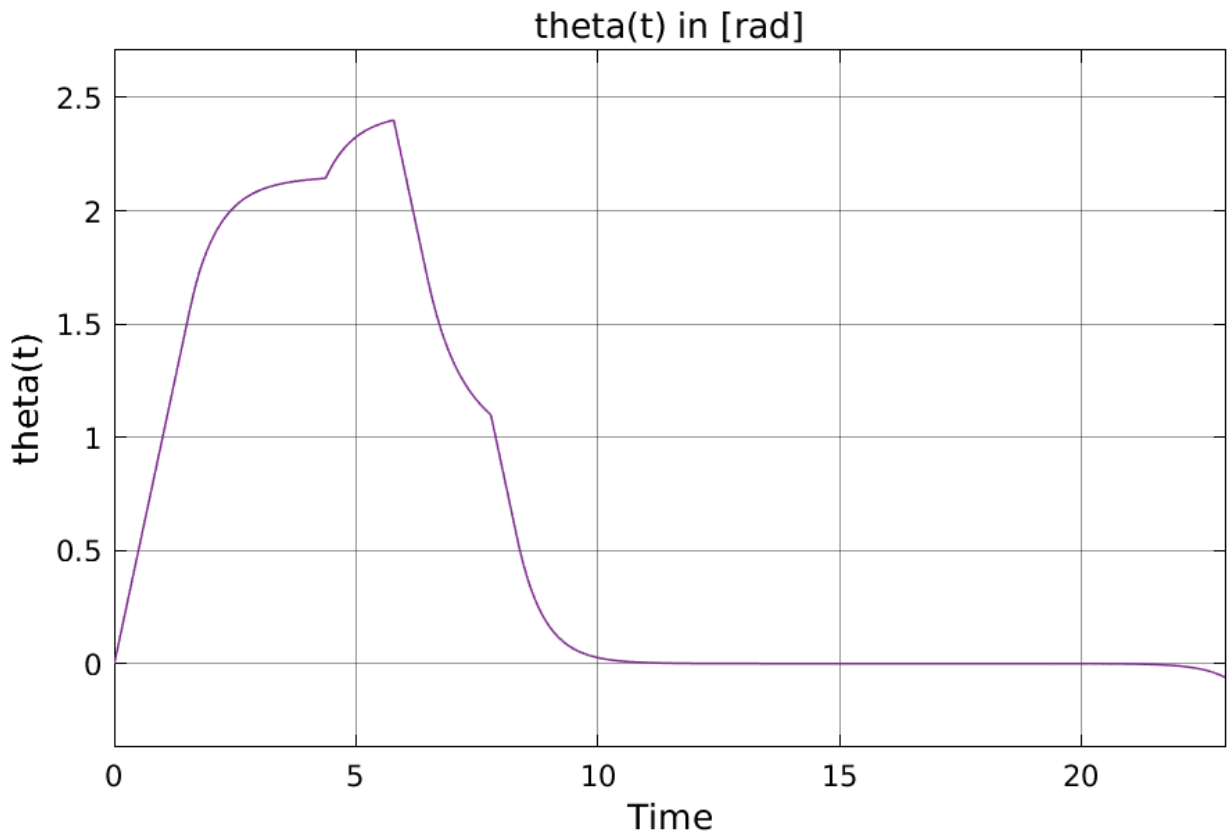
Below are the trajectories of the differential drive vehicle in this problem:



[1] The yellow block to the left contains all the waypoints.


x(t) in [m]
x(t)
6
5
4
3
2
1
0
-1
0
5
10
15
20
Time


y(t) in [m]
y(t)
3
2.5
2
1.5
1
0.5
0
0
5
10
15
20
Time
Offset=0

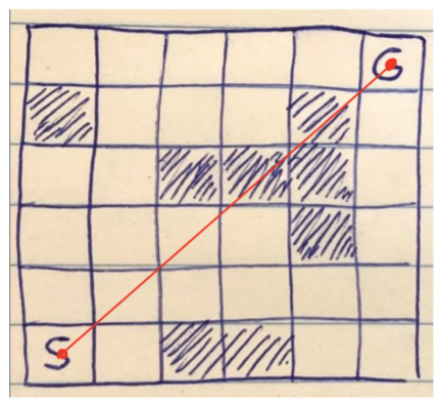

theta(t) in [rad]
theta(t)
Time
2.5
2
1.5
1
0.5
0
0
5
10
15
20

3) Obstacle avoidance and path following are two of the most fundamental difficulties in mobile robotics. The goal of navigation is to find the best path from start to finish in a congested environment. Here we eventually used the command makemap(60) to scale up the 6x6 map used in the description of the homework. We will explain the reason for that in the following pages.

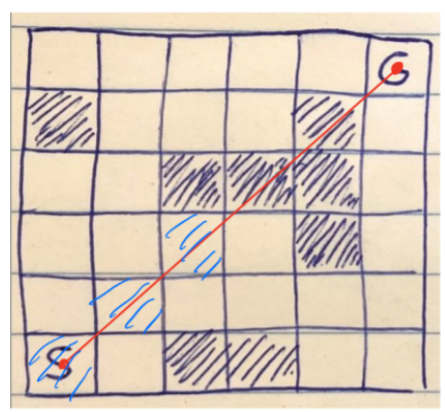
When we have a starting and stopping goal position established, we can use the BUG algorithm. We will use BUG1 algorithm when we want the robot to move in the direction of the goal until it encounters an impediment, and it goes clockwise until it makes a full circle back to the starting position. The robot then follows the boundary to the place closest to the target along the boundary. The robot advances directly toward the target from this point. If a new impediment appears, the same method is followed. This method is not as effective as the BUG2 algorithm, which in turn the robot always tries to follow the line of sight to the goal.

3.1 Using the BUG2 algorithm, the following steps should be performed:

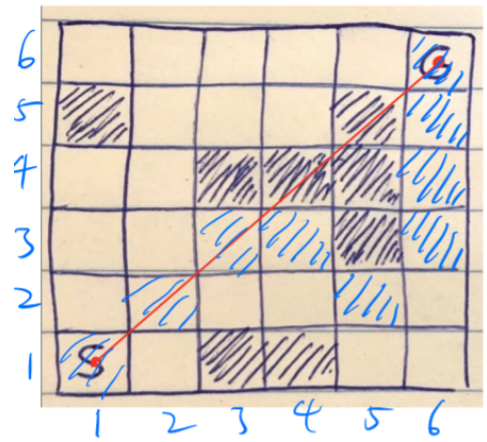
1) Define a straight line between the starting point and goal:



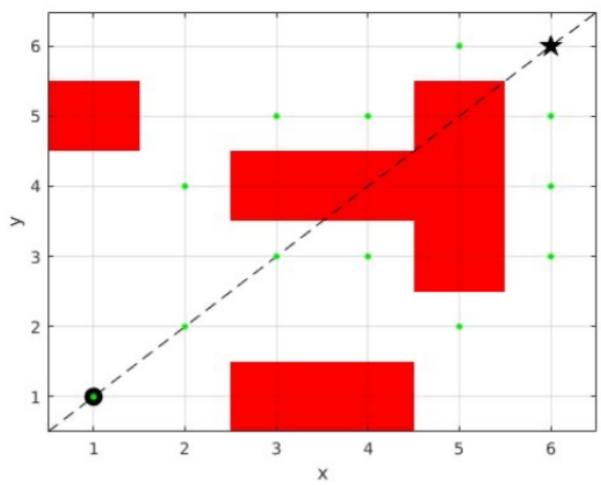
2) Go through the cells that are on the line until the first obstacle



3) Turn right and move along the wall until the start-goal line is encountered



However, the implementation of the 6x6 grid in MATLAB ended in a trap:




```
>> HW2_3_1_Bug2_6x6
Error using Bug2/next (line 262)
robot is trapped

Error in Bug2/query (line 160)
                robot = bug.next(robot);

Error in HW2_3_1_Bug2_6x6 (line 19)
bug.query([1, 1], [6, 6], 'animate');
```

After some debugging, it's determined that the trap came from the edge points created by the edge list function, one of the RTB functions:

Editor - /home/wen-chung/MATLAB-Drive/Robotics_Application/HW2_3_1_Bug2_6x6.m

+8 | Bug2Example.m | DXform.m | HW2_3_1_Bug2_60x60Bug.m | HW2_3_1_Bug2_6x6.m

```
load map6x6
% load map24x24
% load map60x60
% load map60x60Debugged
% load map60x60Bug
% load map60x60Debug
% load BugMap60x60


bug = Bug2(map6x6);
% bug = Bug2(map24x24);
% bug = Bug2(map60x60);
% bug = Bug2(BugMap60x60);
% bug = Bug2(map60x60Debugged);
% bug = Bug2(map60x60Bug);
% bug = Bug2(map60x60Debug);
bug.plot();
% bug.query([2, 2], [22, 22], 'animate');
bug.query([1, 1], [6, 6], 'animate');
% bug.query([5, 5], [55, 55], 'animate');
% bug.query([1, 1], [60, 60], 'animate');
% bug.query([], [], 'animate');
```

The Bug2 class calls the edgelist function to create edge points around the obstacle, as can be seen below

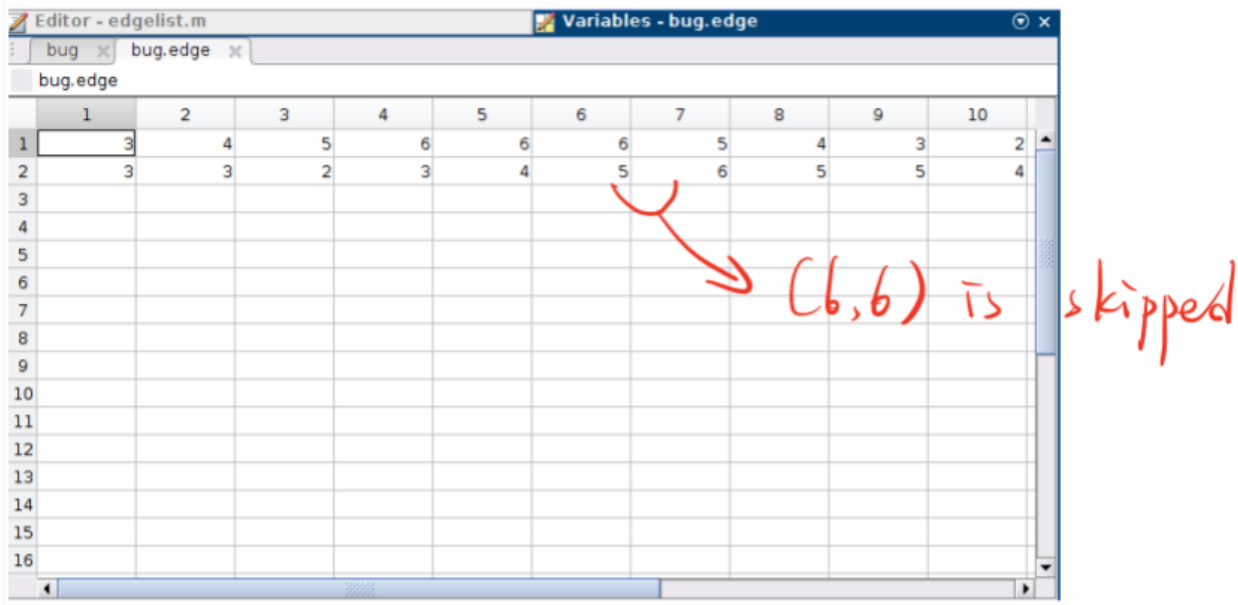
Editor - Bug2.m | Variables - bug

+7 | Bug2.m | edgelist.m | Bug2Example.m | DXform.m | HW2_3_1_Bug2_60x60Bug.m

```
                % detect if next step is an obstacle
                if bug.isoccupied(robot + [dx; dy])
                    bug.message('(%d,%d) obstacle!', n);
                    bug.H(bug.j,:) = robot; % define hit point
                    bug.step = 2;
                    % get a list of all the points around the obstacle
                    bug.edge = edgelist(bug.occgridnav == 0, robot);
                    bug.k = 2;  % skip the first edge point, we are already there
%                     bug.k = 1;  % skip the first edge point, we are already there
                else
                    n = robot + [dx; dy];
                end
            end % step 1

            if bug.step == 2
                % Step 2.  Move around the obstacle until we reach a point
                % on the M-line closer than when we started.
                if colnorm(bug.goal-robot) == 0 % are we there yet?
                    return
                end

                if bug.k <= numcols(bug.edge)
```

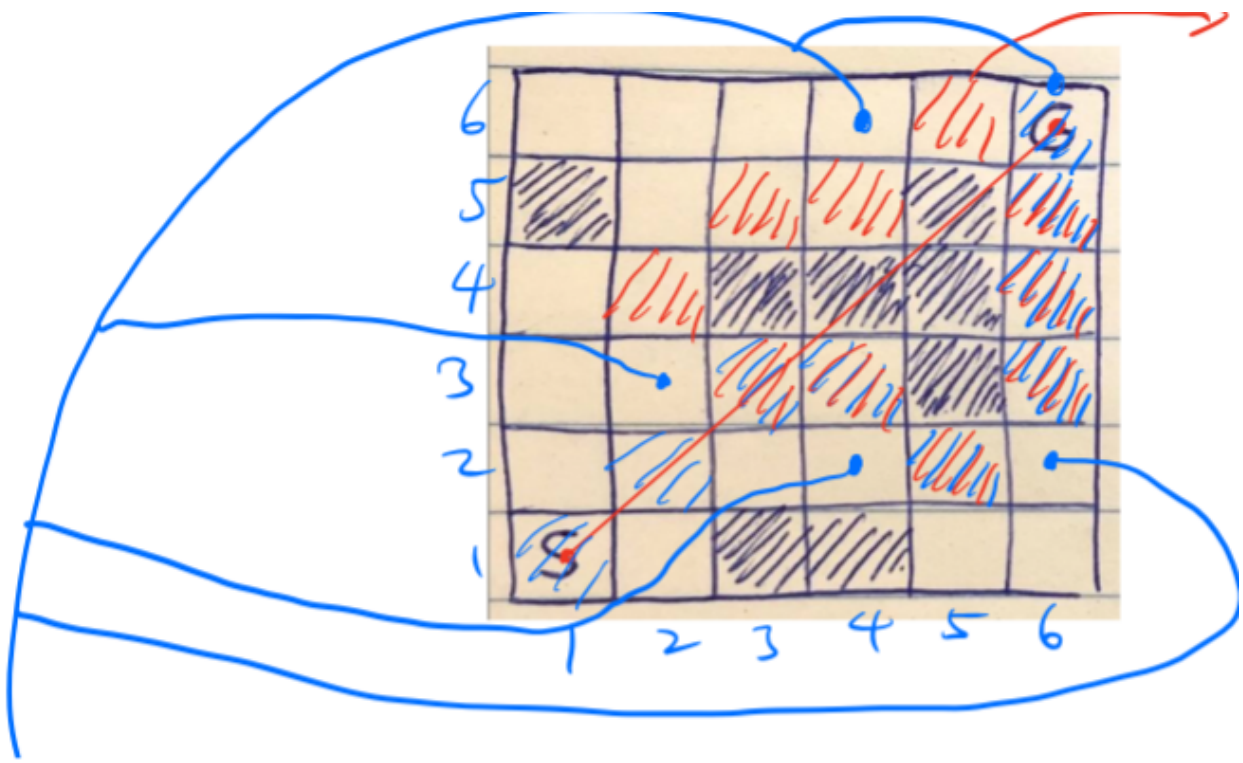
the edgelist function is called to create edge points around the obstacle

Below are some of the edge points created by this function:



	1	2	3	4	5	6	7	8	9	10
1	3	4	5	6	6	6	5	4	3	2
2	3	3	2	3	4	5	6	5	5	4

Compare it with the map, the coordinates of the goal is skipped (6,6):



Note: The red shaded grids are edge points according to the edge list function.

As can be seen above, the grids that are at the corners of the obstacle (dotted in blue) do not count as edge points according to the edgelist function. A little adjustment to the map can clearly show that:

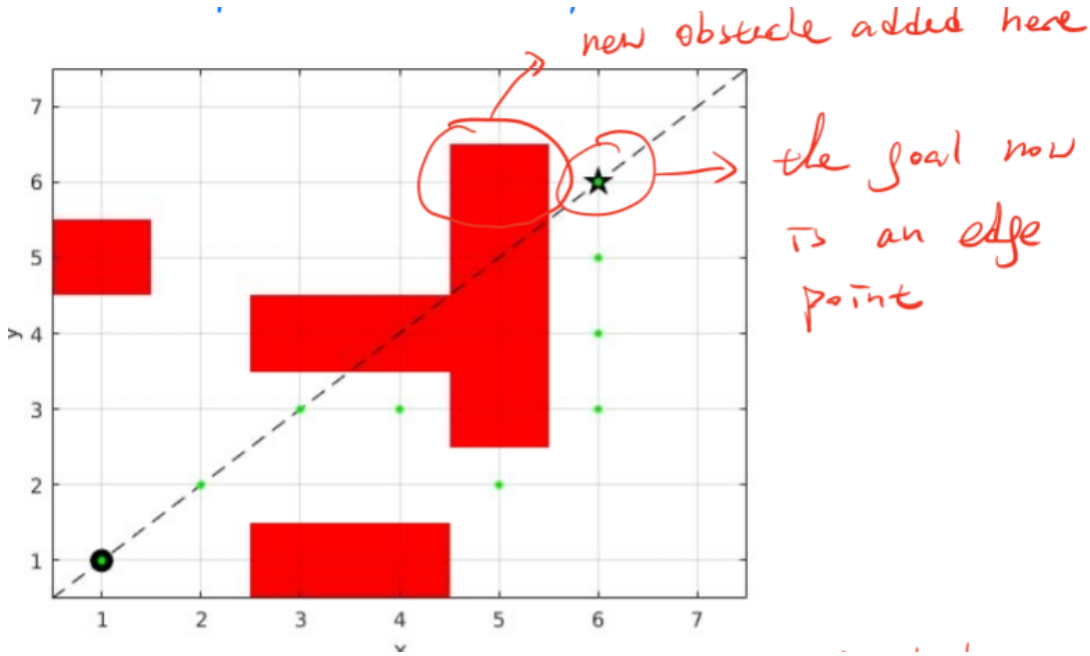


Figure: New map

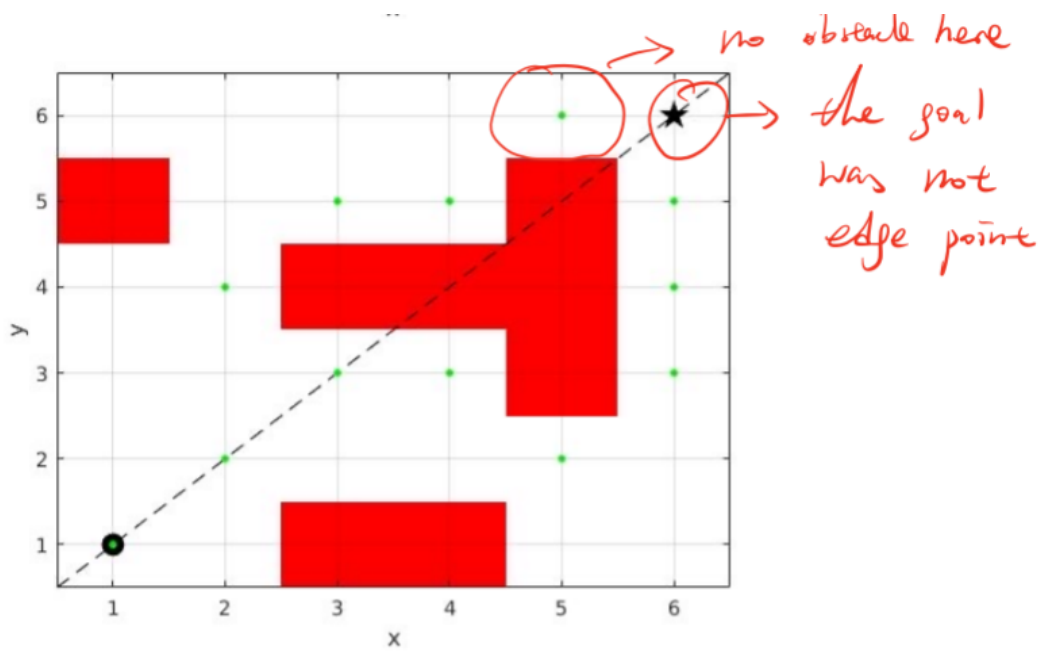
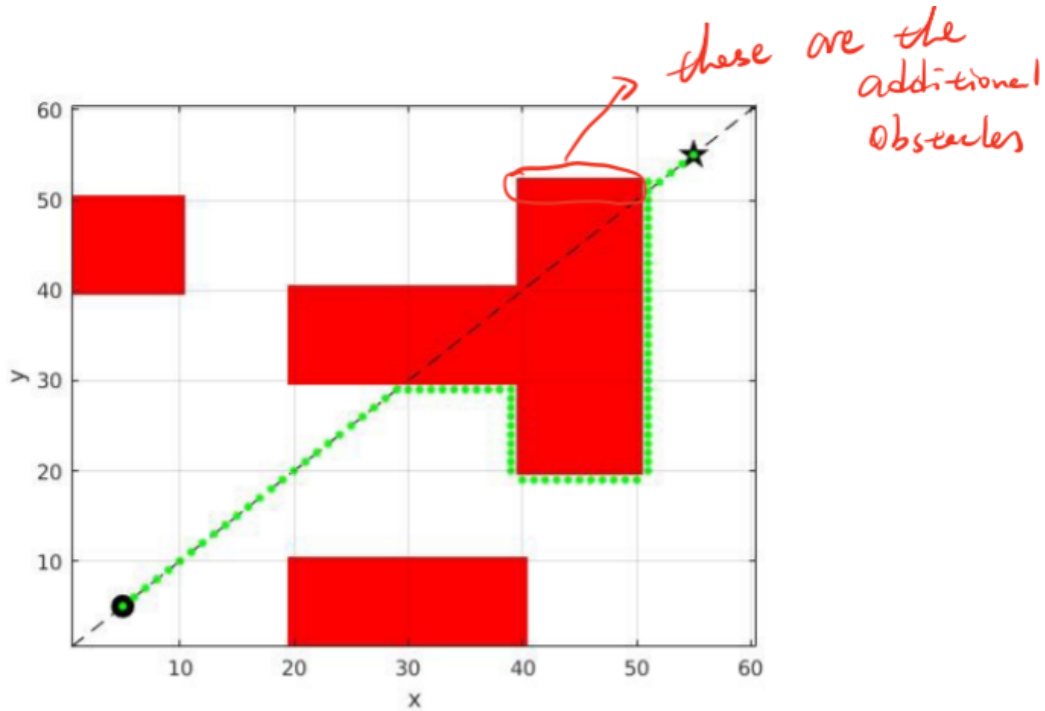
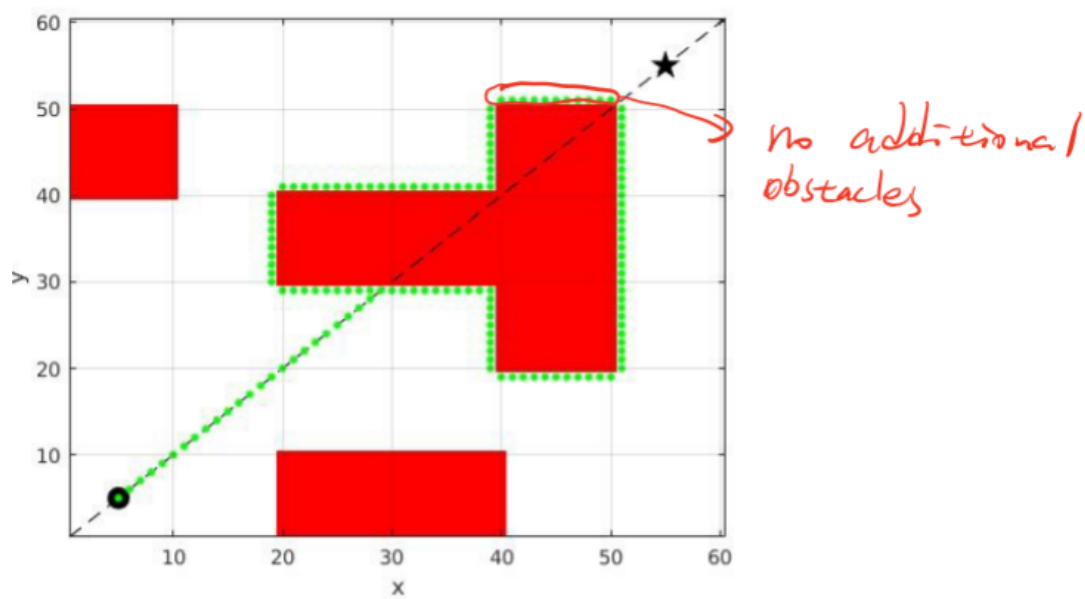


Figure: Original Map

However, this new map does not look consistent with the map given in the problem. So the sort of "workaround" solution was making a 60x60 version of the map with some additional obstacles:



The reason for the slight additional obstacles is that all the points within the (50-60, 50-60) area will then be counted as edge points. If the map were perfectly scaled up without any additional obstacles:



Since there are no additional obstacles, all the points within the (50-60, 50-60) region will not be counted as edge points. ***So to implement the BUG2 algorithm with the type of obstacles given in the problem "almost" correctly, the 60x60 map with some additional obstacles overlapping is the way to go. In fact, the goal given in the problem wasn't well chosen since it's not one of the edge points to begin with.***

The code below demonstrates the implementation of the BUG2 algorithm. The robot moves in a diagonal direction towards the goal, [55, 55] (because it's at the center of the region [50-60, 50-60]), but when it hits an obstacle it follows closely along the edges until the end is in sight. Then, it moves towards the end goal.

Editor - /home/wen-chung/MATLAB-Drive/Robotics_Application/HW2_3_1_Bug2_60x60Debugged.m

+2 | HW2_4_3_Voronoi_80x80.m | HW2_4_3_Show_Voronoi_80x80.m | HW2_3_1_Bug2_60x60Debugged.m

```
% load map6x6
% load map24x24
% load map60x60
load map60x60Debugged
% load map60x60Bug
% load map60x60Debug
% load BugMap60x60


% bug = Bug2(map6x6);
% bug = Bug2(map24x24);
% bug = Bug2(map60x60);
% bug = Bug2(BugMap60x60);
bug = Bug2(map60x60Debugged);
% bug = Bug2(map60x60Bug);
% bug = Bug2(map60x60Debug);
bug.plot();
% bug.query([2, 2], [22, 22], 'animate');
% bug.query([1, 1], [6, 6], 'animate');
bug.query([5, 5], [55, 55], 'animate');
% bug.query([1, 1], [60, 60], 'animate');
% bug.query([], [], 'animate');
```

3.2

We have created a distance transform map for the robot to navigate using DXform. This is a subclass of the Navigation class in the Robotics Toolbox, which computes the path of minimum distance needed to take for the robot. This is a vitally important feature to be implemented in a robot, especially when it comes to collision avoidance. In the code below we set our start and stopping positions, created the navigation object, planned the goal and executed the animation.

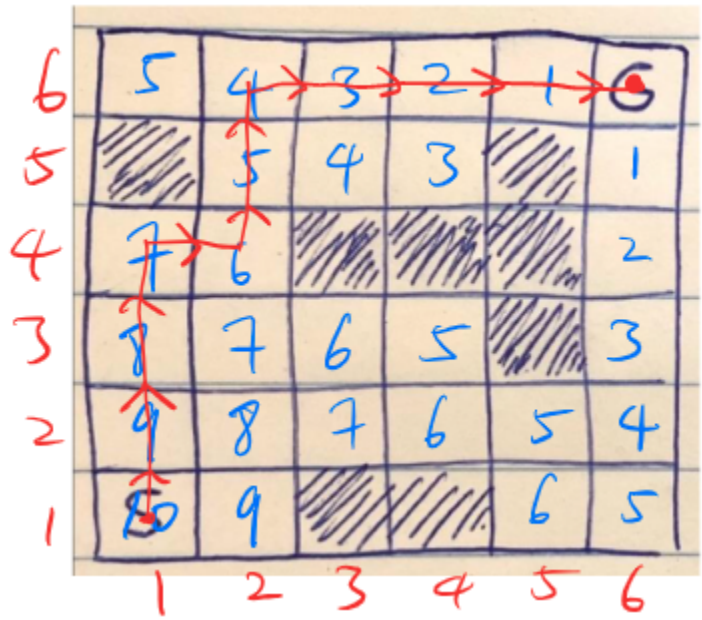
To perform DXform manually, we need to first obtain the cost map using the cityblock method:

6	5	4	3	2	1	G
5		5	4	3		1
4	7	6				2
3	8	7	6	5		3
2	9	8	7	6	5	4
1	S	9			6	5
	1	2	3	4	5	6

City block distance to goal

$$(1,1): |6-1| + |6-1| = 10 = D[(1,1);(6,6)]$$
$$(1,2): |6-1| + |6-2| = 9 = D[(1,2);(6,6)]$$
$$(1,3): |6-1| + |6-3| = 8 = D[(1,3);(6,6)]$$
$$\vdots$$
$$(2,1): |6-2| + |6-1| = 9 = D[(2,1);(6,6)]$$
$$(2,2): |6-2| + |6-2| = 8 = D[(2,2);(6,6)]$$
$$(2,3): |6-2| + |6-3| = 7 = D[(2,3);(6,6)]$$
$$\vdots$$

Since diagonal motion is not allowed, the path that has the least cost is the following:

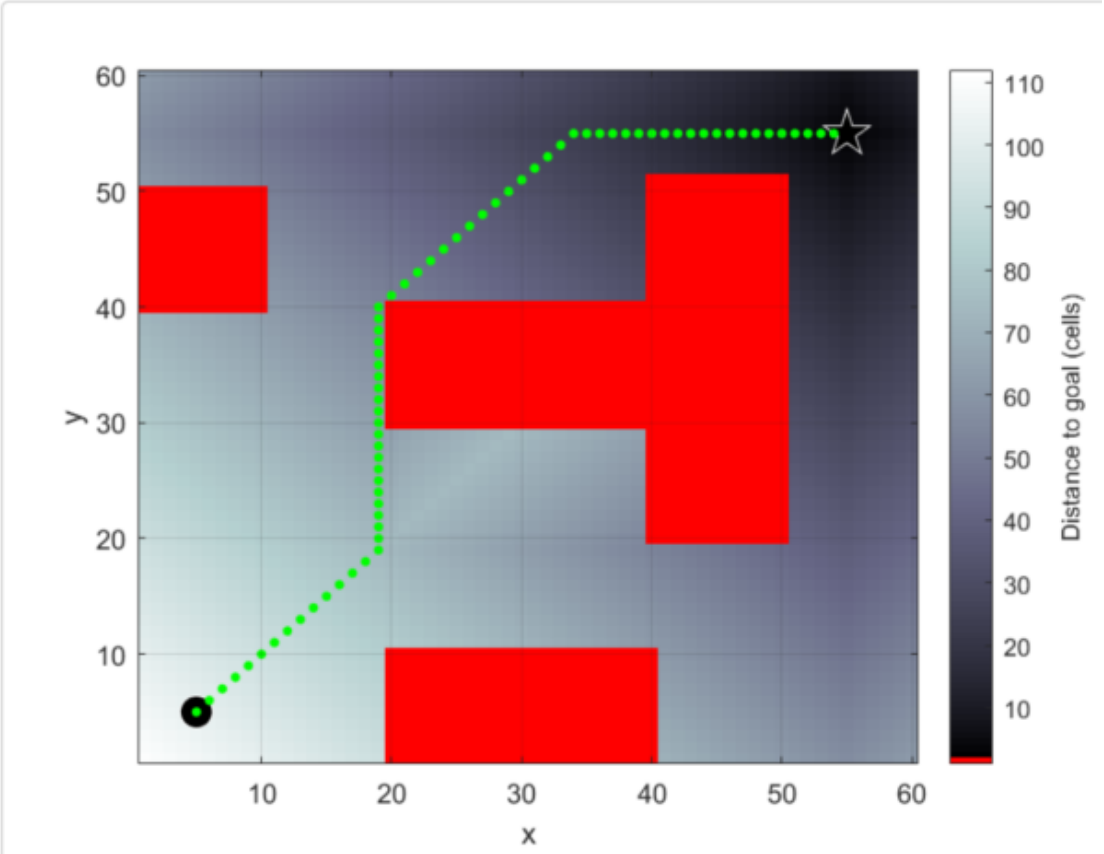


Manual calculation of cost map

Matlab implementation 1 (diagonal motion allowed with 60x60 map)

```
load map2            %load map

start = [5, 5];      %starting point
goal = [55, 55];     %objective point
dx = DXform(world); %create navigation object
dx.plan(goal)        %create plan for specified goal (as Peter Corke says it)
dx.query(start, 'animate')
```

3.2 Code



3.2 Results (with 60x60 map)

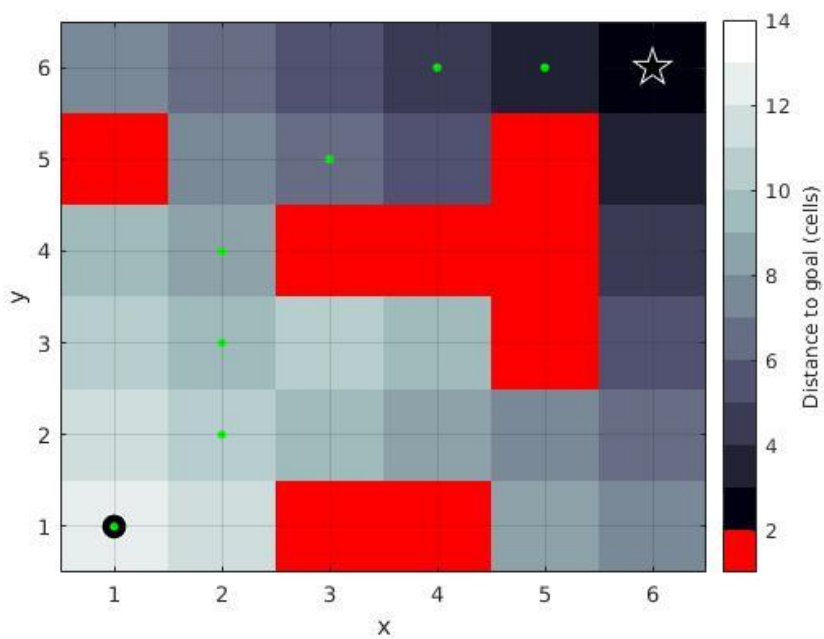
Matlab implementation 2 (diagonal motion allowed with 6x6 map)

Editor - /home/wen-chung/MATLAB-Drive/Robotics_Application/HW2_3_2_DXform.m

+2 | HW2_4_3_Voronoi_80x80.m | HW2_4_3_Show_Voronoi_80x80.m | HW2_3_2_DXform.m

```
%map = makemap(6);
load map6x6

dx = DXform(map6x6, 'cityblock');
% dx = DXform(map6x6);
dx.plan([6, 6]);
dx.plot();
dx.query([1, 1], 'animate');
```

3.2 Code (with 6x6 map)



3.2 Results (with 60x60 map)

Matlab implementation 3 (diagonal motion not allowed with 6x6 map)

With the no diagonal motion allowed option, we need to look into the RTB function DXform to disable any diagonal motion:

Editor - /home/wen-chung/MATLAB Add-Ons/Toolboxes/Robotics Toolbox for MATLAB/DXform.m

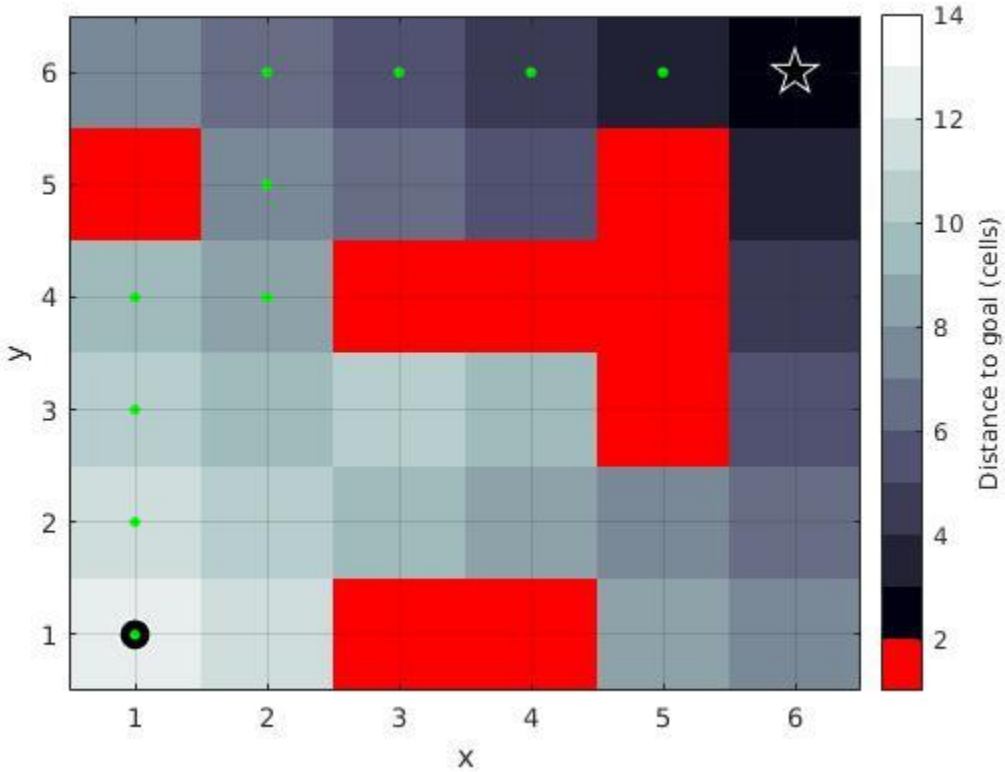
+2 | HW2_3_2_DXform.m | DXform.m | HW2_3_3_Dstar_6x6.m | HW2_4_2_Dstar_80x80.m

```
        function n = next(dx, robot)
            if isempty(dx.distancemap)
                error('No distancemap computed, you need to plan');
            end

            % list of all possible directions to move from current cell

            % original
%               directions = [
%                   -1 -1
%                   0 -1
%                   1 -1
%                   -1 0
%                   0 0
%                   1 0
%                   -1 1
%                   0 1
%                   1 1];

            % no diagonal motion

            directions = [
                0 -1
                -1 0
                0 0
                1 0
                0 1];

            x = robot(1); y = robot(2);
```

diagonal motions were allowed

only vertical and horizontal motions are allowed now

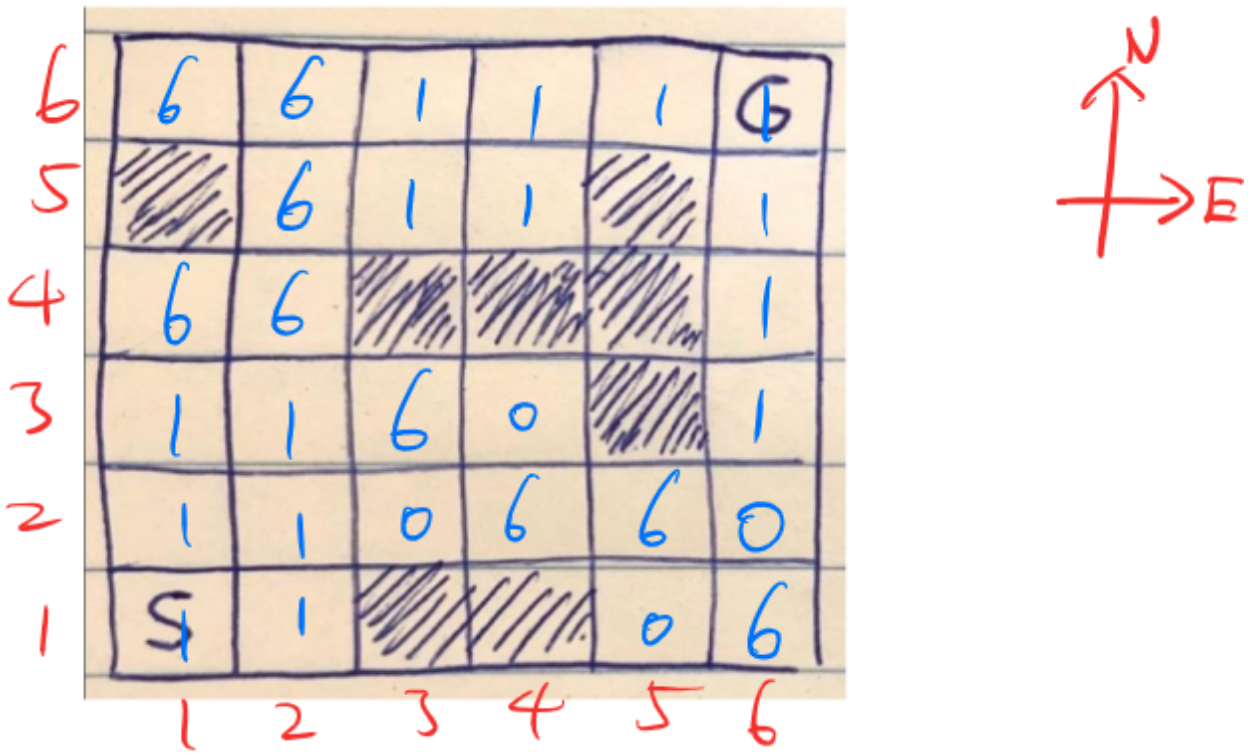
Here's the output with no diagonal motion:



3.2 6x6 map result

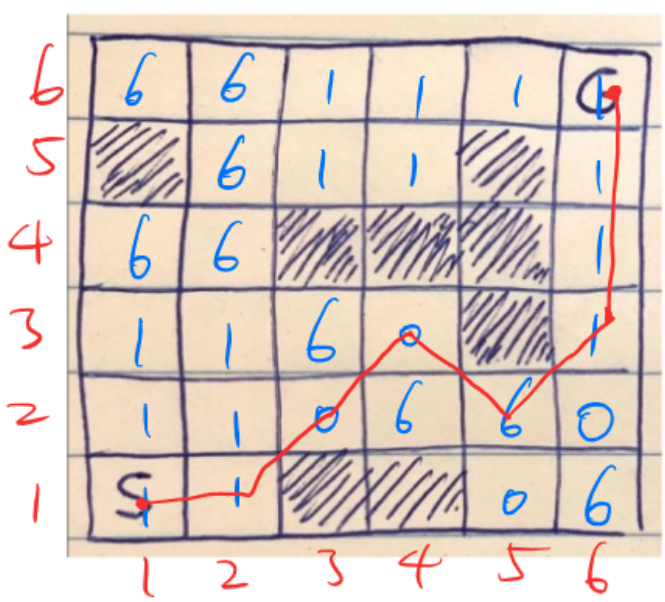
3.3

In autonomous robotics, a costmap is a basic concept. It denotes the expense (difficulty) of traversing certain parts of the map. In the case of a ground robot working on uneven terrain, the cost map could be a map with lower values where the ground is level and larger values where the ground is uneven and sloping. Given the terrain constraints, the following is the new costmap:



3.3 manually modified cost map

Since there are deep puddles in the NW region, the cells in that general area have been raised to costs of 6. Since the SE region has very rough terrain, some cells in the region have a cost of 6 and some have a cost of 0. Since diagonal motions are allowed the path with the least cost would be the following:



With the consideration of path costs (1 for vertical or horizontal direction, $1 \times \sqrt{2}$ for diagonal direction), this path has the least cost. Here's the Matlab implementation:

Editor - HW2_3_3_Dstar_6x6.m Variables - c

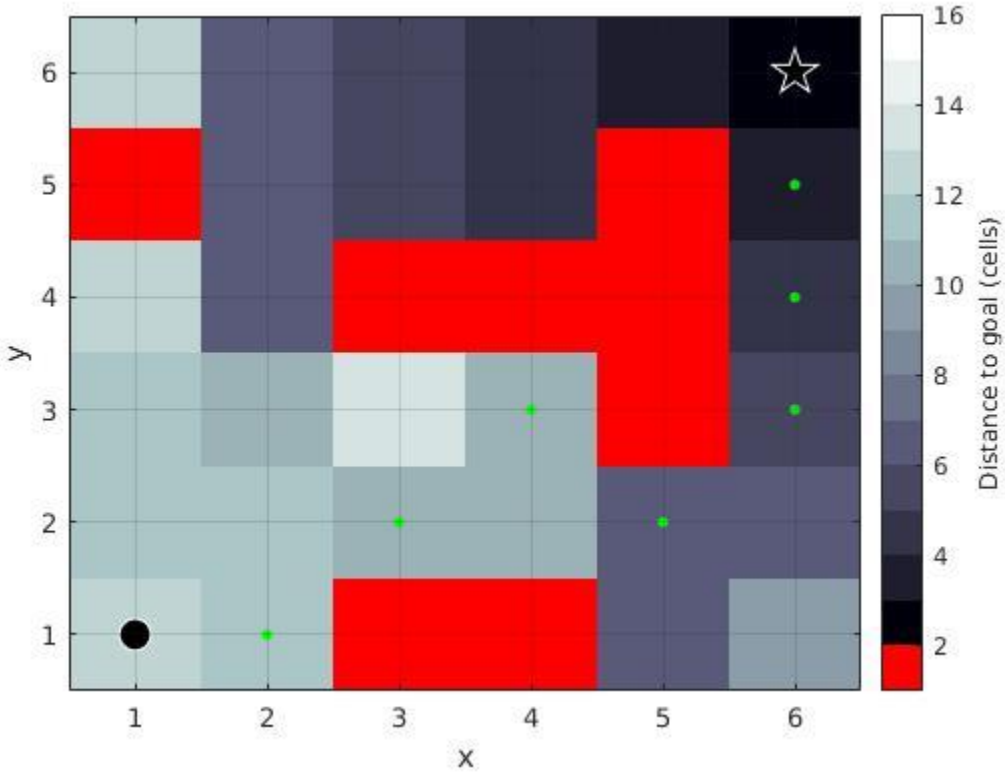
HW2_3_3_Dstar_6x6.m | HW2_4_2_Dstar_80x80.m | HW2_4_3_Voronoi_80x80.m

```
load map6x6

ds = Dstar(map6x6);
c = ds.costmap();
ds.plan([6, 6]);

ds.modify_cost([3; 3], 6);
ds.modify_cost([4; 2], 6);
ds.modify_cost([3; 2], 0);
ds.modify_cost([4; 3], 0);
ds.modify_cost([5; 2], 6);
ds.modify_cost([6; 1], 6);
ds.modify_cost([5; 1], 0);
ds.modify_cost([6; 2], 0);

ds.modify_cost([1; 4], 6);
ds.modify_cost([2; 4], 6);
ds.modify_cost([2; 5], 6);
ds.modify_cost([1; 6], 6);
ds.modify_cost([2; 6], 6);


% ds.modify_cost([3, 4; 2, 3], 12);
% ds.modify_cost([5, 6; 1, 2], 1);
% ds.modify_cost([1, 2; 4, 5], 24);
% ds.modify_cost([1, 2; 5, 6], 24);

% ds.modify_cost([3; 2], 12);


c = ds.costmap();
ds.plan();


% ds.query([1, 1], 'animate');
ds.plot(ds.query([1, 1]));
```

→ rough terrain in SE region

→ deep puddles in NW region

And the resulting path:



4) In this problem we created our own custom map with obstacles and performed various commands and navigation techniques.

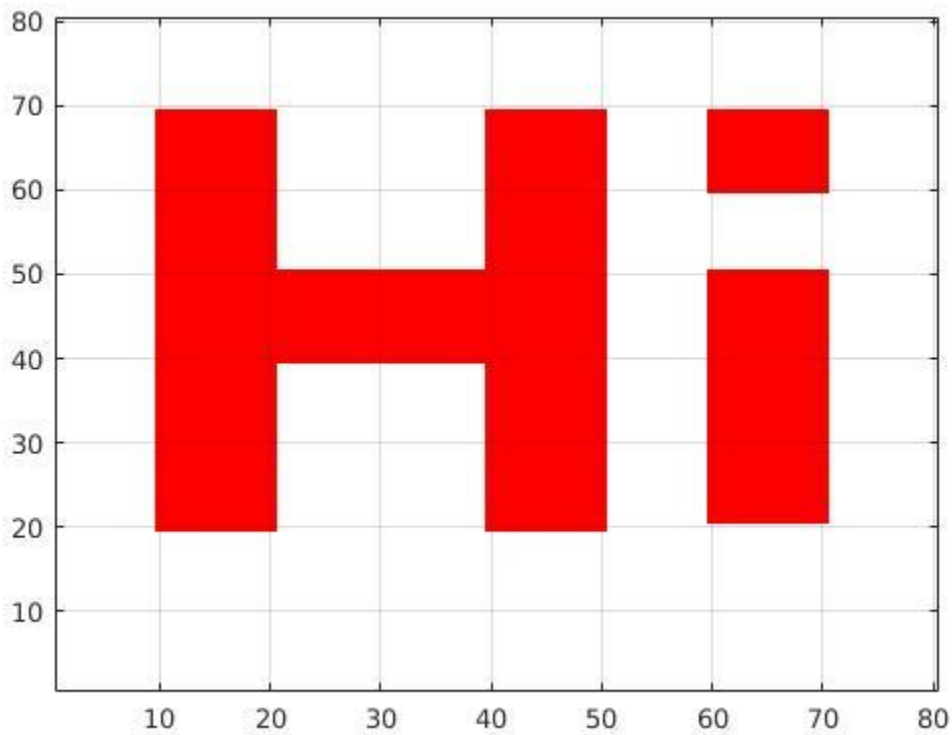
4.1 & 2

First, we used makemap(80) to create an 80x80 map:

```
makemap(80)
```

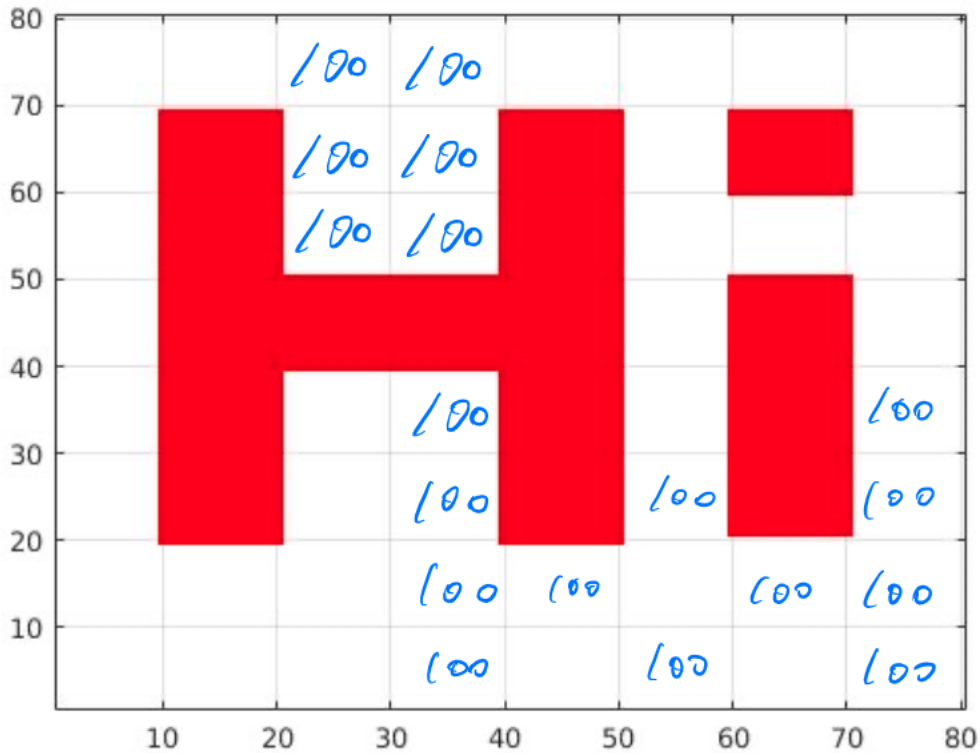
Figure: Creating a custom map

And this is the map that we've created:



We saved the workspace and named it map80x80. Next, we loaded map80x80 and set the start position at (5, 5) and goal position at (65, 55). We ran into many problems when trying to use Dstar, because originally the line was set to Dstar(map), which kept throwing errors. We found that we were trying to move the robot out of bounds. This is because Dstar(map) would create a new 6x6 map, and so when we called the robot to plan for the goal in ds.plan(goal), the

goal would be out of bounds since it is a 80x80 map. Next, we set the special costmap as follows:



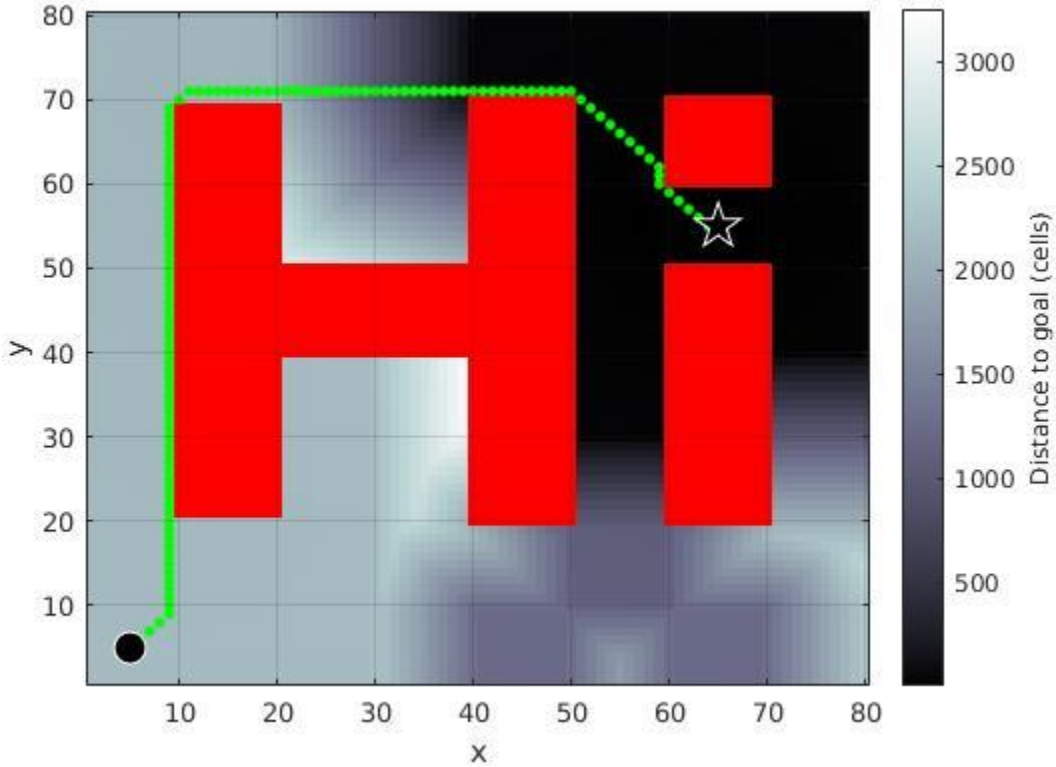
Note that the value 100 in each grid means that each cell within each of the 10x10 grid has the cost value of 100. Then we implemented D* algorithm as follows:

Editor - /home/wen-chung/MATLAB-Drive/Robotics_Application/HW2_4_2_Dstar_80x80.m

HW2_4_2_Dstar_80x80.m | HW2_4_3_Voronoi_80x80.m | HW2_4_3_Show_Voronoi_80x80.m

```
ds = Dstar(map80x80);
c = ds.costmap();

% make plan for goal
ds.plan([65, 55]);

% set some interesting costs
ds.modify_cost([30, 40; 1, 40], 100);
ds.modify_cost([40, 50; 10, 20], 100);
ds.modify_cost([50, 60; 20, 30], 100);
ds.modify_cost([50, 60; 1, 10], 100);
ds.modify_cost([60, 70; 10, 20], 100);
ds.modify_cost([70, 80; 1, 40], 100);

ds.modify_cost([20, 40; 50, 80], 100);



% update cost map
c = ds.costmap();
ds.plan();


% plot from a starting point
ds.plot(ds.query([5, 5]));
```

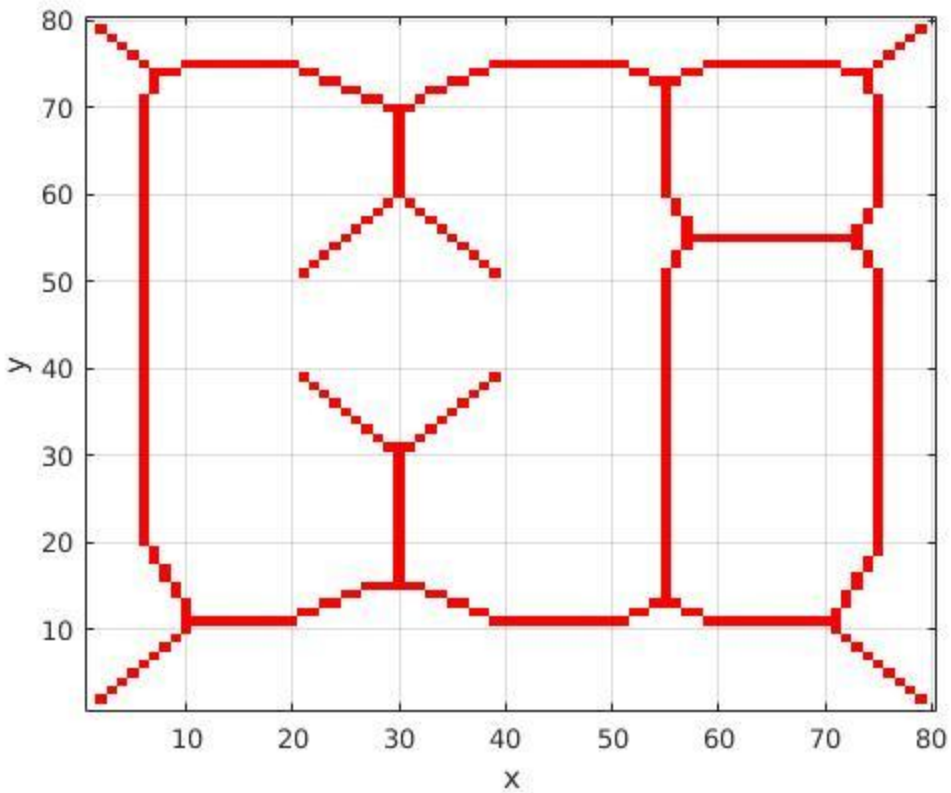
And we got the following path with the costmap overlaid on top:



4.3

We will use the Voronoi diagram as a method for generating a mobile robot's path that dramatically minimizes the likelihood of the robot actually colliding with an obstacle. The graph produced by the boundaries of specially-constructed cells forms the Voronoi diagram for a collection of provided points called sites. The basic idea behind the graph-representation approach is to connect all of the available free spaces in the field (i.e. places where there are no obstacles) via connected networks of lines so that the robot can perform a safe, target-oriented, collision-free motion, where the robot's trajectories are planned.

Using methods explained in section 5.2.3 of the textbook, we generated a free space network for our map:



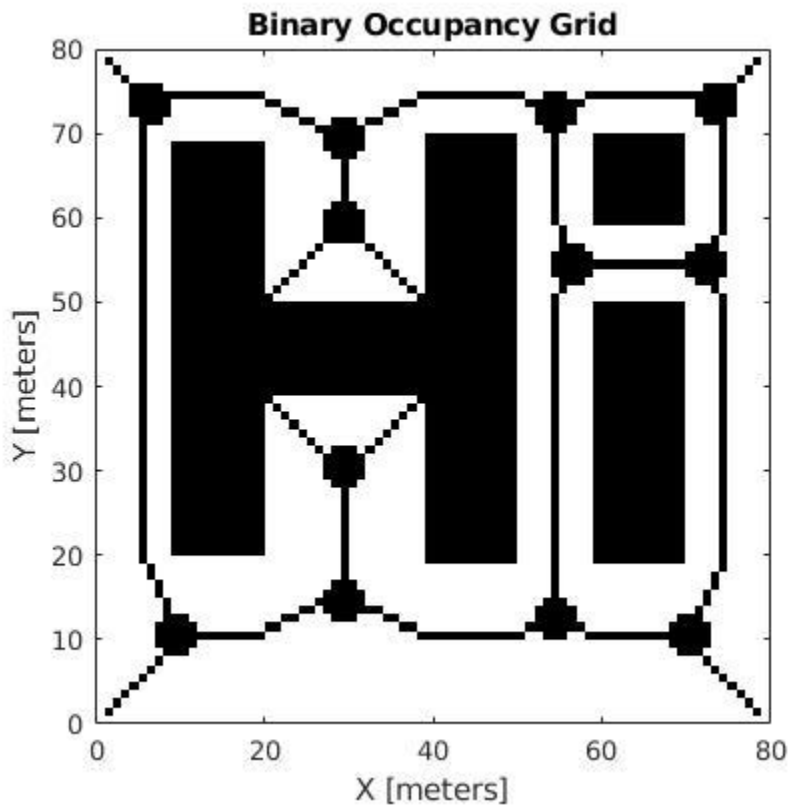
We then included the network joints and overlaid them on top of our map using the following implementation in Matlab:

Editor - /home/wen-chung/MATLAB-Drive/Robotics_Application/HW2_4_3_Show_Voronoi_80x80.m

HW2_4_2_Dstar_80x80.m | HW2_4_3_Voronoi_80x80.m | HW2_4_3_Show_Voronoi_80x80.m

```
load map80x80.mat

% create free space network
free = 1 - map80x80;
free(1,:) = 0;
free(end,:) = 0;
free(:,1) = 0;
free(:,end) = 0;
skeleton = ithin(free);

% create network juncture
joins = itriplepoint(skeleton);
ends = iendpoint(skeleton);

% convert the joints to a binary occ map
joinsbin = binaryOccupancyMap(joins);
% enlarge so they can be seen
inflate(joinsbin,2);
joinsMag = occupancyMatrix(joinsbin);

% combine the free space network, network joints, and obstacles into 1 map
super = map80x80 + skeleton + joinsMag;
super = flipud(super);
superbin = binaryOccupancyMap(super);
show(superbin);
```

And we got the following free space network overview along with our map:

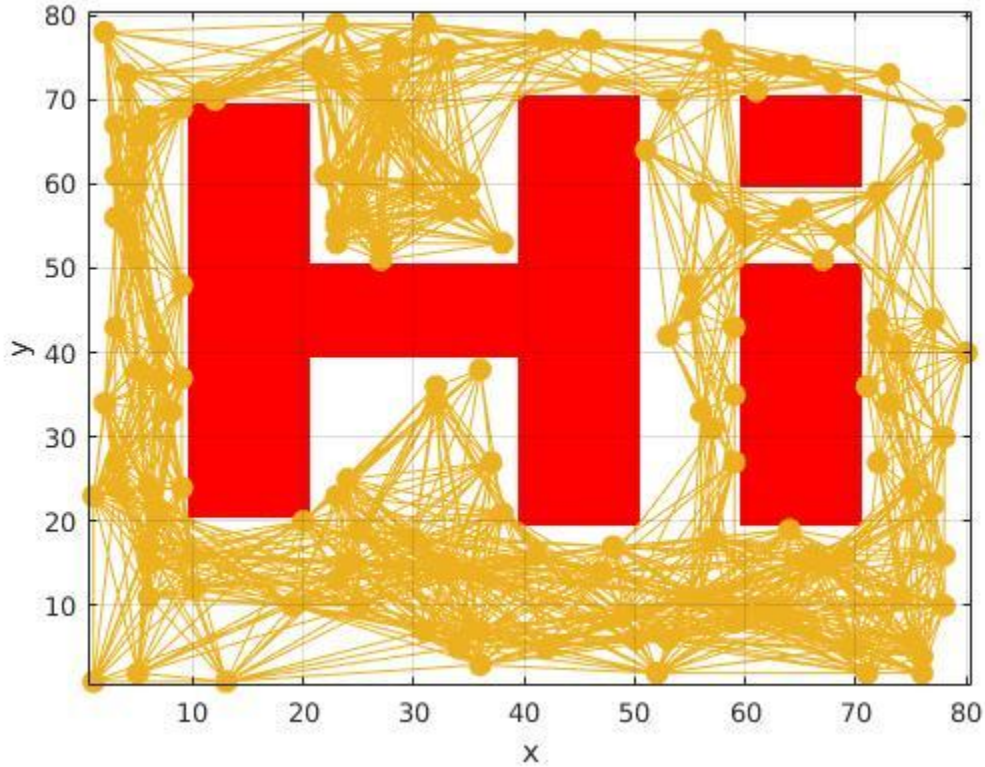


4.4

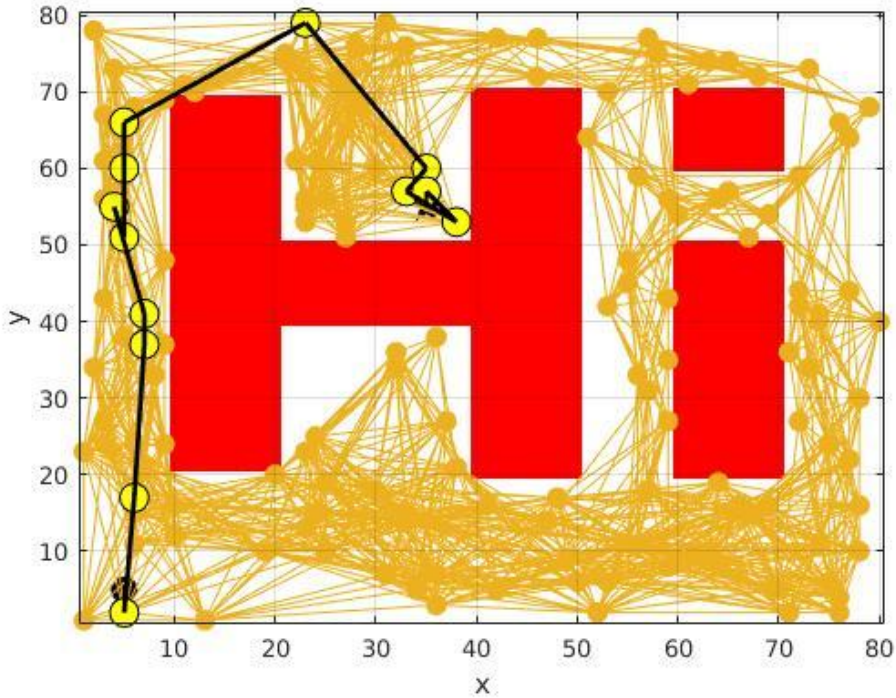
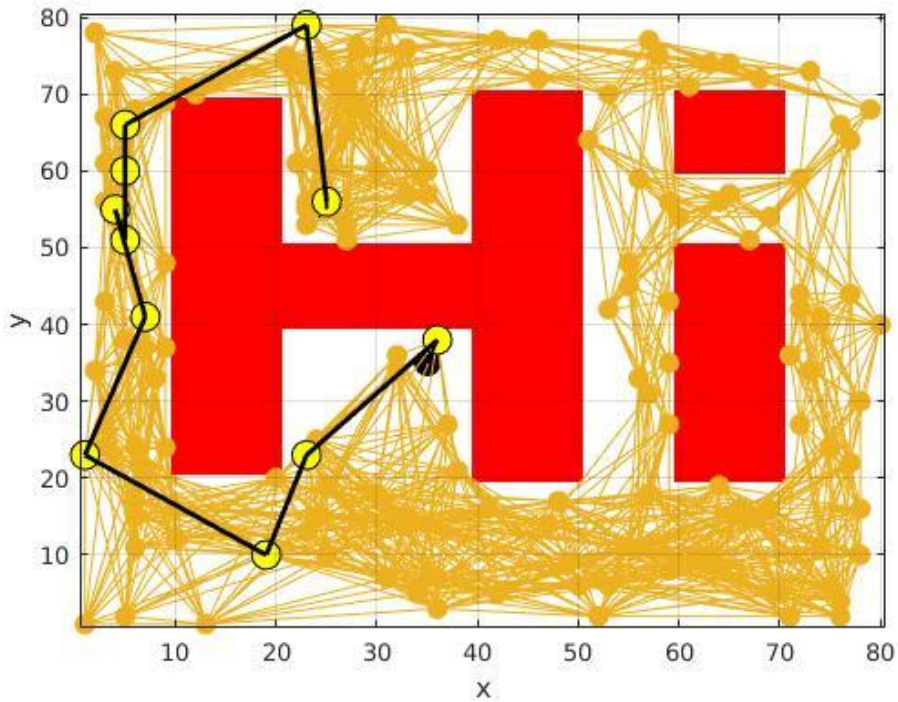
A probabilistic roadmap (PRM) is a network graph based on free and occupied spaces that shows all feasible pathways in a given map. It captures the connectivity of free space with a modest number of points, milestones, and paths is usually enough. This assumption has the potential to make the planning process go much more quickly. In the following Matlab implementation, we used a network of 150 nodes:

```
% prm = PRM(skeleton);
prm = PRM(map80x80);
prm.plan('npoints', 150);
figure()
prm.plot();
% use first set of starting and goal points
prm.query([5, 5],[35, 55]);
figure()
prm.plot();
% use second set of starting and goal points
prm.query([35, 35], [25, 55]);
figure()
prm.plot();
```

We got the following roadmap for our map:



Using the first set of starting (5, 5) and goal points (35, 55) we got the following path:



Using the second set of starting (35, 35) and goal points (25, 55) we got the following path: