Dov Cattan: COT 4930 HOMEWORK 2

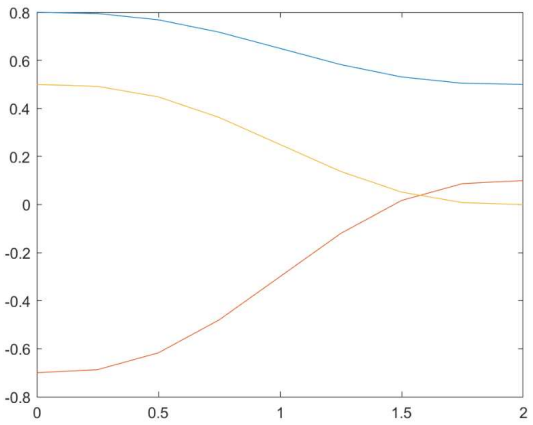
1.1

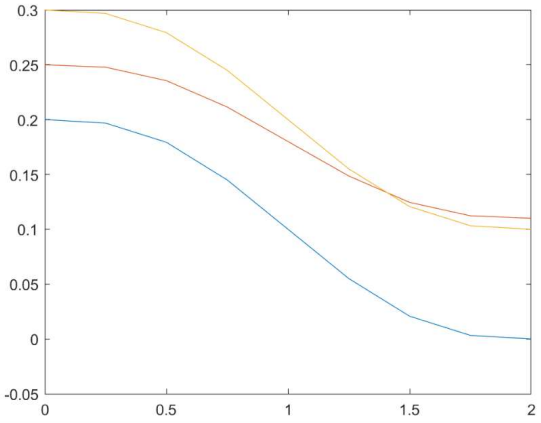
Code:

The tr2py functions are using to rotate the original matrices to their goal position on the graph, along with their rotation vectors found on the last column of each 4x4 rotation matrix

```
% Initial Positions
A = [0.9256 -0.2427 0.2904;
     0.2863 0.9508 -0.1181;
    -0.2474 0.1925 0.9496];


B = [0.9890 -0.0998 0.1092;
     0.0992 0.9950 0.0110;
    -0.1098 0 0.9940];
% Time
Time = 0:0.25:2
% Anglular Functions
angleA = tr2rpy(A)
angleB = tr2rpy(B)

% Rotation Vectors
rot_vectorA = [0.8000,-0.7000,0.5000];
rot_vectorB = [0.5000,0.1000,0]

% Trajectories
p1 = mtraj(@tpoly,rot_vectorA, rot_vectorB, Time)
p2 = mtraj(@tpoly,angleA, angleB,Time)

% Plot Trajectories
plot(Time, p1)
plot(Time, p2)
```

Output and Graph:






```
Time = 1×9
           0    0.2500    0.5000    0.7500    1.0000 ...

angleA = 1×3
      0.2000    0.2500    0.3000

angleB = 1×3
           0    0.1100    0.1000

rot_vectorB = 1×3
      0.5000    0.1000         0

p1 = 9×3
      0.8000   -0.7000    0.5000
      0.7952   -0.6872    0.4920
      0.7689   -0.6172    0.4482
      0.7174   -0.4798    0.3624
      0.6500   -0.3000    0.2500
      0.5826   -0.1202    0.1376
      0.5311    0.0172    0.0518
      0.5048    0.0872    0.0080
      0.5000    0.1000         0

p2 = 9×3
      0.2000    0.2500    0.3000
      0.1968    0.2477    0.2968
      0.1793    0.2355    0.2793
      0.1450    0.2115    0.2449
      0.1000    0.1800    0.2000
      0.0550    0.1485    0.1550
      0.0207    0.1245    0.1207
      0.0032    0.1123    0.1032
     -0.0000    0.1100    0.1000
```

1.2

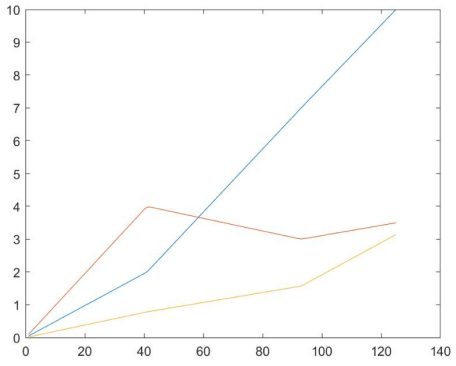
Code:

Instead of using MATLAB functions, using rotation matrices are now necessary for rotations to happen

```
% I is initial position
I =[0, 0, 0]
% Translation From Initial Position
transl = [2 4 (pi/4);,7 3 (pi/2); 10 3.5 pi]
% Speed Of Vector
vectorSpeed = 0.1
% Create Trajectory and Plot Trajectory
traj = mstraj(transl,vectorSpeed,[],I,1,2)
plot(traj)
```

Output and Graph:

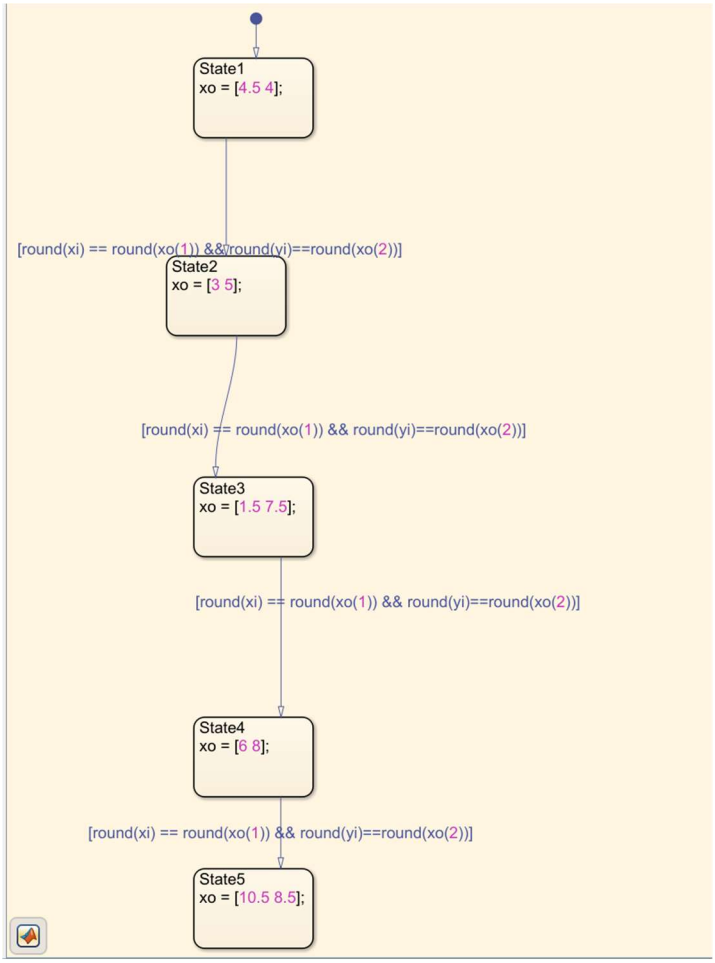
```
I = 1×3
       0     0     0

transl = 3×3
      2.0000    4.0000    0.7854
      7.0000    3.0000    1.5708
     10.0000    3.5000    3.1416

vectorSpeed = 0.1000

traj = 125×3
      0.0488    0.0976    0.0192
      0.0976    0.1951    0.0383
      0.1463    0.2927    0.0575
      0.1951    0.3902    0.0766
      0.2439    0.4878    0.0958
      0.2927    0.5854    0.1149
      0.3415    0.6829    0.1341
      0.3902    0.7805    0.1532
      0.4390    0.8780    0.1724
      0.4878    0.9756    0.1916
        ⋮
```



2.1

Transition State Chart

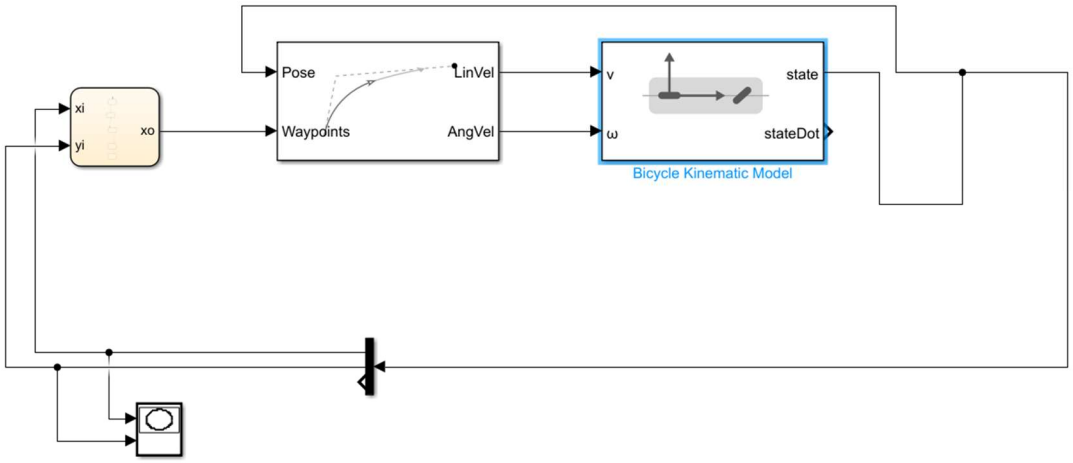
From initial point [4.5 2], the States represent where the bicycle robot will be navigating within the corridor. The round function rounds both x and y inputs to the next state for the robot to transition properly to the next state.



Simulink Model for Bicycle Style Robot:

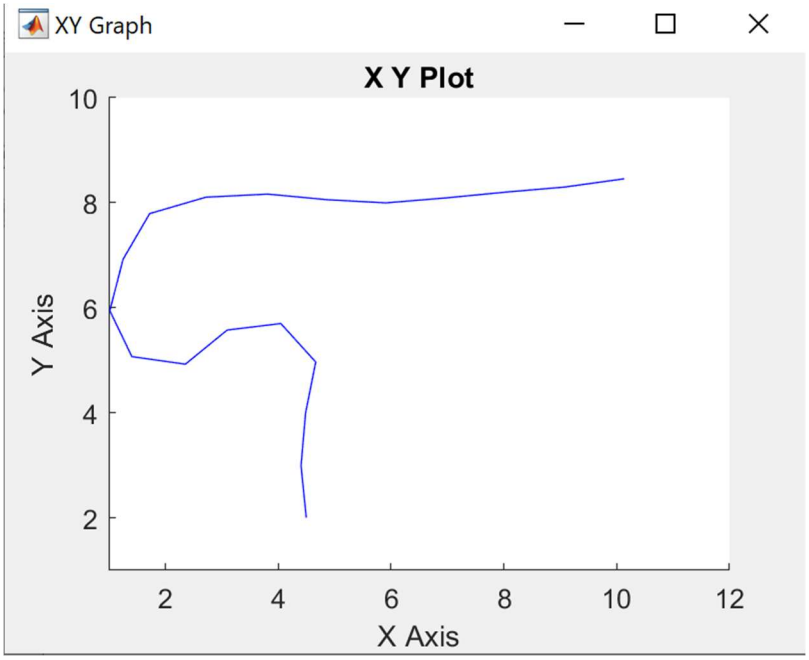
From the chart, we connect to the waypoints of our pure pursuit, that is intertwined with our pose, then they output both a linear and angular velocity to the Bicycle Kinematic Model to emulate the robot. The output of Model aka State creates its pose and inputs for the next waypoint

transition using the Demux. Those inputs are also outputted using an X/Y Graph.



Stop Time: 3 minutes/180 Seconds as the bicycle can get lose trajectory before it reaches the goal if done at any higher time or stop midway and a go off trajectory midway if done at a lower time.

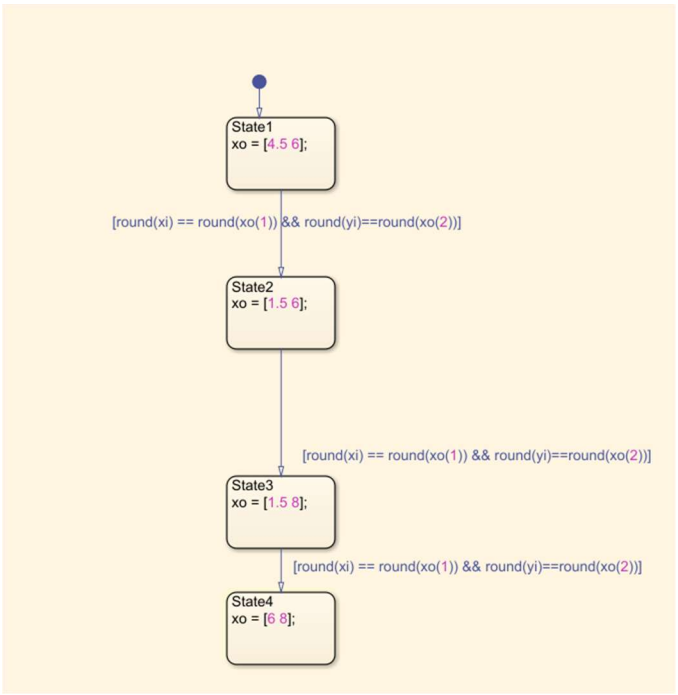
Output/Navigation Graph:



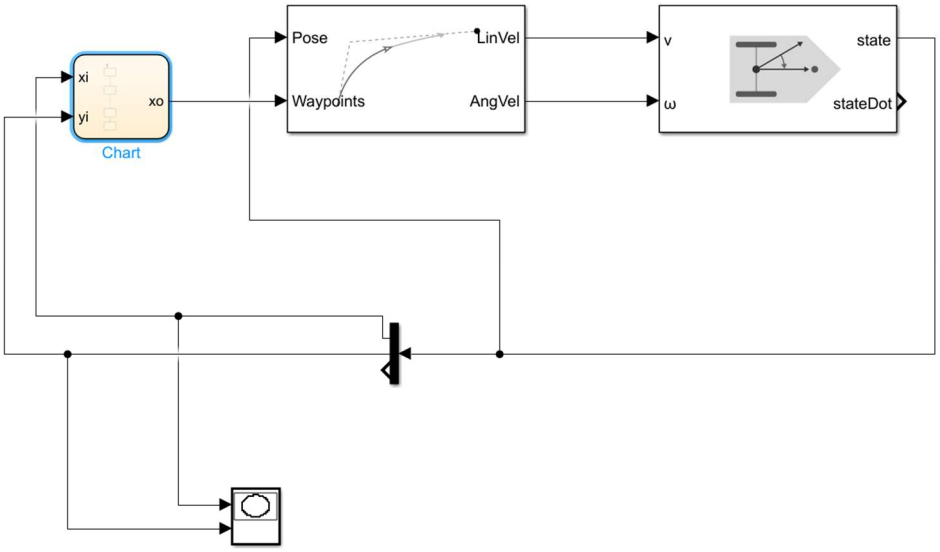
2.2

Transition State Chart

From initial point [4.5 2], the States represent where the differential drive robot will be navigating within the corridor. The round function rounds both x and y inputs to the next state for the robot to transition properly to the next state. Unlike the previous robot, this only changes x and y axis at a time for it to truly emulate the differential drive robot, unlike the bicycle robot where the x and y can change simultaneously.

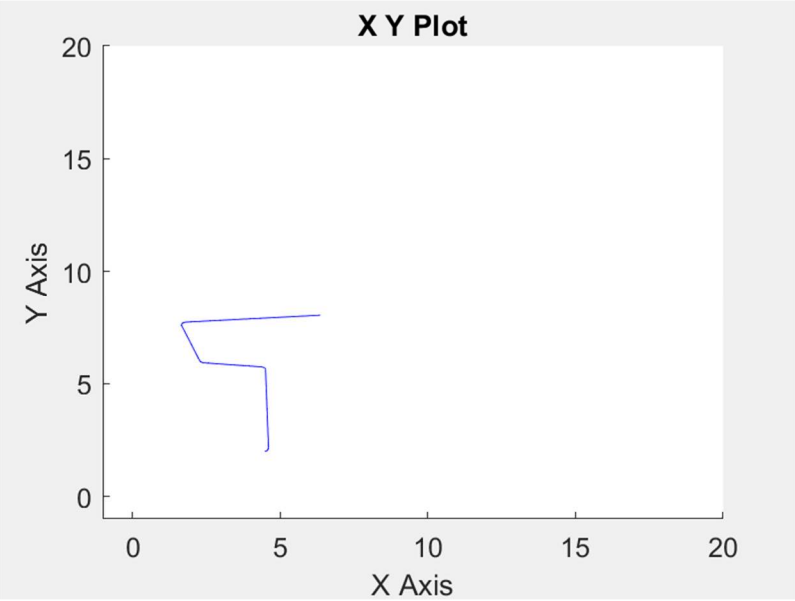


From the chart, we connect to the waypoints of our pure pursuit, that is intertwined with our pose, then they output both a linear and angular velocity to the Differential Drive Kinematic Model to emulate the robot. The output of Model aka State creates its pose and inputs for the next waypoint transition using the Demux. Those inputs are also outputted using an X/Y Graph.



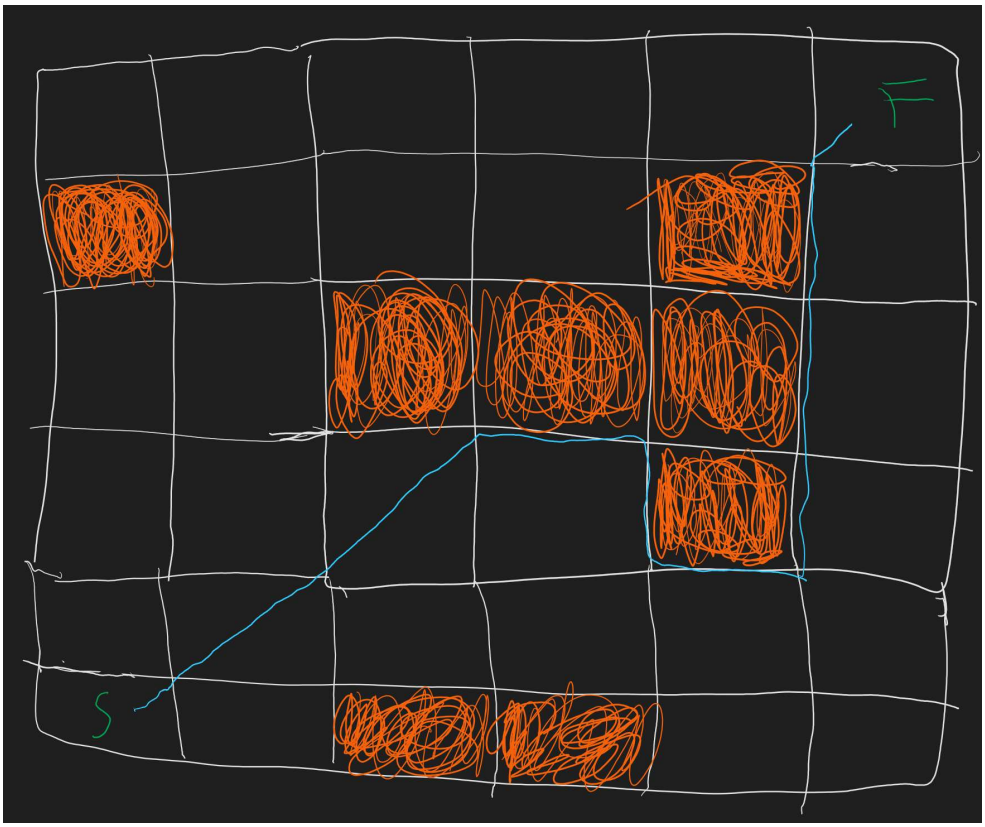
Stop Time: 2 minutes and 5 seconds/125 Seconds as it would loop in the end if put at any higher time, or unfinish at any lower time.

Output/Navigation Graph:



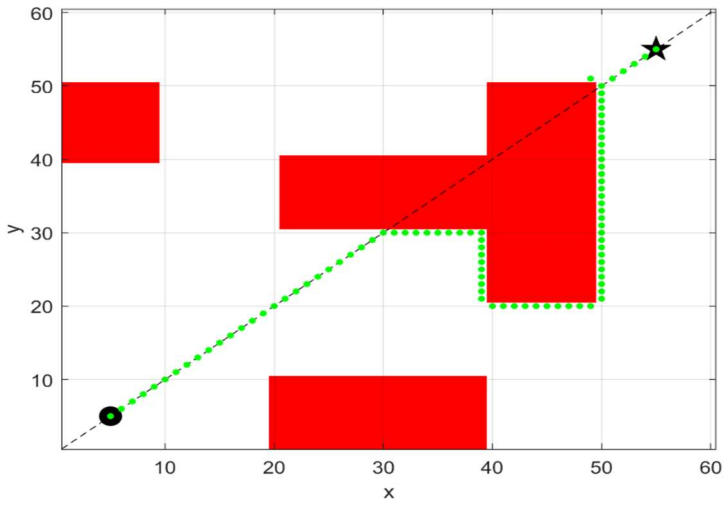
3.1

Manual Bug 2 Algorithm Map



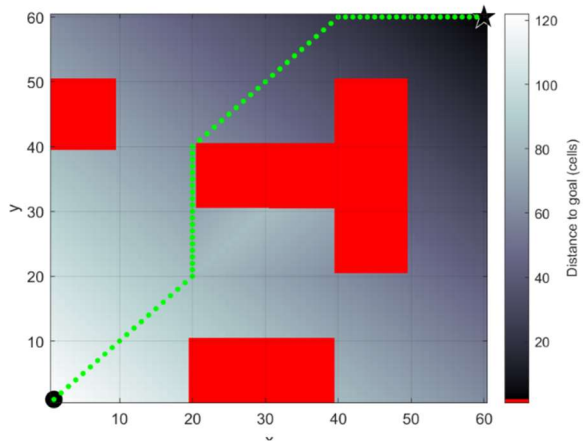
MATLAB Bug 2 Algorithm Map

```
Algo =
Bug2 navigation class:
  occupancy grid: 60x60
```



3.2

```
map = 6×6
     0     0     0     0     0     0
     1     0     0     0     1     0
     0     0     1     1     1     0
     0     0     0     0     1     0
     0     0     0     0     0     0
     0     0     1     1     0     0

start = 1×2
     1     1

stop = 1×2
    60    60


City =
DXform navigation class:
  occupancy grid: 60x60
  distance metric: cityblock
  distancemap: empty:
```

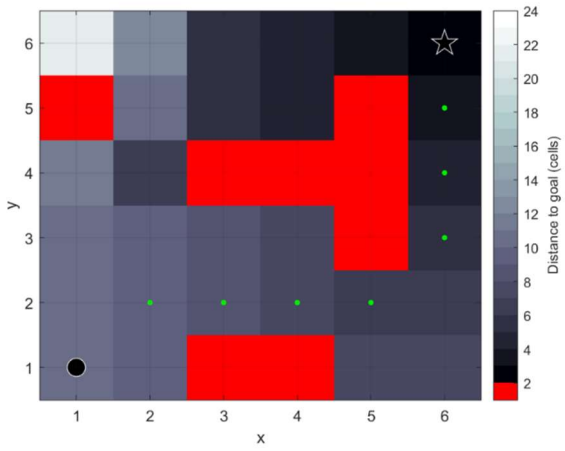



```
% Map Setup
map= zeros(6,6)
map(2,1)=1
map(2,5)=1
map(3,3)=1
map(3,4)=1
map(3,5)=1
map(4,5)=1
map(6,3)=1
map(6,4)=1
% Starting/ Goal Point Setup
start = [1 1]
stop = [60 60]
City = DXform(world,'cityblock')
City.plan(stop)
City.query(start,'animate')
```

Bug2 Algorithm Changed Due to City Structure

3.3

```
27 iterations
p = 10×2
     1     1
     2     2
     3     2
     4     2
     5     2
     6     3
     6     4
     6     5
     6     6
     6     6
```




```
% Map Setup
map= zeros(6,6)
map(1,3:4)=1
map(3,5)=1
map(4,3:5)=1
map(5,1)=1
map(5,5)=1
% Start to Goal Point Setup
start = [1 1]
stop = [6 6]
StarD= Dstar(map)
StarD.costmap()
StarD.plan(stop)
figure(4),StarD.query(start,'animate')

% Terrain Setup
r_terrain1 = [5;1]
r_terrain2 = [1;6]
r_terrain3 = [2;4]
r_terrain4 = [2;5]
r_terrain5 = [2;6]
r_terrain6 = [3;4]
r_terrain7 = [3;6]
StarD.modify_cost(r_terrain1,9)
StarD.modify_cost(r_terrain2,9)
StarD.modify_cost(r_terrain3,9)
StarD.modify_cost(r_terrain4,9)
StarD.modify_cost(r_terrain5,9)
StarD.modify_cost(r_terrain6,9)
StarD.modify_cost(r_terrain7,9)

% Plot Terrain
StarD.plan()
p = StarD.query(start)
StarD.plot(p)
```

Bug2 Algorithm Changed because of City Terrain

4.1

Map Setup Including The makemap that has finished

```
make = 8×8
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
```

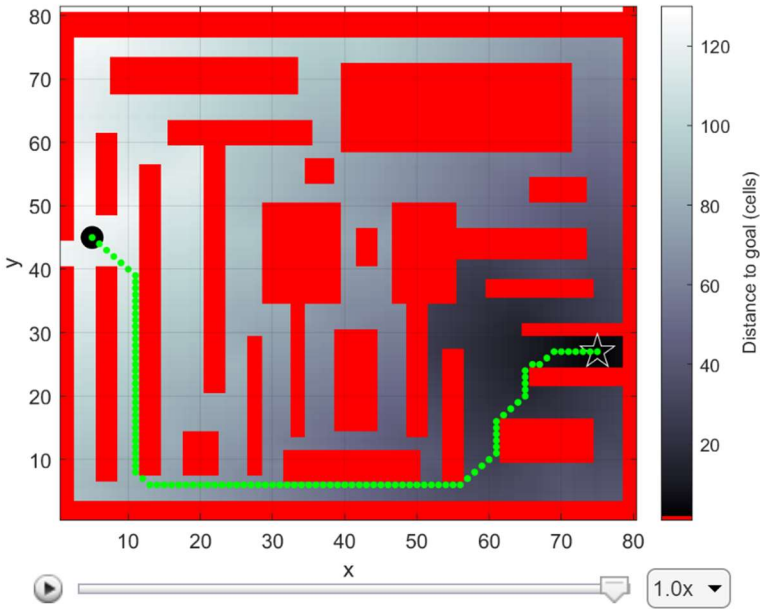
4.1 and 4.2 Code

Both Starting and End Points Made plus crash points planned for the robot to navigate in the city made

```
load 4.1.mat
%4.1
make = zeros(8,8)
%makemap(80)

%4.2
start = [5 45]
stop = [75 27]
ds = Dstar(world)
ds.costmap()
ds.plan(stop)

%crash
crash1 = [15;40]
crash2 = [25;10]
crash3 = [35;10]
crash4 = [72;36]
ds.modify_cost(crash1,20)
ds.modify_cost(crash2,10)
ds.modify_cost(crash3,9)
ds.modify_cost(crash4,7)
ds.plan()
figure(4),ds.query(start,'animate')
```

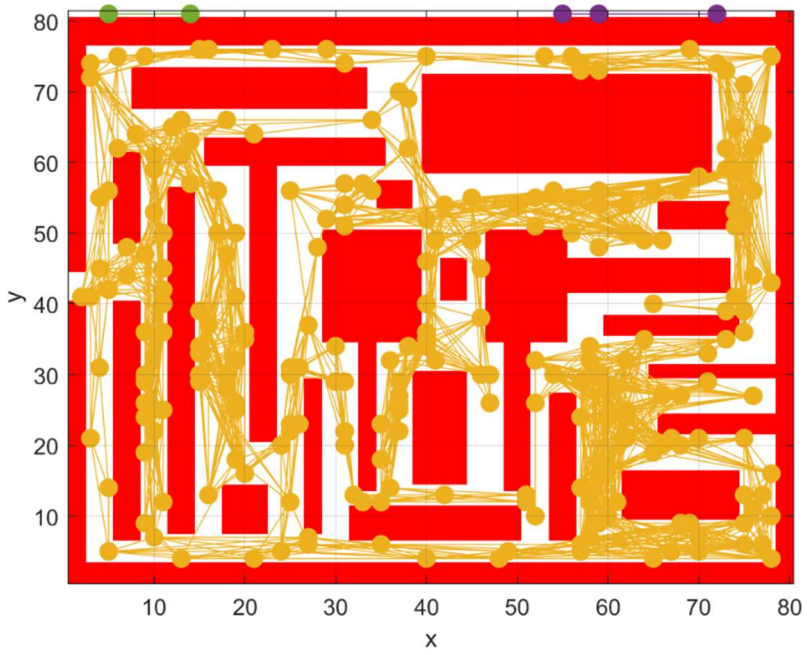
4.2 City/ Costmap Results

```
6557 iterations
crash1 = 2×1
      15
      40

crash2 = 2×1
      25
      10

crash3 = 2×1
      35
      10

crash4 = 2×1
      72
      36

8 iterations
```



4.3 Map Setup for 4.4 Roadmap Navigation Algorithm

```
start = 1×2
       5    42

stop = 1×2
      75    27


prm =
PRM navigation class:
  occupancy grid: 81x80
  graph size:
  dist thresh:
  2 dimensions
  0 vertices
  0 edges
  0 components
```



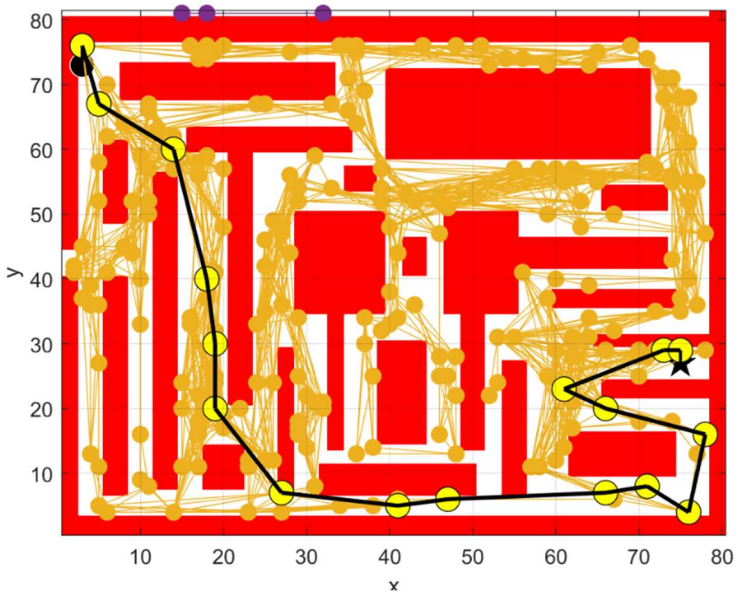
4.3 Setup Code with proposed starting and end points

```
%4.3

start = [5 42]
stop = [75 27]
prm = PRM(world)
prm.plan('npoints',250)
prm.plot()
```

4.4

Roadmap Navigation Map Part 1, with one set of start/finish points

```
start = 1×2
       3    73

stop = 1×2
      75    27


house =
PRM navigation class:
  occupancy grid: 81x80
  graph size:
  dist thresh:
  2 dimensions
  0 vertices
  0 edges
  0 components
```



Code for Part 1

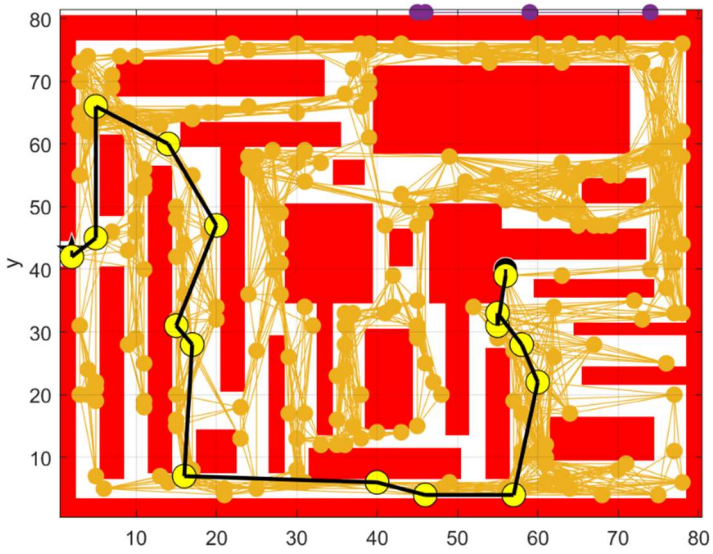
```
%4.4
start = [3 73]
stop = [75 27]
house = PRM(world)
prm.plan('npoints',250)
prm.plot()
prm.query(start,stop)
prm.plot()
```

Roadmap Navigation Map Part 2, with another set of start/finish points

```
start = 1×2
      56    40
```

```
stop = 1×2
       2    43
```

```
house =
PRM navigation class:
  occupancy grid: 81x80
  graph size:
  dist thresh:
  2 dimensions
  0 vertices
  0 edges
  0 components
```



Code for Part 2

```
%4.4
start = [56 40]
stop = [2 43]
house = PRM(world)
prm.plan('npoints',250)
prm.plot()
prm.query(start,stop)
prm.plot()
```