Matthew Daniel
mrd89

Homework 1

MAE 4180 / 5180
2020-Jan-31

Potential Function

1. Please see figure below

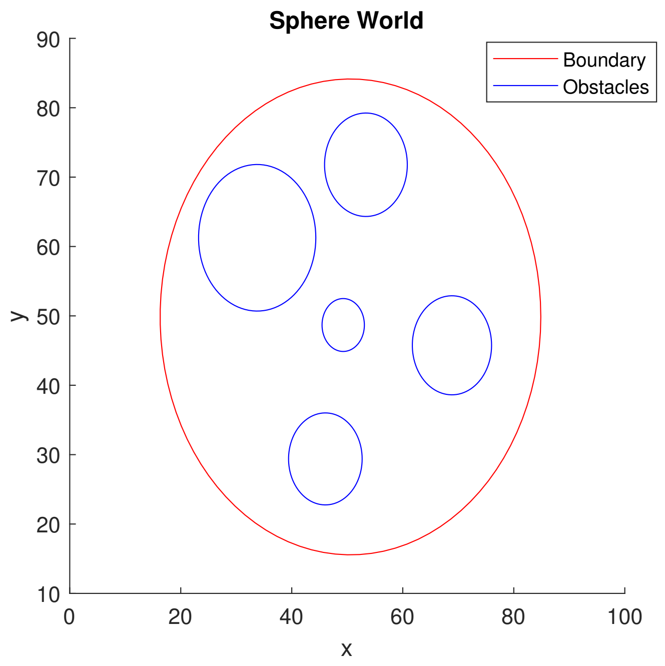

Figure 1: Sphere World Plot

2. Please see code
3. I used a value of 1m for Q for each obstacle. This number makes sense because it allows the robot to get relatively near the obstacles which could be more efficient. Additionally, it is large enough to ensure that the robot with a radius of 0.2m will never run into an obstacle.
4. Please see code
5. I was able to observe the direct affects of changing each of the parameters! The only parameter that did not seem as straight forward was the Q factor. Even as I increased Q by a few meters, I could not *visibly* see a major difference in the plots, but I imageine it makes a much larger difference than I could see.

Values	
C_att	2
C_rep	1
Q	1
maxWeight	200

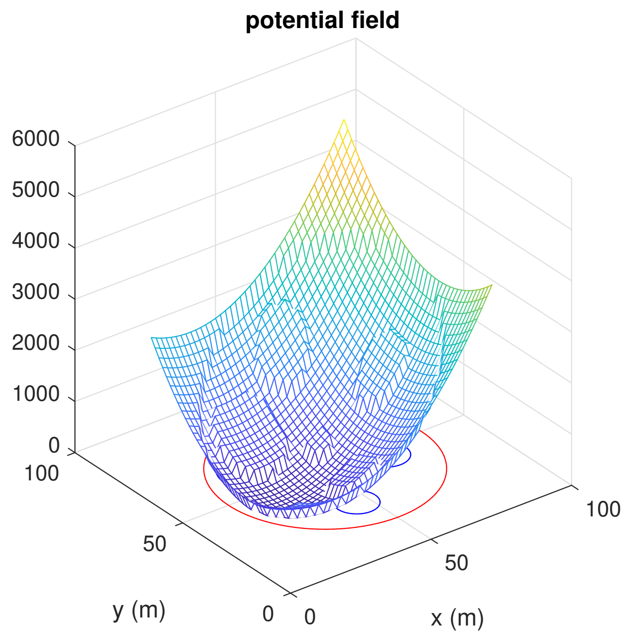

Figure 2: Potential Field of Map

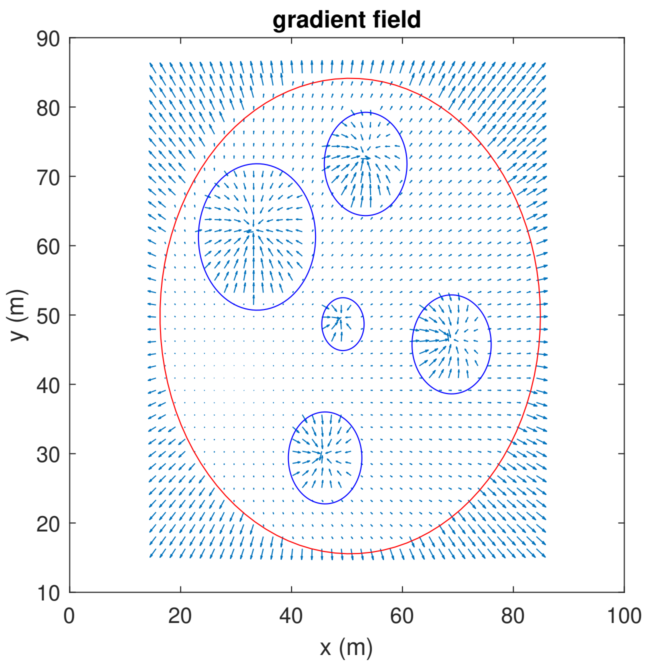

Figure 3: Gradient Field of Map

6. Please see Figures below for the local minimum plots. It can be slightly hard to see from these plots, but while zoomed in there is a noticeable local minimum around 50,50 in between the large and smaller obstacle. I was able to run this in the simulator and found the robot stayed in that position as expected!

Values	
C_att	0.05
C_rep	0.2
Q	10
maxWeight	2

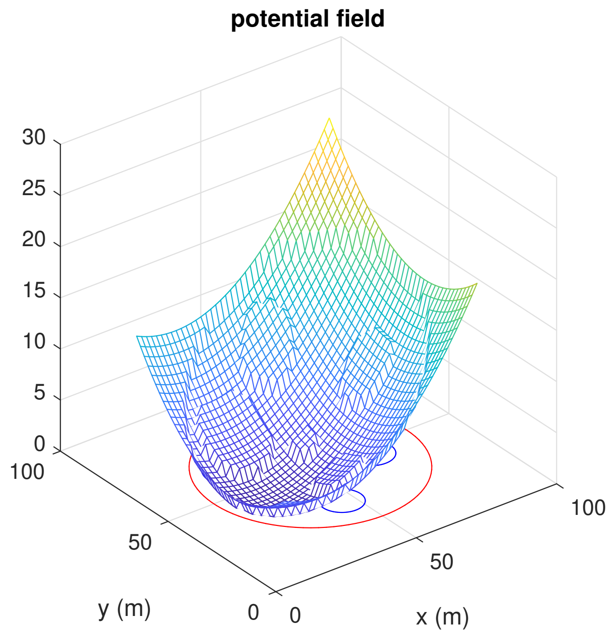

Figure 4: Potential Field of Map

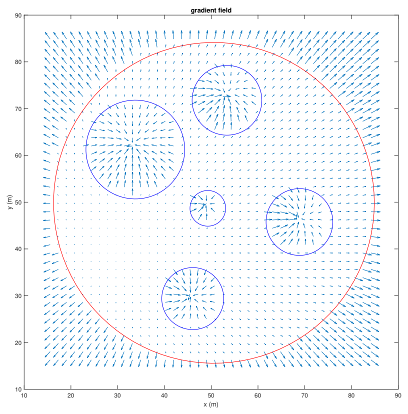

Figure 5: Gradient Field of Map

7. Please see figure below! The robot does not ever touch the obstacles even with radius in account. It took a bit of zooming in to confirm this, but it works as expected!

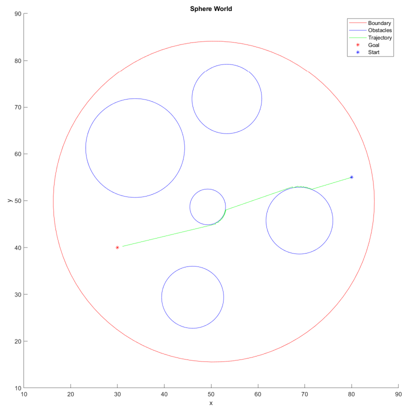

Figure 6: Holonomic Robot Plot

Test Potential Function

Please see Matlab for test function

Controlling the Create using Potential Functions

1. Completed in Matlab

2. For this map as seen below, I found the center point of each square and found the minimum radius to encapsulate the entire square. Additionally, I added a small buffer of .08 to each of the points to ensure that the robot would not accidentally run into the corner. Although this would not happen in the simulation, with the real robot running, there is a long enough delay that might have the robot hitting into the wall.

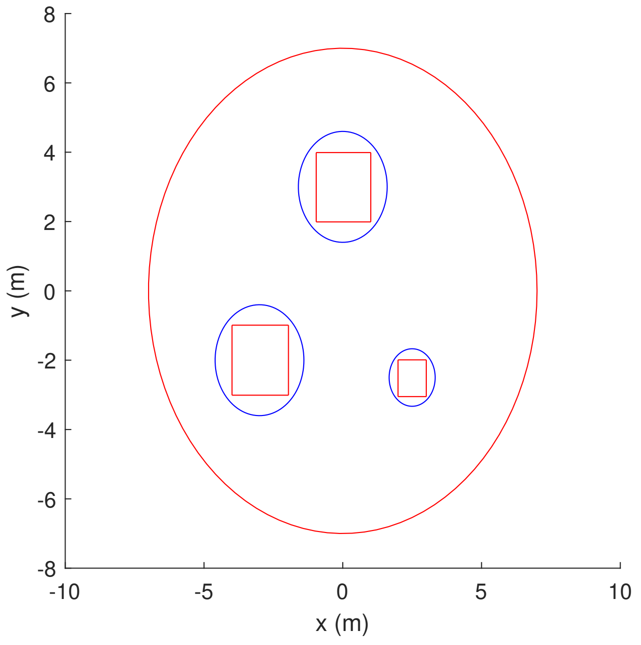

Figure 7: Sphere World Overlay Plot

3. Please see Figures below. The robot never hit any of the obstacles!

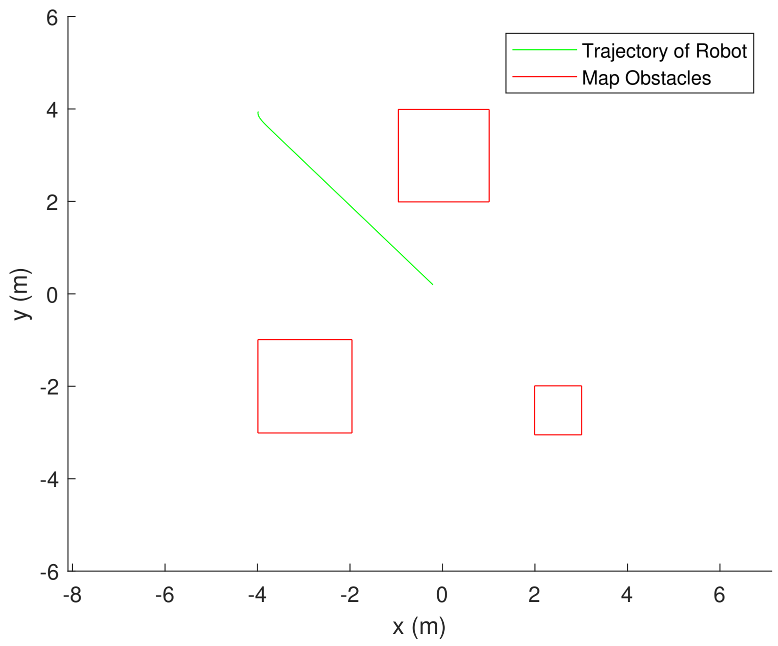

Figure 8: iRobot Create Simulation from [-4,4]

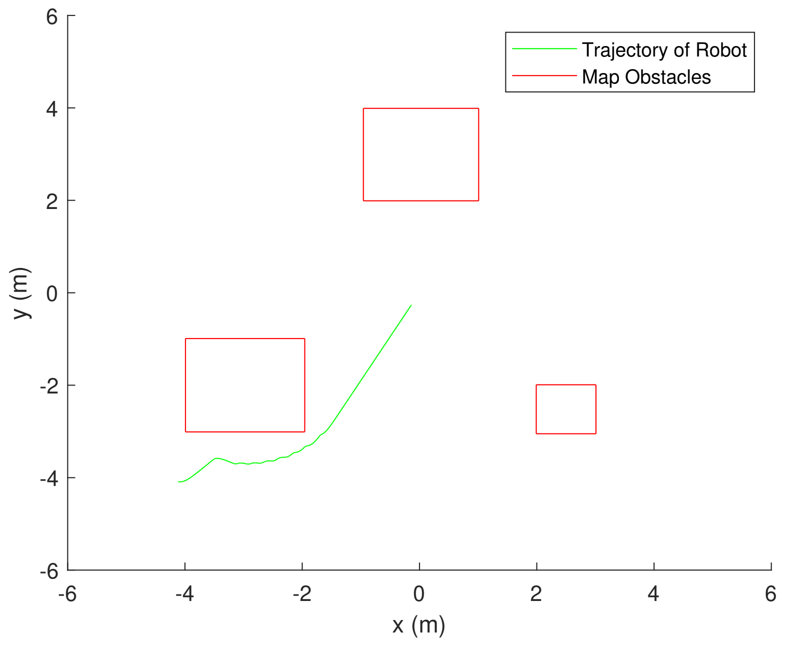

Figure 9: iRobot Create Simulation from [-4,-4]

4. The simplest answer to this is yes if you place them within a boundary as the simulation would not allow the robot to pass through the wall. When done, the robot ends up stuck in the corner of the wall. Without placing it in an obstacle, you can reach a stopping point when the robot corrects itself twice and negates all movement. Essentially the robot is put in a position which requires a fast rotation in one direction and once that occurs, the gradient required the exact opposite movement. Overall it is fairly unlikely that the time steps and movement will exactly cancel out and thus a majority of the time the robot will get to the goal!

Cell Decomposition

1. Please see Figure below

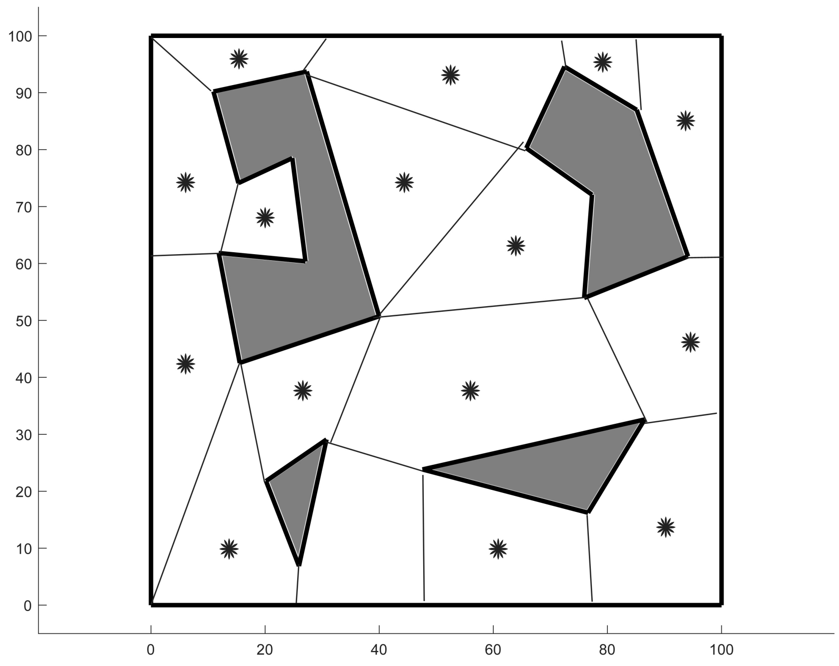

Figure 10: Hand-Made Cell Decomposition

2. Please see figure above!

3. The nodes are the stars in the graph and the edges are all the connecting lines between the cells. The greyed our region which was plotted in Matlab are the occupied cells which are the obstacles.

Roadmap

1. To create the road-map I created a visibility road-map from the polygonal obstacle nodes by connecting each node with a free path to the other. Firstly, I decomposed the map into a N-by-3 matrix where N is the number of nodes. Specifically, each row stores the obstacle number, x coordinate, and y coordinate. I decided to store obstacle number to keep track of the nodes. The road map is stored in a N-by-N matrix where the rows and columns each correspond to the node numbers. If node 1 is connected with another node, node 2, the value of the cell at [node1,node2] and [node 2, node 1] is the euclidean distance between them. If the two are not connected, the value at both indices is 0 to indicate not-connected.

2. Please see Figure below.

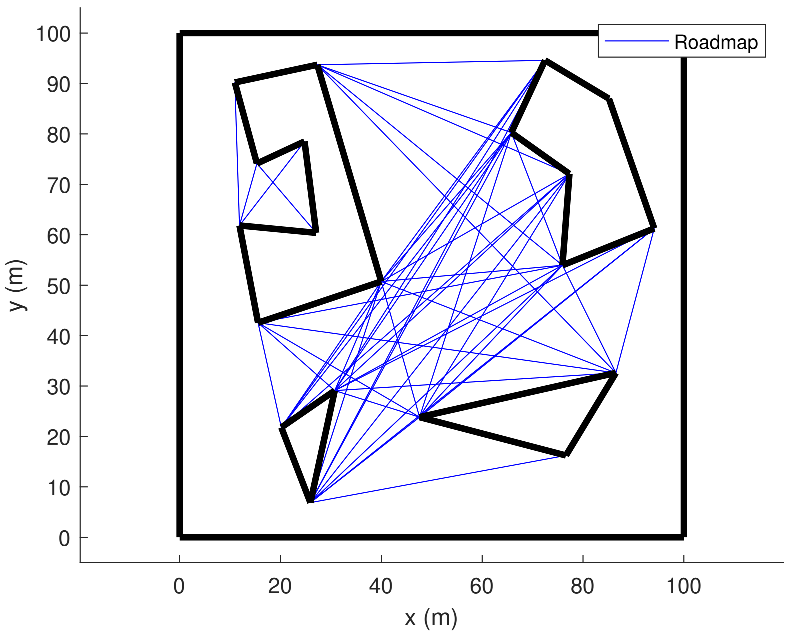

Figure 11: Road-map for Included Map

3. The nodes as briefly mentioned earlier are the vertices of the obstacles. The edges are the path inbetween the nodes.

Paths in the Discrete Abstraction

1. Please see matlab for my function, I used a Dijkstra's algorithm from Matlab Exchange and the file is included in my submission.

Homework 1

2. Please see Figures below.

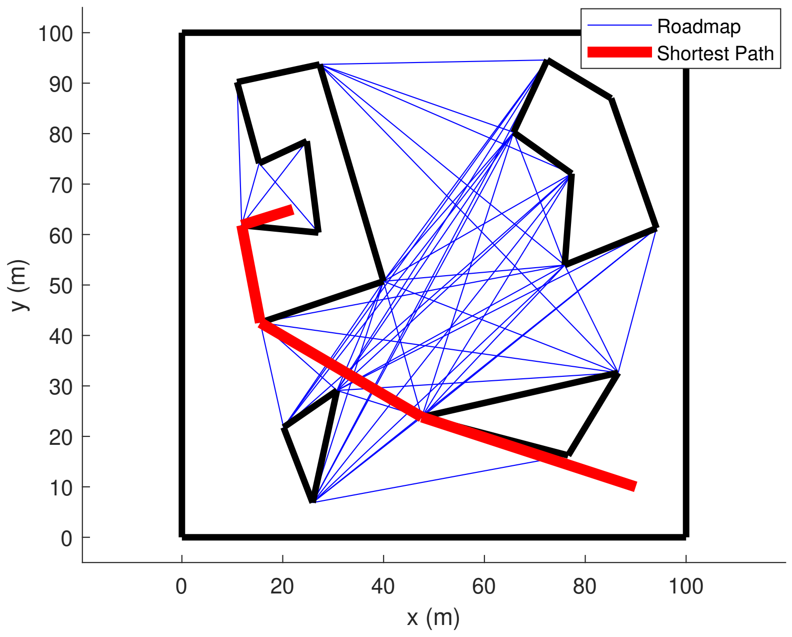

Figure 12: Shorted Path with Road-map Overlay

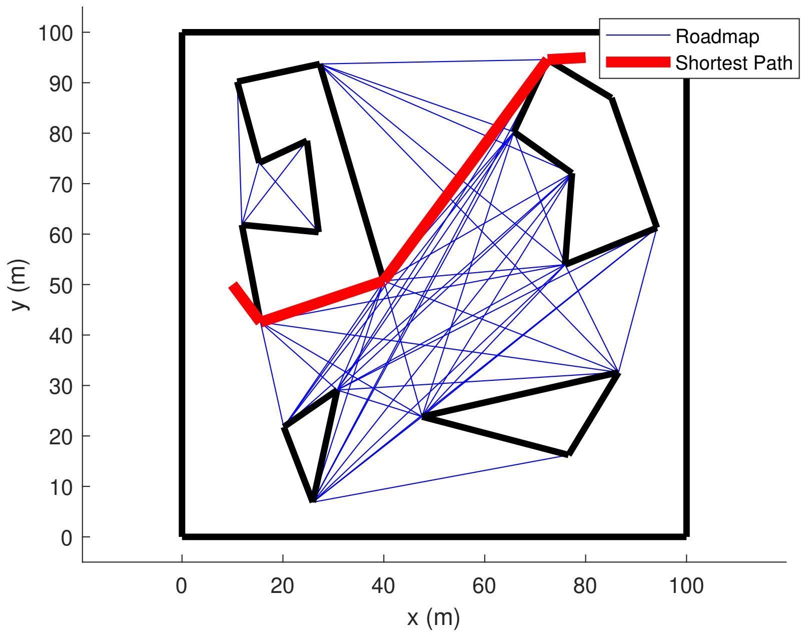

Figure 13: Shorted Path with Road-map Overlay

3. The weight for this is based off of Dijkstra's algorithm. In this case, the weight is based on the distance between the two nodes since we want to find the shortest distance needed to travel from the pose to goal.

Rapidly Exploring Random Trees

1. Please see matlab

2. Please see figures for the maps. We can see that each one is completely random which means some run times are excellent and some not. Additionally the path is not the shortest path typically due to the random nature. To ensure faster path, there are other smoothing algorithms which I did not implement.

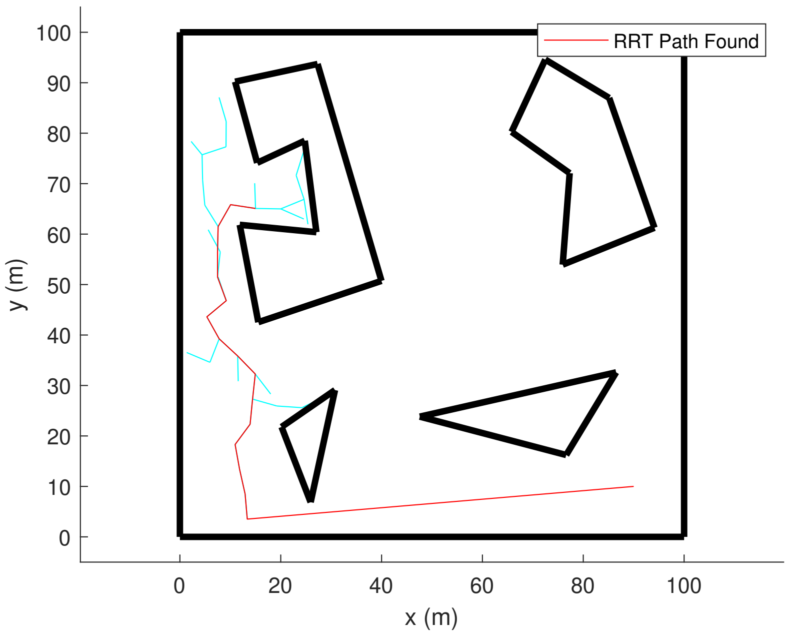

Figure 14: Shorted Path with Map and Tree Overlay

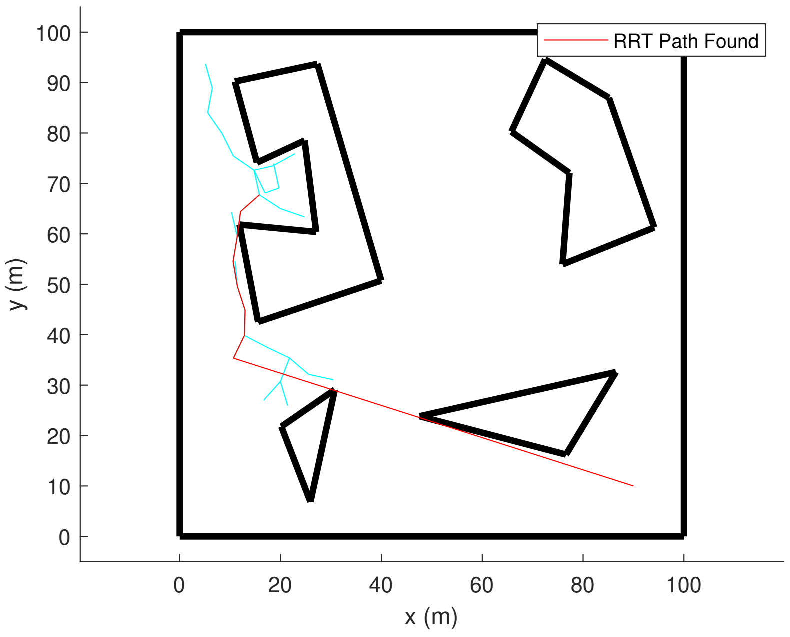

Figure 15: Shorted Path with Map and Tree Overlay

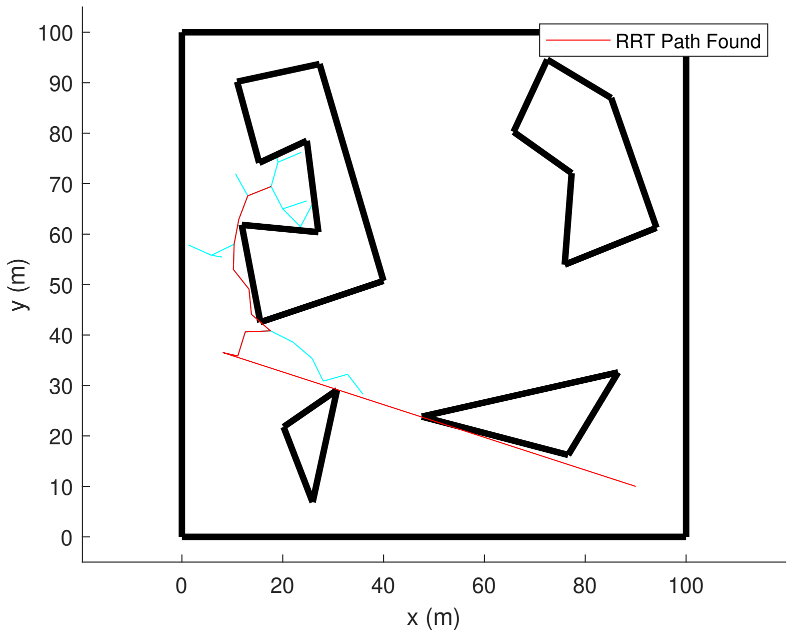

Figure 16: Shorted Path with Map and Tree Overlay

3. Please see above for plots and comments, sorry for confusion.

4. None of the paths except for the pose to node or node to goal are smooth lines. Often they end up going the wrong direction and coming back to a similar point which shows the inefficiency of the robot's motion. On the other hand this comes at the trade-off of an accurate map plan. Overall the paths are pretty good, but by no means the best!

Circular Robot: RRT

1. Please see matlab for program

2. To take the radius in account, I had my program go through every line in the map file and place a rectangle around it. Additionally for each obstacle (in this example each obstacle is a single line) my program finds the minimum convex hull vertices and exports these as the map.

   See figure below for the example. The map is shown and all of the stars are locations of the rectangles that are used as the new map. Sorry I did not have time to actually plot the rectangles...it's late

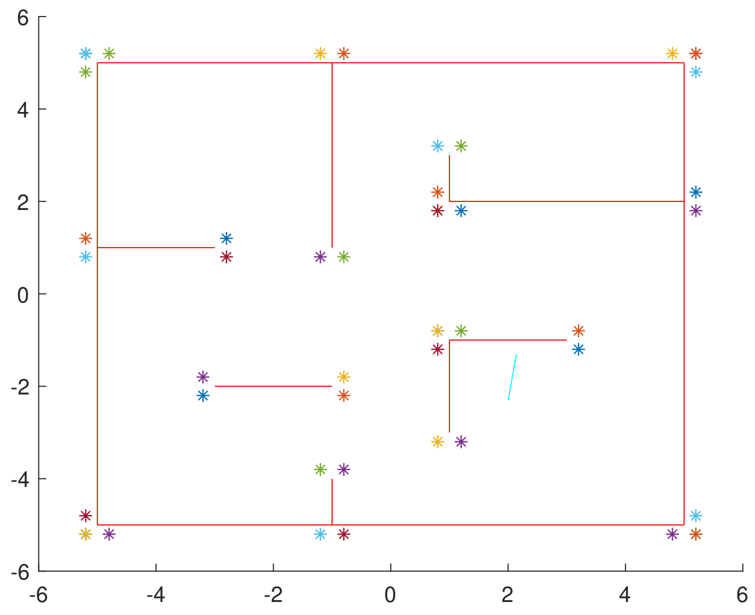

Figure 17: Map with Radius Account

3. Here is the plot of my robot running in corner map!

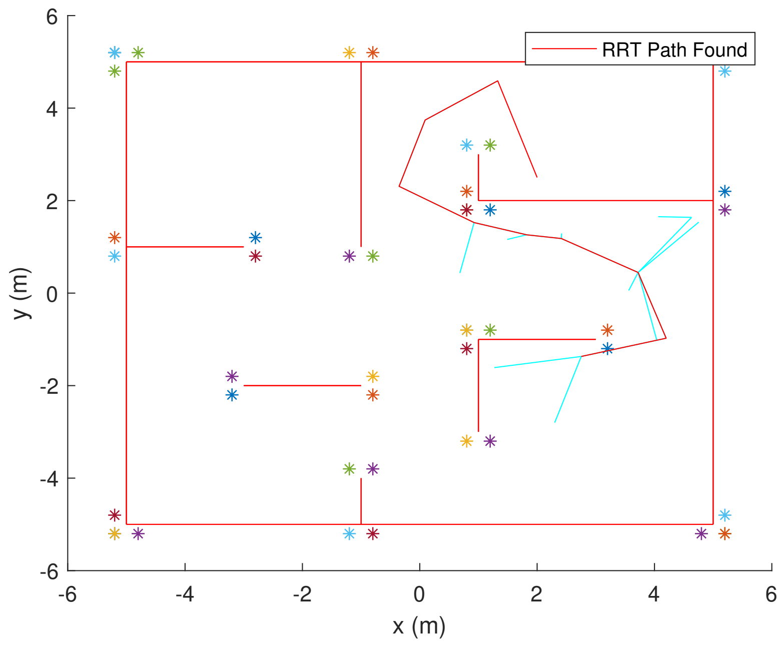

Figure 18: Shorted Path with Map and Tree Overlay

Test Function

Please see matlab for the function :)