Robotics: Planning and Navigation **Spring 2021**

Assignment-1: Report

Prepared by: Rahul Sajnani; Amarthya Sasi Kiran Dharmala; Abhiram Kadiyala

Contents

1 The problem statement **1**

2 Algorithm and Results **1**
2.1 Obstacle Map and Configuration map 1
2.2 RRT for Holonomic bot 3
2.3 RRT for Non-Holonomic bot 4

3 Results Link **6**

4 Work Division **8**

1 The problem statement

The robot has to navigate a two dimensional space, avoiding known locations with obstacles, traveling from its initial location to a goal location. Implement the RRT path planning algorithm for two cases, (1) Holonomic Robot and (2) Non-Holonomic Robot. Assume localization information i.e., robot's initial position, obstacle location, goal location is given.

2 Algorithm and Results

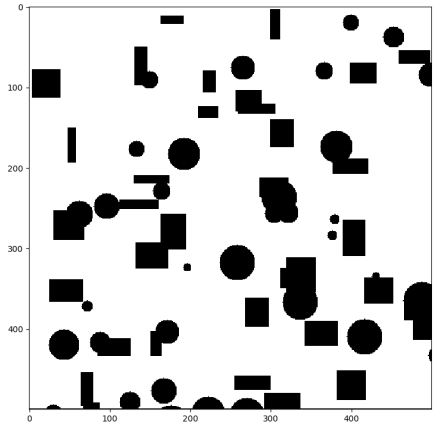
The RRT algorithm has been implemented on an obstacle map that we built. The configuration map is obtained from the obstacle map assuming the ego robot to be circular.

2.1 Obstacle Map and Configuration map

- To build the obstacle map, we first instantiated a map of required dimensions,with no obstacles. That is, a grid/matrix with only ones. The obstacle shall be represented using zeros.
- **For rectangular obstacles**, we sample random positions from the map for the top-left position of the rectange. Then, the width and height of the rectangle are sampled within fixed upper and lower limits

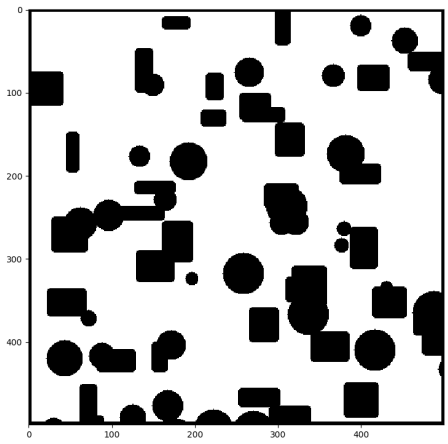
- From the top-left corner position, the width and the height we check if the obstacle is completely with the map dimensions and then embed it into the map.
- **For the circular obstacles**, we sample top-left corner postion of the tightest square which contains the circle of the required radius. We know the width and height are equal to the diameter of the circle. So, from this we check if the entire square containing the circle is within the map dimensions and then embed the circle into the map.
- Thus, we embed multiple circular and rectangular obstacles into the map and create the obstacle map.

Figure 1: An obstacle map with circular and rectangular obstacles.



- Configuration map is now obtained by using dilation. The ego robot is assumed to be a cirular object with a certain radius and the obstacle map has value "1" in free space and value "0" for the obstacle. So, we dilate the obstacle map using a circular disk with the radius of the robot to get the configuration map

Figure 2: Configuration map corresponding to the above obstacle map.



2.2 RRT for Holonomic bot

1. The initial node, final node and the obstacle/configuration map are avilable. A tree is initiated with the initial node as the root.

2. A random location is sampled from the free space in the free space in the configuration map.

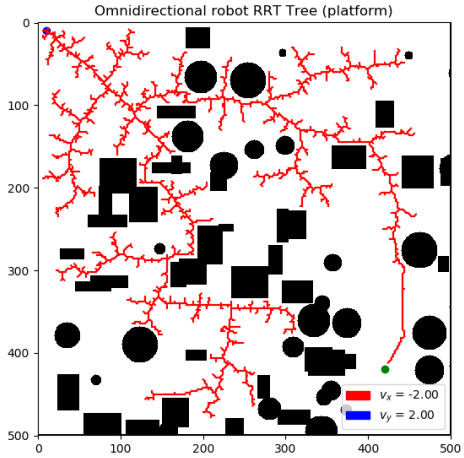
3. The nearest neighbour to this random node is them computed. And, a direction vector is found between the nearest neighbour and the random node.

4. The ego then travels in this direction from the nearest node on the tree, with a fixed velocity for one time step. This new position node is then added to the tree as the neighbour of the nearest neighbour.

5. The sampling of the new node is done with an epsilon greedy policy, where the new node is sampled in the direction of the goal instead of a random node with a certain probability.

6. The steps 2-5 are done repeatedly until convergence. The algorithm converges if the distance between the goal and any node in the tree is lesser than a threshold or if it reaches the maximum number of iterations.

Figure 3: Center trajectory for an omni-directional holonomic robot (platform)

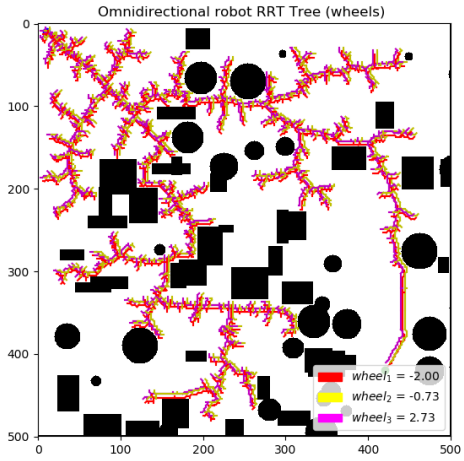


7. The wheel trajectories are computed from the center velocity using the following transformations for an omni-directional robot.

Figure 4: Wheel velocities from the center for an omni-directional robot

```python
def getWheelVelocity(self, v_x, v_y, omega, position_old):
    '''
    Function to obtain wheel velocity of omnidirectional drive
    '''

    theta_old = np.deg2rad(position_old[2])
    velocity_vector= np.array([[v_x], [v_y], [omega]])

    # Local to global transform
    local_T_global = np.array([[np.cos(theta_old),                0,           0],
                               [               0, np.cos(theta_old),           0],
                               [               0,                0,           1]])

    # Local to wheel transform
    wheel_T_local = np.array([[               -np.sin(theta_old),               np.cos(theta_old), self.radius_bot],
                              [ -np.sin(theta_old + self.alpha_2), np.cos(theta_old + self.alpha_2), self.radius_bot],
                              [ -np.sin(theta_old + self.alpha_3), np.cos(theta_old + self.alpha_3), self.radius_bot]])

    # Apply global to wheel transform to get velocity for each of the wheels in Omnidirectional robot
    wheel_velocity_vector = wheel_T_local @ local_T_global @ velocity_vector

    return wheel_velocity_vector
```

Figure 5: Wheel trajectories for an omni-directional holonomic robot



2.3 RRT for Non-Holonomic bot

1. The initial node, final node and the obstacle/configuration map are available. A tree is initiated with the initial node as the root.

2. In the non-holonomic case, we also store the heading of ego at every node in addition to the position.

3. A random location is sampled from the free space in the free space in the configuration map.

4. The nearest neighbour to this random node is them computed. And, a direction vector is found between the nearest neighbour and the random node.

5. Now, we find the angle this direction vector is making w.r.t the axis and then find the difference between the heading of the nearest neighbour and this angle.

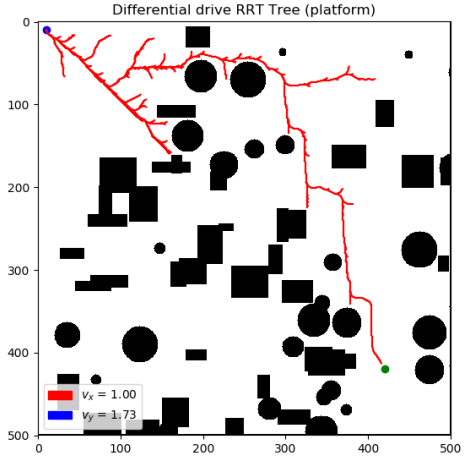
6. This is the angle by which the ego must turn. Now, we find the velocity vectors in the x and y components using $V_x = V * cos(\theta_{diff})$ and $V_y = V * sin(\theta_{diff})$.

7. The ego then travels in this velocity direction from the nearest node on the tree, for one time step. This new position node is then added to the tree as the neighbour of the nearest neighbour.

8. The sampling of the new node is done with an epsilon greedy policy, where the new node is sampled in the direction of the goal instead of a random node with a certain probability.

9. The steps 2-8 are done repeatedly until convergence. The algorithm converges if the distance between the goal and any node in the tree is lesser than a threshold or if it reaches the maximum number of iterations.

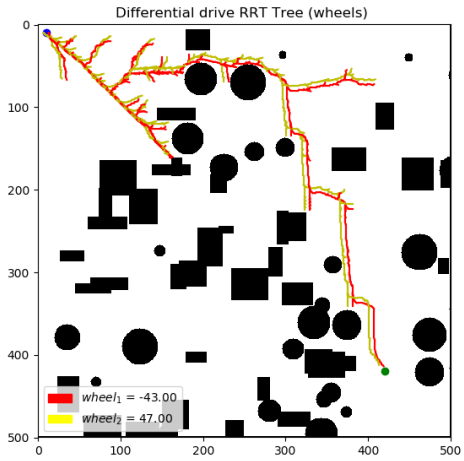
Figure 6: Center trajectory (platform) for a differential drive non-holonomic robot



10. For, obtaining the wheel velocities, we use the v and ω values.

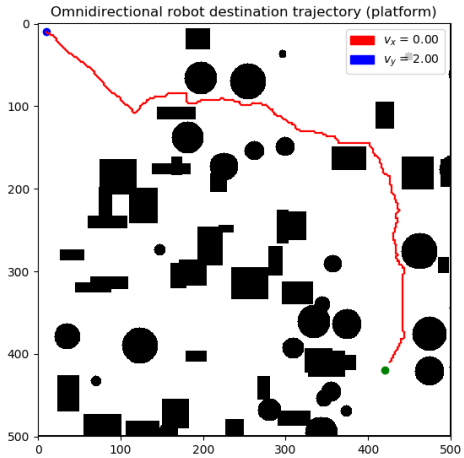
11. The we get R = v/ω. Now, from kinematics, Left wheel velocity is $V_l = (R-d)*\omega$ and right wheel velocity is $V_r = (R+d) * \omega$. When $\omega = 0$, then the $V_l = V_r = V$ and it takes a straight line path.

Figure 7: Wheel trajectories for a differential drive non-holonomic robot

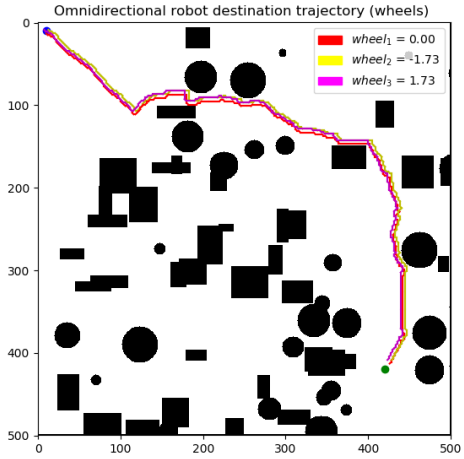


3 Results Link

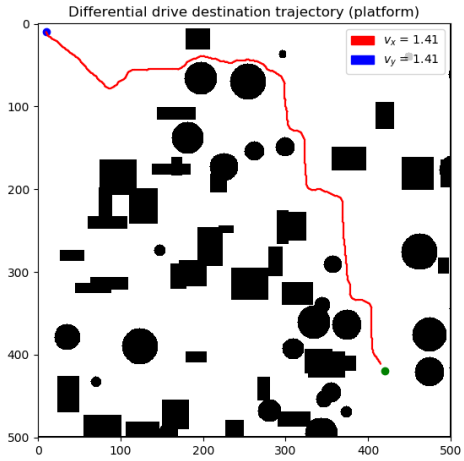
- The final center (platform) trajectory of the robot after **back-tracking** for Omni-directional bot:



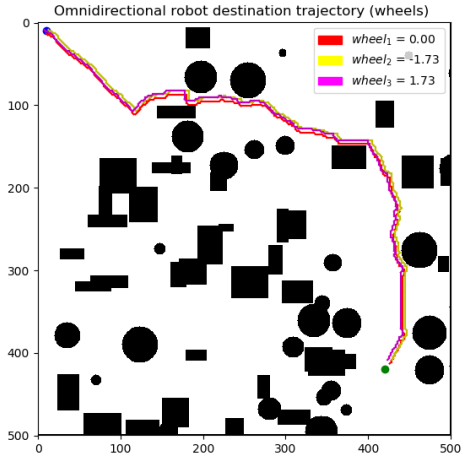
- The final wheel trajectory of the robot after **back-tracking** for Omni-directional bot:



- The final center (platform) trajectory of the robot after **back-tracking** for Differential drive bot:



- The final wheel trajectory of the robot after **back-tracking** for Differential drive bot:



PFA the result videos in this link - `https://drive.google.com/drive/folders/12lggRxKheQF8dmi6t5lcueoSUFNzi5r9?usp=sharing`

4 Work Division

- Building Map - Rahul and Sasi
- Configuration Map - Sasi and Abhiram
- Sampling nodes and Tree. - Rahul and Sasi
- Holonomic Agent center and wheel - Abhiram and Rahul
- Non-Holonomic Agent centre and wheel kinematics - Sasi and Rahul
- Plotting - Rahul and Abhiram
- Report - Sasi, Abhiram and Rahul.