Mobile Robots Assignment 2

Charlie Maclean

January 2020

Exercise 1

(a) To calculate the vector field, I normalize the vector from the current position to the goal position, then multiply it by the desired speed, which in this case I set to MAX_SPEED / 2.1. See Figure 1 for results.

(b) For each obstacle, the distance to the outside of the cylinder from the current position is calculated. I then set the vector to point in the direction of the current point from the centre of the cylinder, with a magnitude = MAX_SPEED / 2 - distance. This ensure that the magnitude is greater than the goal based vector field when close so it can overcome it. Results are in Figure 2.

(c) Results are in Figure 3. The vector field approaches the goal until it gets to the obstacle at which point it is forced to go around the cylinder. This is as the vector field points towards the goal from the current position, but is overcome by a stronger velocity pointing away from the cylinder when it gets too close.

(d) The path followed is a straight line which gets stuck in front of the cylinder. This is as when we place the cylinder at (0,0), the gradient field for obstacle avoidance is the exact opposite direction to the gradient field for finding the goal. Hence the velocities simply go back and forth, never actually getting past the obstacle.

A solution to get around this would be to rotate every vector from the obstacle avoidance slightly. Hence, instead of getting stuck, our trajectory will go around the cylinder.

(e) I implemented the solution, so every velocity around the cylinder is rotated by $\pi/8$ counter-clockwise. The results are in Figure 4.

(f) With two obstacles close to each other, my solution from part b fails. This is as the vector fields from both obstacles combine and push against the robot with equal magnitude. This means the robot does not move around either cylinder. The solution is the same as in the last question, I rotate the repulsive vector field by $\pi/8$. This rotation causes the robot to avoid entering the midpoint between the two obstacles. The results are in Figure 5

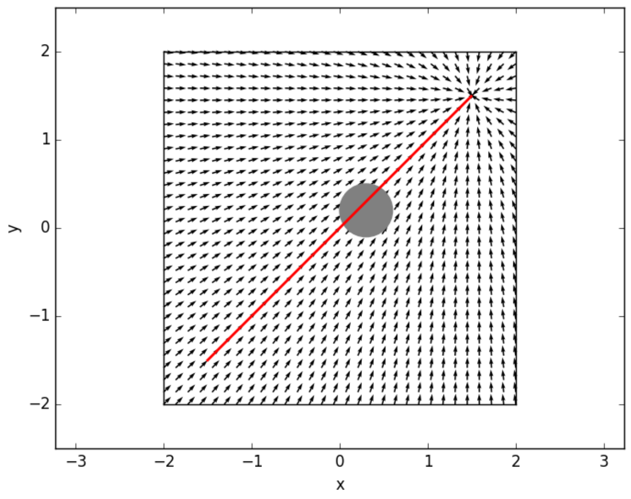


Figure 1: Vector Field in Goal Mode

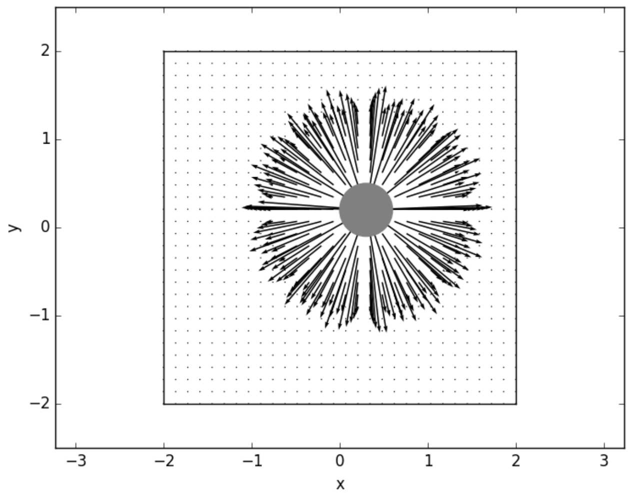


Figure 2: Vector Field in Obstacle Mode

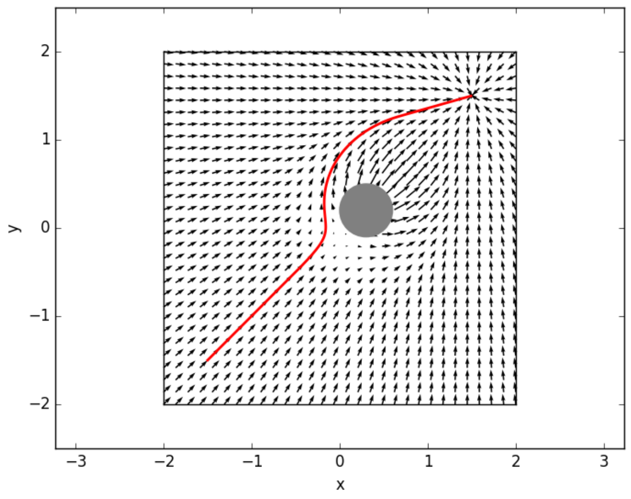


Figure 3: Vector Field

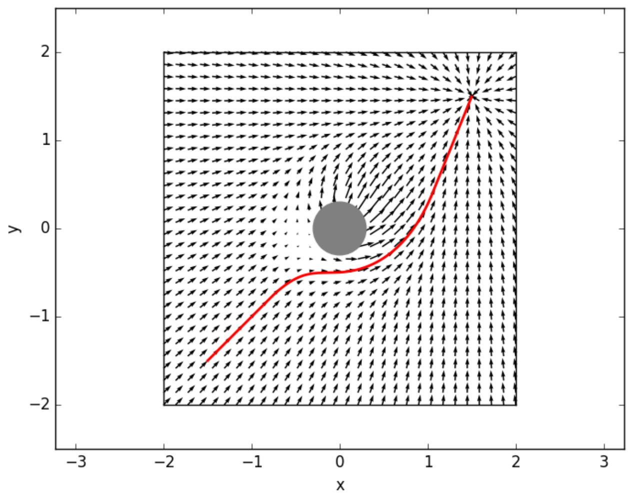


Figure 4: Vector Field with Fix

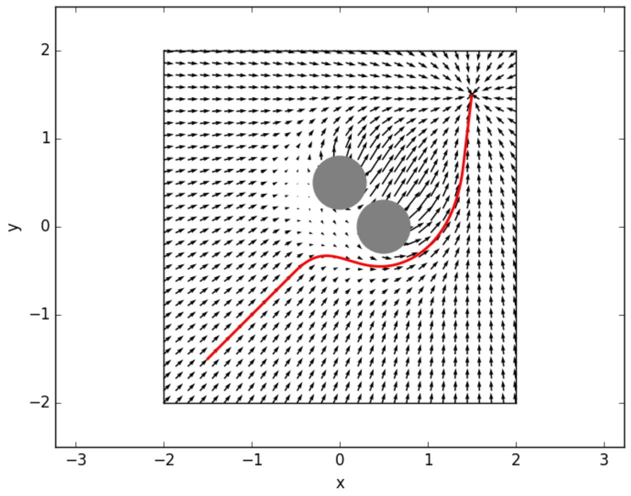


Figure 5: Trajectory with two Obstacles

Exercise 2

(a) Relating holonomic point position to robot position:

$$x_p = x + \epsilon \cos \theta$$

$$y_p = y + \epsilon \sin \theta$$

Differentiating with respect to time, substituting in u and ω:

$$\dot{x_p} = u \cos \theta + \epsilon(-\omega \sin \theta)$$

$$\dot{y_p} = u \sin \theta + \epsilon(\omega \cos \theta)$$

Solving for u and ω:

$$u = x_p \cos(\theta) + y_p sin(\theta)$$

$$\omega = \frac{-x_p sin(\theta) + y_p cos(\theta)}{\epsilon}$$

(b) Feedback Linearization converts a non-holonomic problem to a holonomic one. We can more easily solve problems in a holonomic problem as we can control every axis individually. In our case, the linearization has an effect of smoothing out sharp corners that our robot would not be able to turn through.

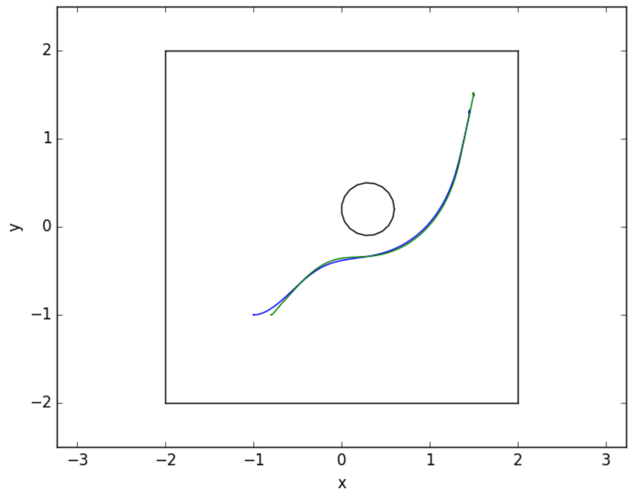


Figure 6: Trajectory of Feedback Linearization Solution

(c) The trajectory is in Figure 6. We see the end of the 'rod' in green follows the path generated in Exercise 1 - this is the solution to the holonomic problem as we are moving without constraints in x and y. Then, we see the robots path in blue which is the non-holonomic problem, we calculate ω and u by feedback linearization. In other words, the robots path is chosen as if it's been dragged by a rod from a point following the green trajectory.

(d) We don't require absolute pose of the robot, as the trajectory is calculated based on the vector field. The vector field is not reliant on any specific coordinate frame, meaning we can change to a different basis.

get_relative_position is implemented such that it transforms to a coordinate system with origin at *absolute_pose*. The implementation then calculates velocity using all coordinates as compared to the current robot pose.

This is beneficial as in many situations we do not have any absolute positions of anything. All we have is the position of objects as measured by sensors attached to the robot, with the robot as a basis.

You can see the result of the relative solution (in Figure 7) is the same as the absolute solution.

Exercise 3

(a) *sample_random_position* takes a random position within the range of the grid. If that position is not free in the occupancy grid, then it generates new positions until one is free.

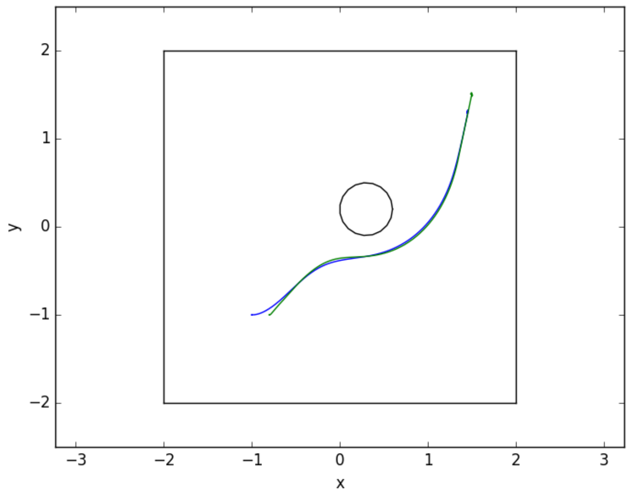


Figure 7: Trajectory of Relative Solution

(b) In *adjust_pose*, I first calculate the YAW of *final_pose*. Using geometry (see image below) I worked out that this value (θ_1) could be calculated using the YAW of *node* (θ_0) and the angle between the x axis and the vector between *node* and *final_pose* (ϕ), with the following equation $\theta_1 = 2\phi - \theta_0$. See Figure 8 for a labelled diagram.

    Next, the chosen path is analysed to check it is actually valid. The algorithm iterates around the arc between the points, at each step checking all the grid squares the robot may be in. I check the grid every robot/2 m around the arc, to ensure I cover all possible grid squares.

    See Figure 9 for the plot of RRT running.

(c) Having run the program multiple times, there is wide variety in the length of the path it selects. This is as it is just a random algorithm - it does not search for the shortest route, it just finds any route.

    A solution would be to use the algorithm RRT* instead. This algorithm extends on RRT in two ways. First, instead of connecting the random node to the nearest node, it connects to a near node which will give the lowest distance to the start node. Secondly, it 'rewires' itself, by checking whether any of the random node's neighbours would have a shorter route to the start through the random node instead.

(d) I implemented RRT* as described above. The parent is chosen based on which node gives lowest total distance from the start. I added the rewiring, using a recursive function. The function is called if we find any node which

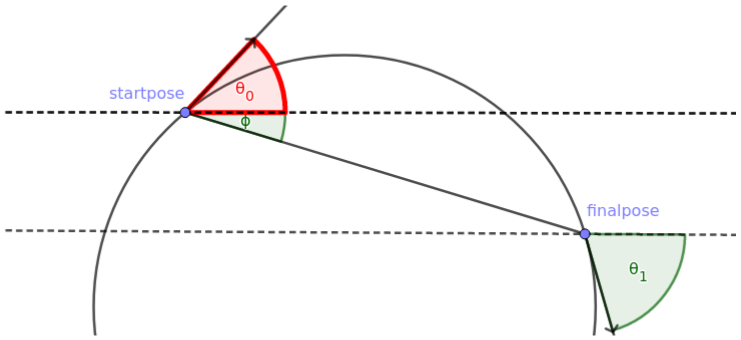


Figure 8: Node Pose Problem

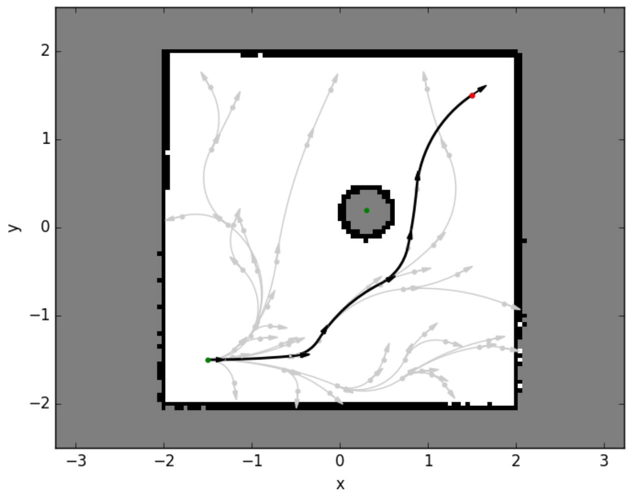
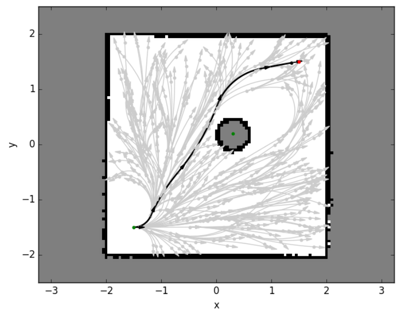


Figure 9: RRT Algorithm Plotted


(a) RRT*

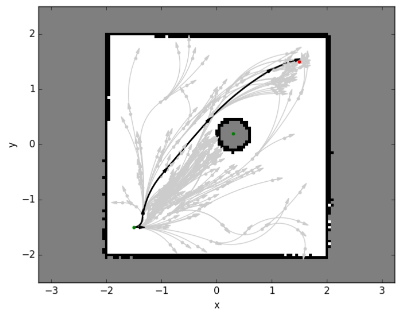

(b) Informed RRT*

Figure 10: Comparison of RRT* and Informed RRT*

would have a shorter path from the start through the random node. It recursively goes through the tree, updating all angles so the tree is valid.

I found that RRT* found consistently shorter paths than RRT, however it does this at the cost of time. RRT* can be thought of as finding the shortest path to any position on the map, which is clearly a waste of resources - in our case, we only have one goal. Hence, Informed RRT* (Jonathan D. Gammell et al.) was created to provide a quicker algorithm that uses knowledge of the position of the goal state to guide the solution to a shorter path in less time. The algorithm runs as RRT*, until it finds a solution, at which point it restrains the sample space for random points to an ellipse containing the goal, start and current path. We know any shorter path must fall in this ellipse.

I implemented Informed RRT* in the file *rrt_informed.py*. A plot comparing Informed RRT* and RRT* is in Figure 10.

The performance can be seen in the table below. All averages are calculated over 40 runs.

Algorithm	Average Computation Time (s)	Average Path Length (m)
RRT	0.21	4.94
RRT*	3.33	4.43
Informed RRT*	1.58	4.41

Exercise 4

(a) I copied and pasted from Exercise 2.

(b) Using motion primitives can ensure that non-holonomic robots can traverse every path, as they allow for restrictions to be put into place in the

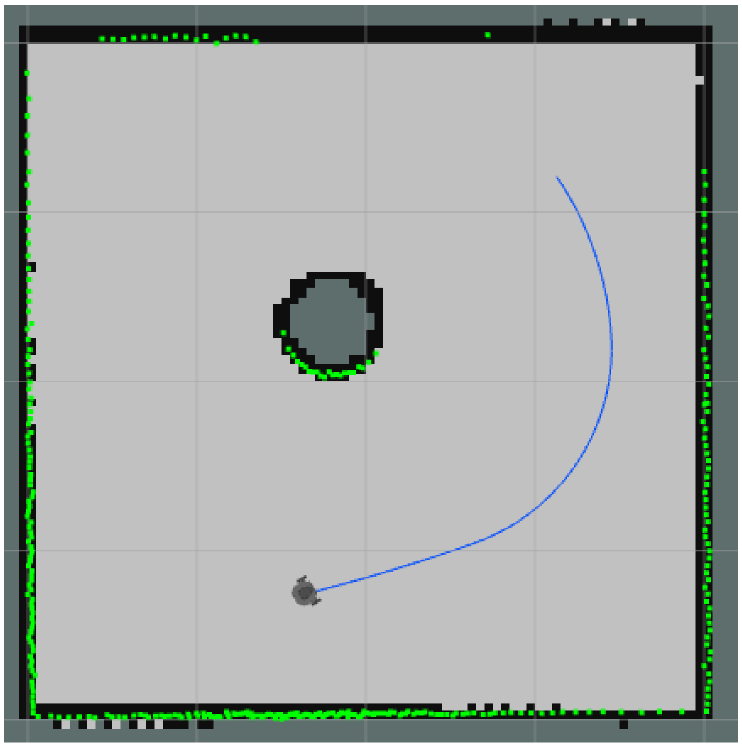


Figure 11: RViz Simulation of RRT

planning algorithm. The disadvantage is they require additional calculation.

(c) The path provided in *get_velocity* is a set of points each 5cm apart. Additionally, the controller updates velocity every 100ms and the robot has a max-speed of 0.5m/s which means that the robot will move a maximum of 5cm between velocity updates. Hence, a good method of deciding velocity would be to direct the robot towards the next point.

Therefore, my implementation of *get_velocity* finds the nearest node along the path, and then sets the velocity to point towards the next node in the path. I normalize and multiply by the max speed to ensure constant velocity.

See Figure 11 for the RViz simulation showing the path it has chosen.

(d) The *slam.launch* command launches the gmapping package and opens an RViz window. The gmapping package is responsible for Simultaneous Localization and Mapping (SLAM), which means it attempts to locate the robot in an unknown environment, by building up a map based on sensor values it has received.

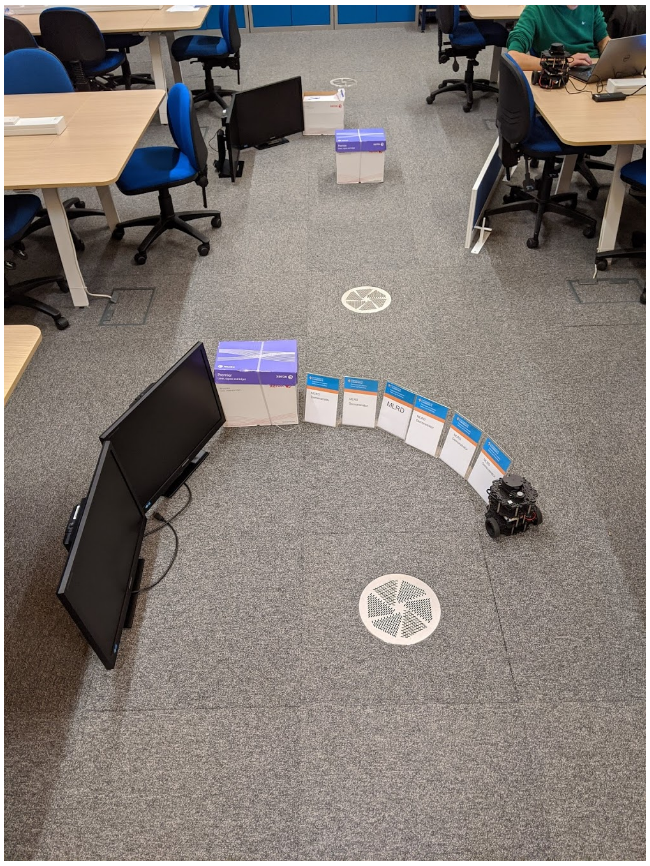


Figure 12: Robot's Environment

The RRT algorithm takes the output map from SLAM and uses it as an occupancy map for finding routes to destinations.

(e) I setup an environment out of monitors and boxes that I could find, see Figure 12. I then ran the robot, which used SLAM to generate a map, which resembled the environment well, as can be seen in Figure 13.

The robot was very successful in navigating the environment, even able to travel around the center box in the back of the environment. However one thing it struggled with was the thin display cards that the robot is attempting to drive into in the image. The issue with those was they were so thin that the radar has very little to reflect of off and hence struggles to detect them when driving into their edges.

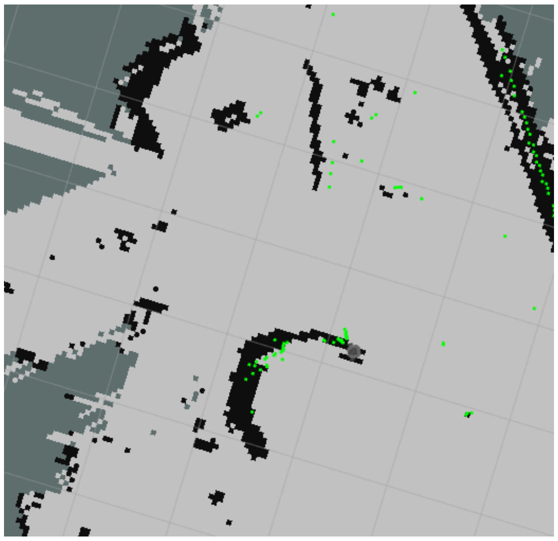


Figure 13: Map Generated by RViz