Introduction to Robotics

Assignment 2

Jingyu Huang

Queens' College

Potential Field Method

Exercise 1

(a) The attractive velocity field is implemented as:

```
x = goal_position - position
v = MAX_SPEED * np.tanh(x)
```

The velocity is pointing from the current position to the goal position. The `tanh` function maps the range $(-\infty, \infty)$ to $(-1, 1)$, which we used to map position difference to the range between 0 and `MAX_SPEED`.

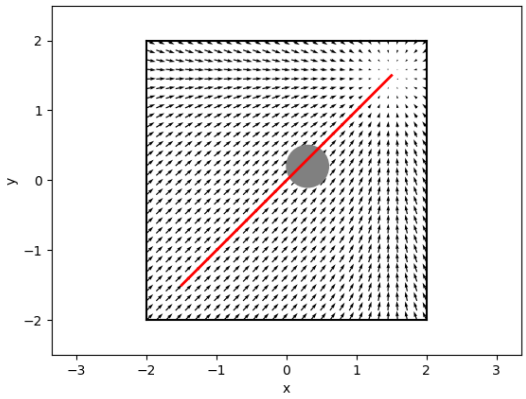

Figure 1: Plot of velocity field that reaches the goal.

(b) The repulsive velocity field is implemented as:

```
farmargin = 1.2
nearmargin = 0.1

for i in range(len(obstacle_positions)):
  x = position - obstacle_positions[i]
  d = np.linalg.norm(x)-obstacle_radii[i]
  if d <= nearmargin:
    v += MAX_SPEED * normalize(x)
  elif d <= farmargin:
    x = normalize(x) * (farmargin-d)
    v += MAX_SPEED * np.tanh(x)
```

This is basically the opposite of the attractive field, where far and near margins are defined. The velocity is defined based on the distance between the robot position and the edge of the obstacle. If the robot is outside the far margin, velocity is not affected. If the robot is within the far margin and outside the near margin, velocity scaled with the negative of the displacement from the current position to the edge of farmargin. If the robot is within the near margin, it is pushed out with the strongest velocity possible. The far and near margins can be tuned according to performance of object avoidance.

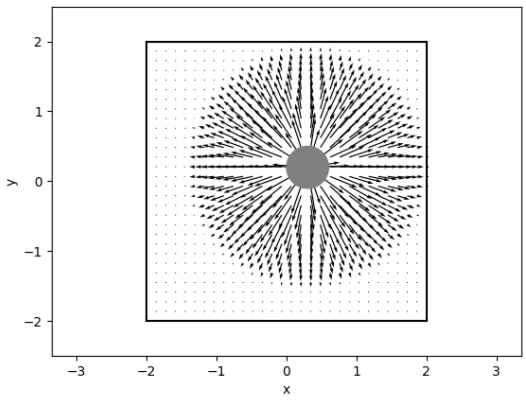

Figure 2: Plot of velocity field that avoids the obstacle.

(c) The combination of the two fields gives the result as follows:

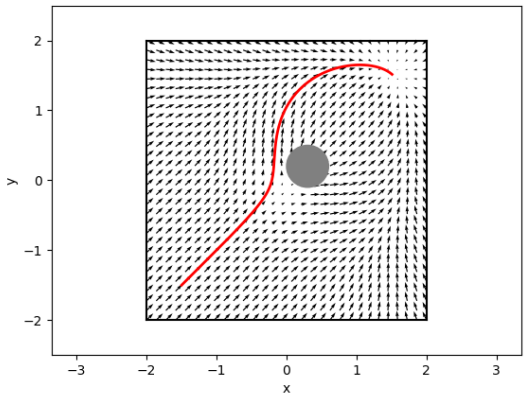

Figure 3: Plot of combined velocity field.

The robot initially follows the attractive field only, but when close enough to the obstacle it is pushed away as the repulsive field there is stronger than the attractive field in the direction normal to the wall. The robot then follows a curve defined by the combination of the two fields into the goal.

(d) When the obstacle is placed at $(0, 0)$, the robot, the obstacle, and the goal are placed along the same line. This means that the velocity field lines from both fields would be pointing in the same direction. As the result, the robot would not be able to turn as neither of the velocity fields are contributing in directions tangential to the obstacle when the robot comes in this direction. The robot would either stop in from of the obstacle, or, as in our case, try to go straight through the obstacle.

A possible mitigating solution to this is to introduce an artificial small velocity that is normal to the original. The choice between the additional velocity pointing to the left or to the right is arbitrary and makes no difference in this particular scenario. We chose to have the additional velocity point to the right.

The following graphs shows what happens when the obstacle is placed at $(0, 0)$ (without the mitigating solution):

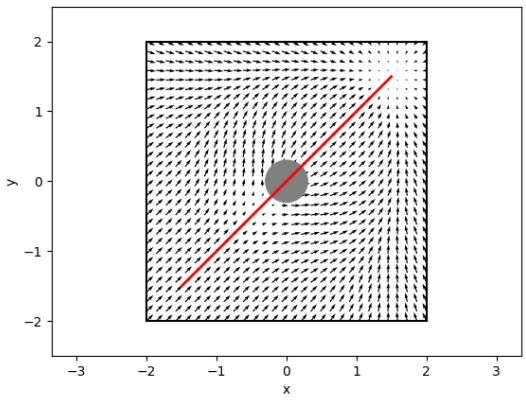

Figure 4: Plot of combined velocity field.

(e) The mitigating solution is implemented in the repulsive field as:

```
farmargin = 1.2
nearmargin = 0.1

for i in range(len(obstacle_positions)):
  x = position - obstacle_positions[i]
  d = np.linalg.norm(x)-obstacle_radii[i]
  if d <= nearmargin:
    v += MAX_SPEED * normalize(tilt(x))
  elif d <= farmargin:
    x = normalize(x) * (farmargin-d)
    v += MAX_SPEED * np.tanh(tilt(x))

def tilt(v):
return v + 0.1 * np.array([-v[0], v[1]], dtype=np.float32)
```

The code is very simple and self-explanatory. Instead of directly computing `v` from normalised `x`, we first add an additional velocity that is normal to the original and with a magnitude that is 10% of the original.

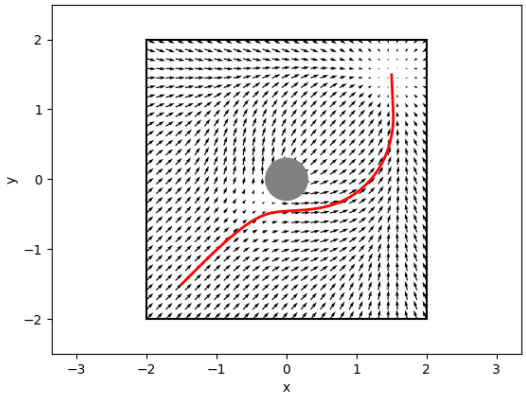

Figure 5: Plot of combined velocity field with mitigating solution.

As can be observed, the robot now turns to the right when in a course of direct collision with the obstacle as a result of the additional normal term in the velocity field, and reaches the goal successfully.

(f) The path taken by the robot in the scenario with two robots is as follows:

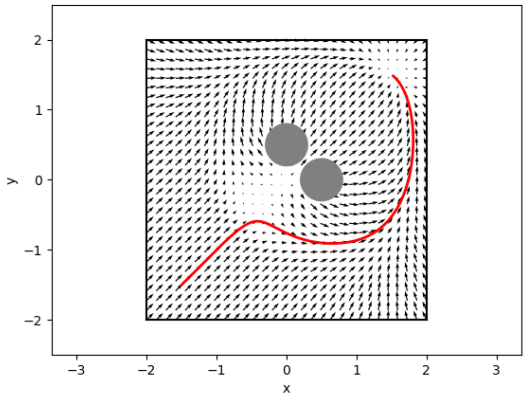

Figure 6: Plot of combined velocity field with two obstacles.

As a result of the mitigating solution in the previous part, the robot is able to turn right and take a save route around the two obstacles in stead of trying to squeeze between them. The solution apparently also works in this scenario.

Exercise 2

(a) Let the speed of the control point in X and Y directions be $\dot{x_p}$ and $\dot{y_p}$, respectively, and let the forward velocity and angular velocity of the robot be u and ω. The equations that provide feedback linearisation would be given by:

$$\begin{cases} u = \dot{x_p}\cos\theta + \dot{y_p}\sin\theta \\ \omega = \epsilon^{-1}(-\dot{x_p}\sin\theta + \dot{y_p}\cos\theta) \end{cases} \tag{1}$$

(b) Feedback linearisation allows the non-holonomic differential drive robot to be controlled by linear control of the holonomic feedback point. This vastly simplifies problems such as trajectory tracking as we do not need to worry about the non-holonomic kinematics of the robot itself. We only need to control the holonomic feedback point, and the equations from the previous section would be able to take care of the actual robot control inputs for us.

(c) The velocity field and trajectory plot from the implementation in webots are as follow:

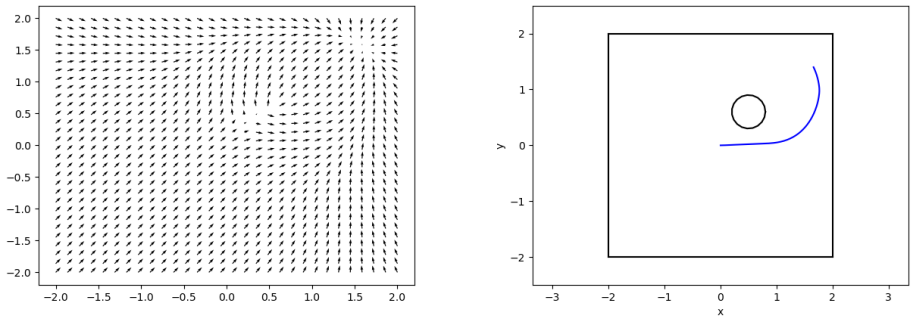

Figure 7: Plot of combined velocity field in webots and the resulting path.

The robot is clearly capable of following the trajectory and reaching the goal. The trajectory of the robot is mostly in line with the arrows in the velocity field, which is what we expected. Notably however, The robot follows a slightly wider trajectory than expected as it is trailing behind the holonomic feedback point. Making it more likely to collide with the side walls which are not registered as obstacles in the code. This is fixed by tuning down the `far_margin` in the code to a smaller value.

(d) The implementation does not require knowledge of the absolute pose of the robot since only the distance of the robot from the goal and the obstacles are required to compute the current velocity. Changing from absolute position to relative position is merely a linear change of coordinate systems and should not affect how the implementation works. This is reflected in the implementation of `get_relative_position`, and the fact that the exact same plot as in part (c) is produced after running the robot with relative positions.

The advantage of using relative position is that is a real life application with the program running on the robot, the program would only be able to acquire information about relative positions. Using an implementation using relative positions means that the robot does not require external sources of information such as GPS to get the absolute coordinates, which make the robot more robust as it can still work when GPS is not available.

Rapidly-Exploring Random Trees

Exercise 3

(a) The implementation of `sample_random_position` is simple: a random coordinate is chosen between the range $[-2, 2]$ by scaling and offsetting `np.random.rand`. Occupancy of this coordinate is checked using `occupancy_grid.is_free(position)` is a while loop, which will keep running until a free position is found.

(b) The implementation of `adjust_pose` uses a brute force search method, which is surprisingly not very slow at all. In the first part of the function, the yaw of the `final_node` is found by searching through the range $[0, 2\pi]$, using `find_circle` to generate a circle at each yaw value and validating the circle by making 5 checks:

1. The distance from the starting node to the center of the circle is equal to the radius
2. The distance from the final node to the center of the circle is equal to the radius

3. The direction at the starting node is perpendicular to the radius

4. The direction at the final node is perpendicular to the radius

5. The starting node and the final node are going in roughly the same directions

A yaw that satisfies all 5 checks would make a valid yaw for the `final_node`.

After finding the valid yaw, collision checks must be made to ensure that the arc does not go through obstacles. This is done by sampling points along the arc and using `occupancy_grid.is_occupied(position)` to check if these points are inside an obstacle. The arc is sampled by first calculating the angle subtended by the arc using arctan, then dividing by the number of points to get a small angle increment. The sample points are obtained by incrementing the yaw of the radius from the center to the starting point by this small angle and recalculating its position.

The resulting path generated by RRT is as follows:

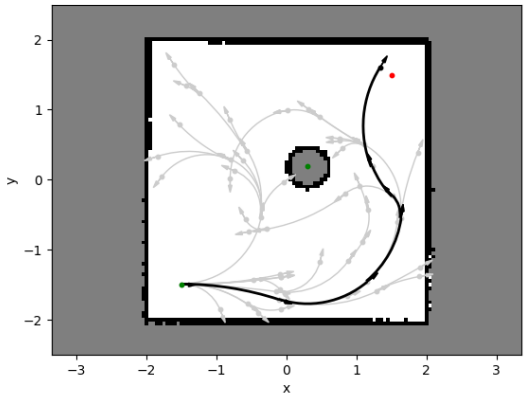

Figure 8: Plot of the trajectories of RRT.

A smooth connected path is generated using this method, and observing the grey test paths show that points close to the obstacle have been sampled but then abandoned as there is no way forward, showing that the obstacle avoidance part of the algorithm does in fact work.

(c) An obvious drawback of RRT is that the optimality of the paths are not guaranteed. In the graph above, the robot initially goes downwards, which is not ideal as the goal is upwards. The suboptimality is more obviously shown in Figure 9, where the robot can move further away from the goal.

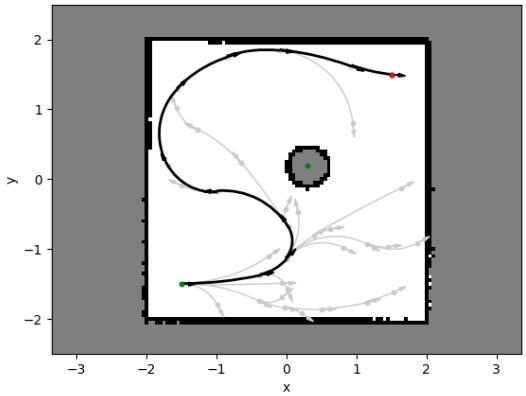

Figure 9: Plot of a suboptimal trajectory of RRT.

(d) A solution to this would be to assign a cost to each node, based on their distance from the goal. The closer the distance, the lower the cost. By making sure that the cost of the next node is always lower than the cost of the current node, we can ensure that the robot is always moving closer to the goal and as a result, recover some form of optimality.

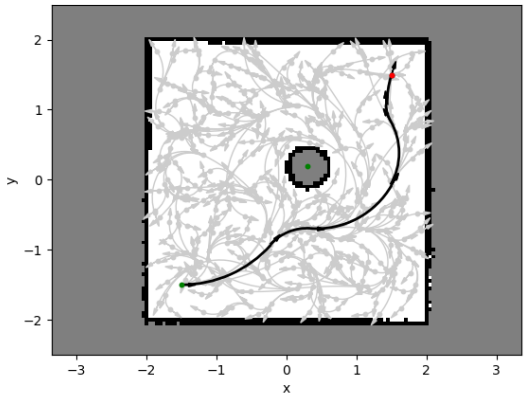

Figure 10: Plot of a trajectory of RRT with node cost taken into consideration.

From the graph one can see that the robot is constantly trying to move towards the goal, as expected. One can also see that there are a lot more grey arrows in the plot, indicating that the algorithm is attempting a lot more trial paths in trying to find a more optimal solution.

The resulting performance is not ideal as it becomes notably slower, to the point where one can argue for another solution of dealing with this problem which is recording the total cost of a path and just running the vanilla RRT several times and pick the best one. Additionally although optimality is improved, it is still not as optimal as a search procedure such as A* would be, which is perhaps an issue inherent to the concept of RRT itself.

Exercise 4

(a) Feedback linearisation in this section is implemented in the exact same way as in Exercise 2. The initial setup is shown as follows:

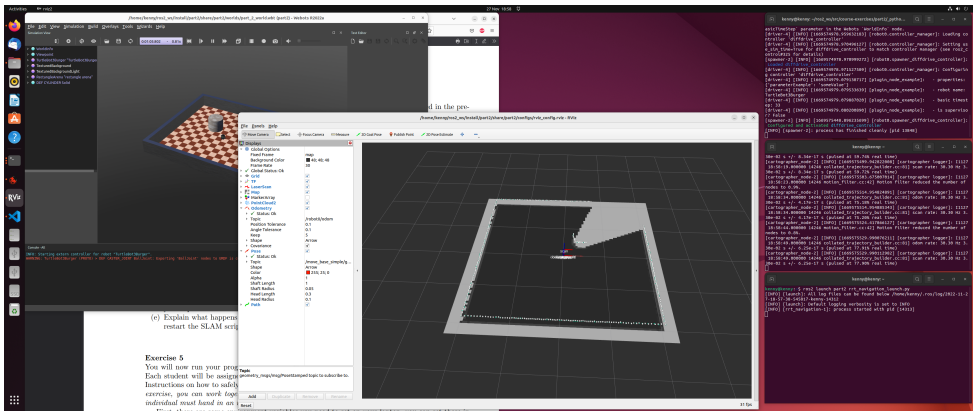

Figure 11: Initial setup for RRT navigation in webots.

(b) In the RRT algorithm, the primitives needed are method to sample configurations in the free configuration space and method to check for collisions in the obstacle configuration space. These two are implemented in `adjust_pose`, as explained in Exercise 3(b).

The advantages of RRT algorithm are that it is probabilistically complete, with an exponential decay in probability of failure, and also that asymptotically optimal algorithms exist such as RRT* exist which can be leveraged to improve the pathing.

The disadvantages are that if the robot is non-holonomic, edge generation would require cumbersome solutions to the two-point boundary value problem.

However, only a tree is constructed in RRT instead of a graph like in PRM. This means that one only needs to plan forward, so motion primitives can be leveraged to sample edges by rolling out the forward kinematics model of the robot in time. In this way one can know by construction that the points reached are feasible.

(c) `get_velocity` is implemented as follows:

1. Using the current position of the robot and the positions of all the path points, find the path point that is closest to the robot, and record the index and the distance.

2. If the closest distance is above than a certain threshold, set the velocity to be the displacement of the closest point from the current robot position. Otherwise, set the velocity to be the displacement of the next point (found by incrementing the index) from the current robot position.

Noteably, the speed of the robot is set to only half of the `SPEED` value provided in the code, as a high speed causes the robot to overshoot and move in a jaggy fashion while the path planned changes drastically with each update. A lower speed provides better stability and more precise path following.

Setting the goal on RViz populates the topic named: `/move_base_simple/goal`.

(d) A screenshot of the RRT algorithm running with robot, path and map shown is included in Figure 12.

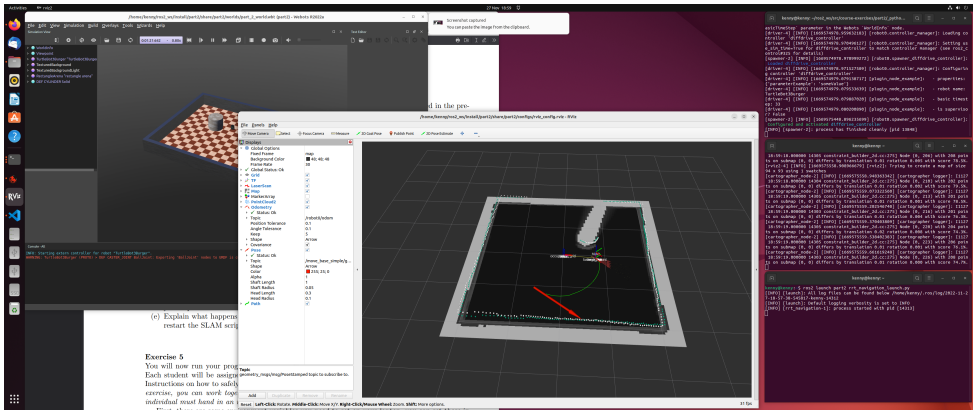

Figure 12: RRT navigation running with visualised goal and path in webots.

(e) SLAM provides real time simultaneous localisation and map generation, which populates the occupancy grid and allows the RRT algorithm to execute path building and obstacle avoidance. After the robot has started moving, the RRT algorithm would already have a rough map such as the one shown in Figure 11, and the fact that the robot moves means that a path is already built. If SLAM is shut down at this point, the robot still follows the path already built to the goal, as the path following is solely dependent on RRT.

When SLAM is restart, the previously explored portions of the map are reset and the map is rebuilt using the current laser sensor values. New goals can be set where paths exist on the current map between the new goal and the position of the robot.

Exercise 5

SLAM updates in real time to draw the map of the surrounding area based on the readings of the laser scanners and provide localisation to pinpoint the position of the robot on the drawn map. These pieces of information are used to populate the `occupancy_grid` with information about obstacles and free spaces, and to fill in `position` of the robot. The former is used by `rrt.py` in `adjust_pose` for obstacle collision detection, and the later is used by `rrt_navigation.py` in `get_velocity` for path following.

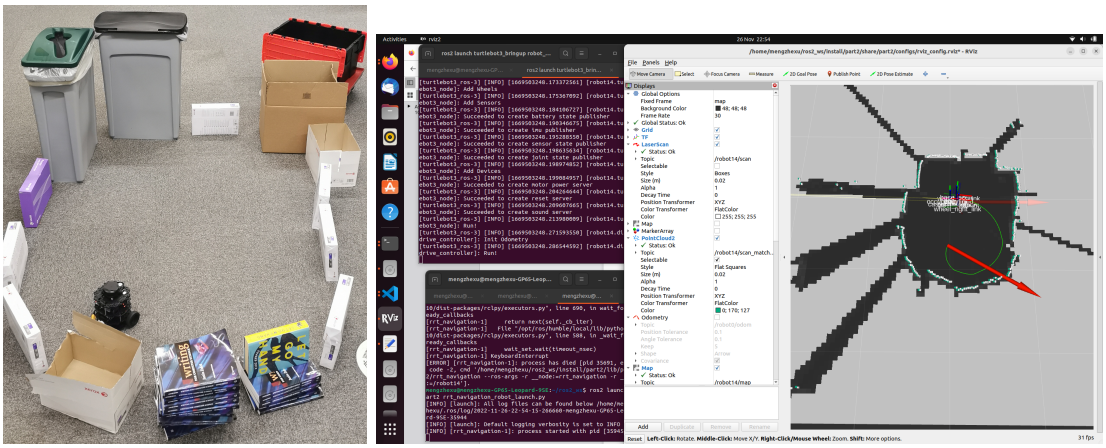

Figure 13: Setup of physical robot playground and RRT navigation running with visualised goal and path visualised on the computer.

The setup of environments in real world and on the computer are shown in Figure 13. We noted that SLAM only works with a large enough playground, and also that obstacles must be high enough for the real laser scanner to register. The robot can follow the path set out by `rrt.py` reasonably well, provided that the given velocity is low enough.

We also noted that in mapping the environment, just having the robot spin in one place could mess up the odometry and have the robot position be registered far away from the actual playground. Therefore it is best to run the robot along the circumference of the playground to get a good mapping.

Exercise 6

(d) In a centralised system, a single unit fuses information from all the individual robots and processes information using e.g. Kalman filter. A single unit would also give commands to all robots and tell each what they have to do in order to achieve a predefined global pattern.

In a decentralised system, each robot make their own decisions based on their own sensor inputs and through communications with neighbours. There's no one central processing unit for all robots.

A centralised system is best used when where's a centralised optimisation objection, such as minimum dispatch time in a warehouse. In the context of this exercise, we are trying to make all of the robots reach their respective destinations as quickly as possible, which is a centralised objective and can benefit from having a centralised system to achieve optimality.

However, if a problem occurs with the central processor, all robots in the centralised system would break down. A decentralised system would not have a single-point failure, which makes it much more robust. The challenges of a decentralised system is that it depends on the reliability of asynchronous communications.

Overall, a centralised system could guarantee completeness and optimality provided that the centralised system have perfect control over all robots and if all robots work as intended. This could be achieved in simulation, but would be very difficult in practice based on our experience in working with the robots. The centralised system would not be very robust in response to events such as temporary lag or even break down of individual robots. And since the pattern is predetermined, the system can suffer from increasing build up of small errors due to noise in sensing and control and communication with the central unit.

A decentralised system is more robust to noise as each robot communicates with one another and correct their controls accordingly all the time. It is also more robust to failure in single robots. However, these all depend on reliable asynchronous communications, which again might only be achievable in simulation. There is also no guarantee for perfect optimality compared to centralised control, although robots can converge to optimum as time progresses.