4M20 Assignment 2

Tom Ryan - tr452

November 27, 2022

1 Exercise 1

1.1 Part (a)

To cause the robot to travel from the start point $\mathbf{r}_s = [-1.5, -1.5]$ to the goal point $\mathbf{r}_g = [1.5, 1.5]$ we can create a velocity field as a function of position $\mathbf{V}_{goal}(\mathbf{x})$. We want the robot to constantly move towards the goal, which is the direction $\mathbf{r}_{g/x} = \mathbf{r}_g - \mathbf{x}$. We want to limit the robot's speed to u_{max} and so the velocity field is given by $\mathbf{V}_{goal}(\mathbf{x}) = u_{max} \frac{\mathbf{r}_{g/x}}{\|\mathbf{r}_{g/x}\|}$. The path from this velocity field is shown in figure 1, the robot moves in a straight path from the start to the goal point.

1.2 Part (b)

In order to avoid the obstacle, we want to create a vector field that both: moves the robot radially away from the obstacle, and has decreasing magnitude moving away from the obstacle. We ensure the correct direction by taking obstacle positions $\{\mathbf{r}_o^{(i)}\}_{i=0}^{N-1}$ and choosing a vector field of the form

$$\mathbf{V}_{obstacle}(\mathbf{x}) = -\sum_{i=0}^{N-1} k^{(i)} \mathbf{r}_{o/x}^{(i)} = \sum_{i=0}^{N-1} k^{(i)} (\mathbf{x} - \mathbf{r}_0^{(i)})$$

Where $k^{(i)}$ is a scaling term that we can set so that the vector field decreases in magnitude further from the obstacle. To achieve this we set $k^{(i)} = a^{(i)} \exp(b^{(i)} |\mathbf{r}_{o/x}^{(i)}|)$ where $a^{(i)}$ and $b^{(i)}$ are constants to be set. If we set $b = \frac{\log(u_{max}) - \log(a)}{R^{(i)}}$ where $R^{(i)}$ is the radius of the particular obstacle, this ensures the velocity field equals u_{max} at the edge of the obstacle. We can then tune a to change the slope of the function given this constraint, I found $a = 0.8$ to ensure the obstacle is avoided, without effecting the general path of the robot too much. Figure 2 shows the path with this vector field, the robot moves radially away from the obstacle, but as it is so far away it only moves a short amount.

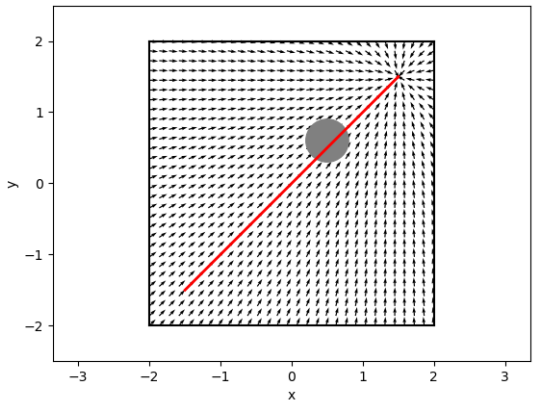

Figure 1: Path of robot with goal-driven potential field

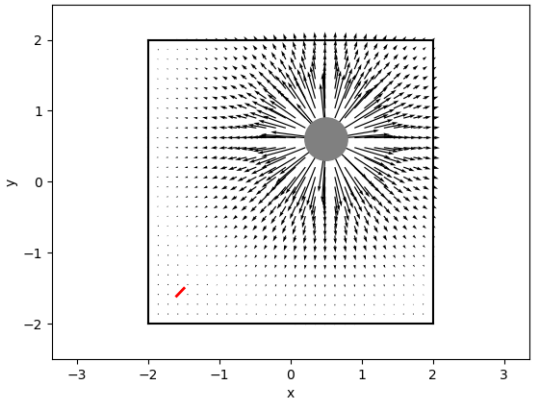

Figure 2: Path of robot with obstacle-avoidance potential field

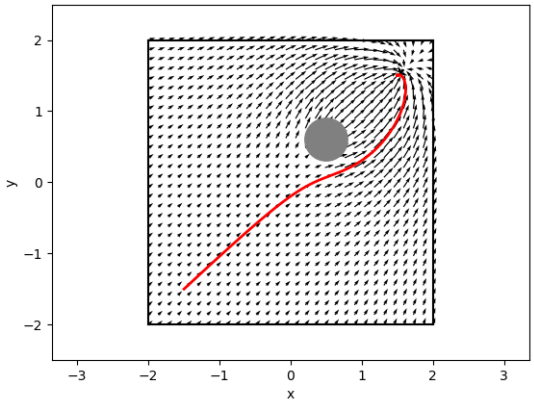

Figure 3: Path of robot with both goal-driven and obstacle-avoidance potential field

1.3 Part (c)

Figure 3 shows the path taken with a summation of the two vector fields, one that is goal driven, and one that aims to avoid the obstacle. The vector field is now given by $\mathbf{V}(\mathbf{x}) = \mathbf{V}_{goal}(\mathbf{x}) + \mathbf{V}_{obstacle}(\mathbf{x})$. The two potential fields combine in their effects so that the robot moves towards the goal, but has velocity moving away from the obstacle when it is close. This can be seen in figure 3, the field generally points towards the goal, but it points around the obstacle when it is close.

1.4 Part (d)

Were we to put the obstacle at $[0, 0]$ the obstacle avoidance would not work correctly, this is because both the goal field and the obstacle avoidance field would be symmetric about the line connecting the start and goal positions. So given that the robot starts along this line, the field would never be able to move it off this line, and it would fail to avoid the obstacle. In fact, any obstacle position along this diagonal will lead to failure of the obstacle avoidance. A solution to this would to be add a small field component in either the upwards, or the rightwards direction, to ensure that the robot could move off this invariant line in the field. We can run the script with the obstacle at $[0, 0]$ to verify this, and as we expected we get a path along the diagonal as in figure 4.

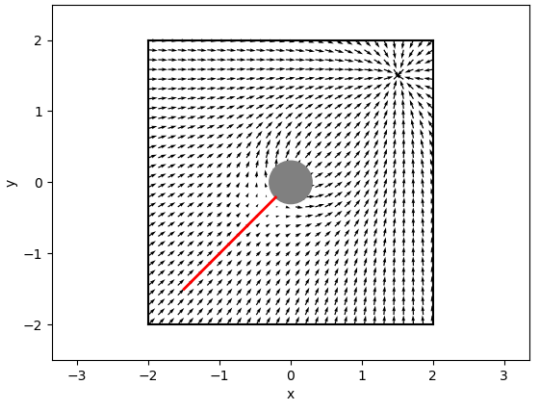

Figure 4: Path of robot when obstacle is placed at $[0, 0]$

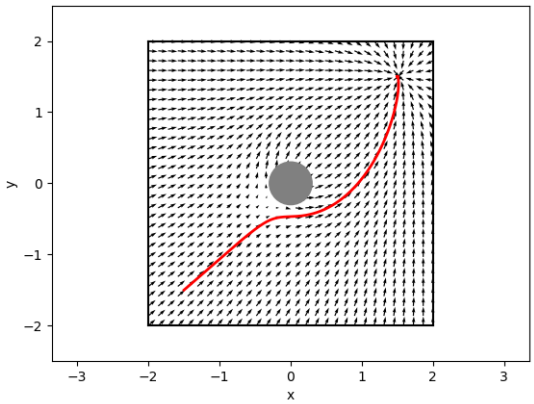

Figure 5: Improved path of robot when obstacle is placed at $[0, 0]$

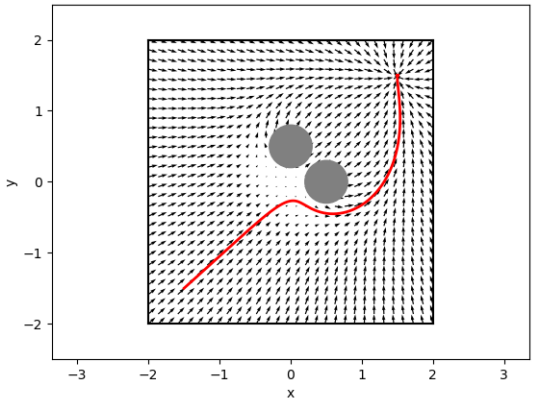

Figure 6: Path of robot with two obstacles

1.5 Part (e)

To avoid the situation described in part (d) we can add a small constant component to our field in the upwards or rightwards (+ve x or +ve y) directions. To do this we update our goal-driven velocity field to become:

$$\mathbf{V}_{goal}(\mathbf{x}) = u_{max}\frac{\mathbf{r}_{g/x}}{\|\mathbf{r}_{g/x}\|} + \begin{bmatrix}\epsilon \\ 0\end{bmatrix}$$

We can set epsilon as required, a large epsilon ensures the field will not be stuck on the invariant line but adds a large directional bias to the field. Generally a small epsilon is enough to move the robot away from the invariant line, a value $\epsilon = 0.05$ gives the robot path shown in figure 5, which is a sufficient solution to the problem that is not impacted too heavily by directional bias.

1.6 Part (f)

If we add a second obstacle, so that we have two obstacles at $[0.5, 0]$ and $[0, 0.5]$, the solution I have proposed still works, the result is shown in figure 6. However in different stages of testing I found that if the constant a in the obstacle-avoidance velocity field was not set well, the summation of two obstacle-avoidance fields could lead to large values and cause the robot to move outside of the arena boundaries. A potential solution to this issue is to penalise the magnitude of the robots position, so that it tends to move towards the arena center. This is achieved by updating the obstacle-avoidance field equation as follows:

$$\mathbf{V}_{obstacle}(\mathbf{x}) = \sum_{i=0}^{N-1} k^{(i)}(\mathbf{x} - \mathbf{r}_0^{(i)}) - \eta\mathbf{x}$$

Where η is a hyper-parameter that can be tuned.

2 Exercise 2

2.1 Part (a)

Since our robot is non-holonomic we cannot directly control it using the $\dot{\mathbf{x}}$ values we get from our velocity field in exercise 1. Instead we control a holonomic point a distance ϵ in front of the robot, and use feedback linearization to in turn control the robot. Given a holonomic point with state $[\mathbf{x}_p, \dot{\mathbf{x}}_p]$ a distance ϵ in front of the robot, we set $\mathbf{x}_r = [u, \omega]$ using the equations:

$$u = \dot{x}_p\cos(\theta) + \dot{y_p}\sin(\theta)$$

$$\omega = \frac{1}{\epsilon}(\dot{y_p}\cos(\theta) - \dot{x}_p\sin(\theta))$$

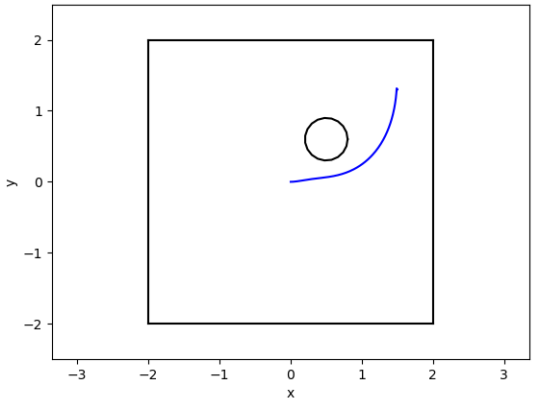

Figure 7: Path of non-holonomic robot

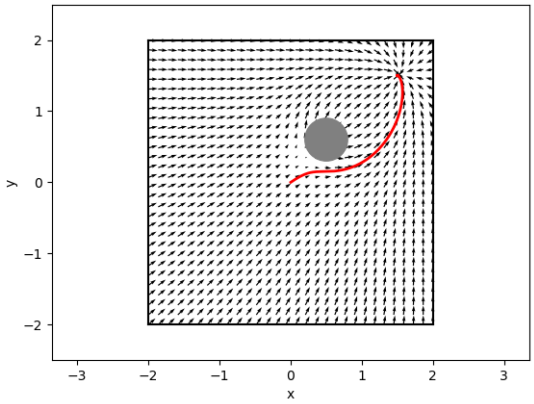

Figure 8: Path of holonomic point

Where θ is the global orientation of the robot.

2.2 Part (b)

Feedback linearization is used because our velocity field can only be used to control a holonomic point, it assumes that the state-space in which it controls a point has every point reachable. Our differential-drive robot however is not able to do this, its movement is defined by $[u, \omega]$, so it could not, for example, move solely in the x direction, if it was not facing that way. Feedback linearization allows us to control a holonomic point using our velocity field $\mathbf{V}(\mathbf{x})$ and then propogate the movement of that point to the non-holonomic movement of the robot, as if the holonomic point was at the end of a rod connected to the robot. We can move the end of the rod however we like, and the robot will follow in a non-holonomic manner.

2.3 Part (c)

Figure 7 shows the trajectory of the non-holonomic robot moving according to the movement of the holonomic point dictated by $\mathbf{V}(\mathbf{x})$. The robot avoids the obstacle as expected and converges to the goal point. We can compare the robot movement to the movement of the holonomic point from the start position as shown in figure 8. We see a slightly differing path, as the robot is being driven by the feedback linearization of the holonomic point. It is as if the robot averages out the path taken by the holonomic point. A reduced ϵ would make the robot follow the holonomic path more closely

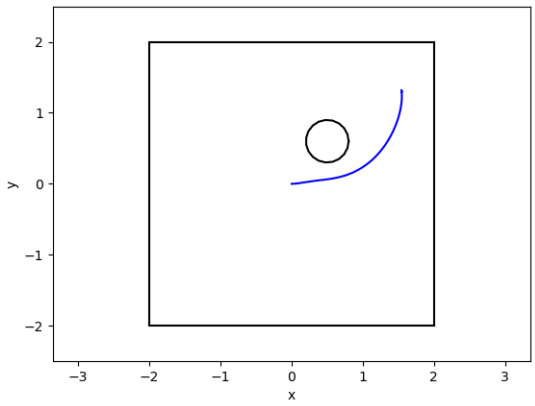

Figure 9: Path of non-holonomic robot with relative coordinates implementation

but would lead to large accelerations. We have to be sure that the potential field does not lead to the holonomic point, and thus the robot, exceeding the maximum speed - as this leads to wheel slip and the robot will not follow the path correctly.

2.4 Part (d)

Instead of using absolute positions of obstacles, and the start and goal positions, we can use relative positions based on the local robot coordinate system. As we are computing vectors and their gradients between locations in the arena, so as long as all vectors are in the same coordinate system it should not affect the results. This is useful because if we did not have the absolute coordinates of the obstacles and target positions, and we instead acquired them from sensor values, this would be in the robots own coordinate system and not in the global one. To compute the relative vectors we take the x and y coordinates of the absolute robot pose $\mathbf{r}_r^{(xy)}$ and the object position $\mathbf{r}_o$ and compute $\mathbf{r}_{r/o} = \mathbf{r}_r^{(xy)} - \mathbf{r}_o$. This is a relative x, y vector between the robot and the obstacle. Then to convert this to the robots coordinate system (which is at a different angle) we compute $\alpha = \theta - \arctan(\frac{\mathbf{r}_{r/o}^{(y)}}{\mathbf{r}_{r/o}^{(y)}})$ and the relative position of the object in the coordinate system of the robot is then $[|\mathbf{r}_{r/o}|\cos(\alpha), -|\mathbf{r}_{r/o}|\sin(\alpha)]$. Using this approach the path of the robot is as shown in figure 9.

3 Exercise 3

3.1 Part (a)

To sample a random position in the arena for an RRT algorithm, that is in a valid position, we can generate a joint uniform random variable $[x, y]$ where $x, y \in [-2, 2]$, this is the relevant arena size (between start and goal points). Given this random position we can use the `occupancy_grid` object which has a method `is_free` that determines if a point in continuous coordinates is free in the occupancy grid, which is a discrete representation of the occupancy of the arena. If the randomly sampled point is free, we return it, otherwise we sample another and repeat.

3.2 Part (b)

The `adjust_pose` function takes a node corresponding to initial position and direction $[x_0, y_0, a_0]$, and a final position $[x_1, y_1]$. The aim is to determine if an arc between the two nodes can be generated that does not collide with an obstacle. Given that we have the two end positions of the arc and a tangent, the curve is fully defined. We can thus compute the direction at the final position a_1 by computing the angle (in global coordinate system) of the vector between the start and end points θ. We then

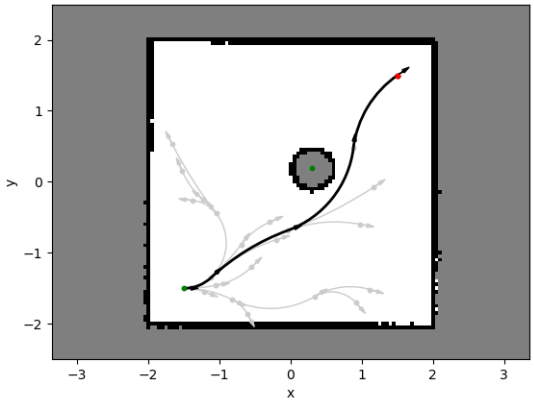

Figure 10: RRT algorithm

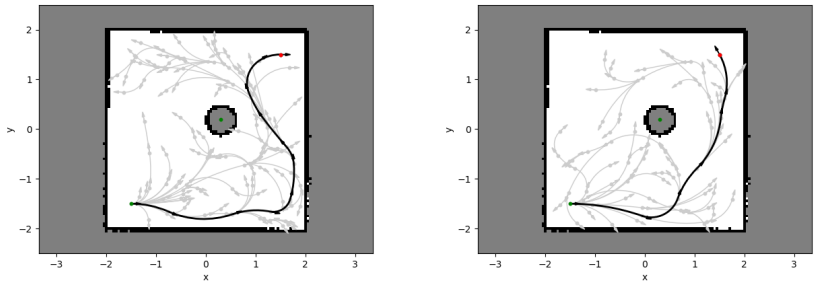

Figure 11: RRT algorithm taking inefficient route to goal

have that

$$a_1 = 2a_0 - \theta$$

$$\theta = \arctan(y_1 - y_0, x_1 - x_0)$$

Where $\arctan(a, b)$ computes the angle between the positive x axis and the point (x, y). Equivalent to `np.arctan2` in Python. Now that there are two fully defined nodes corresponding to the start and end poses we can use `find_circle` to acquire the center and radius of the arc between the two poses. From here we need to check that the arc does not pass through invalid points by iterating over points along the arc. We can compute the angle of the vector between the arc center and the start point θ_i and then increase this by $d\theta$ each iteration until we reach θ_f which is the angle of the vector between the arc center and the end point. For each θ in-between we can compute the normal of the vector between the center and the point on the arc $\hat{\mathbf{n}} = [\cos\theta, \sin\theta]$, the point on the arc is then $R\hat{\mathbf{n}} + \mathbf{r}_c$ where R is the radius and $\mathbf{r}_c$ is the center of the arc. Each point can then be checked to ensure it isn't occupied. Figure 10 shows the RRT algorithm output with this implementation of `adjust_pose`.

3.3 Part (c)

The main drawback to the current RRT algorithm, is that it has no notion of efficiency in terms of choosing a path that leads to the robot reaching the goal more quickly. Figure 11 shows two instances where the robot takes convoluted, inefficient paths to reach the goal.

3.4 Part (d) [bonus]

Since we are already iterating over the arc between nodes to ensure it does not pass through a boundary, we can add a heuristic cost based on the length of arcs to our algorithm. This means that nodes will be selected that are shorter in arclength, favouring more efficient paths to the goal. Another form of

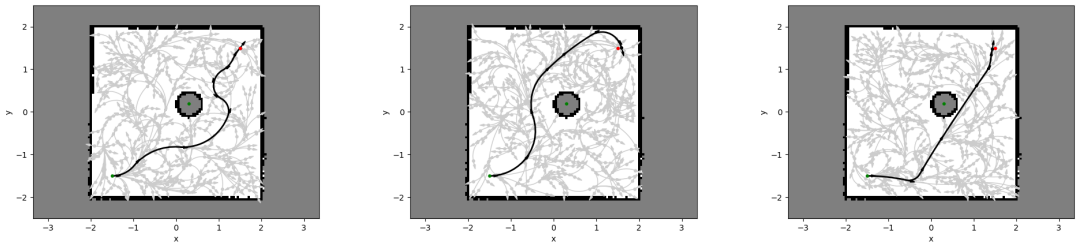

Figure 12: Improved RRT algorithm favouring paths with shorter lengths

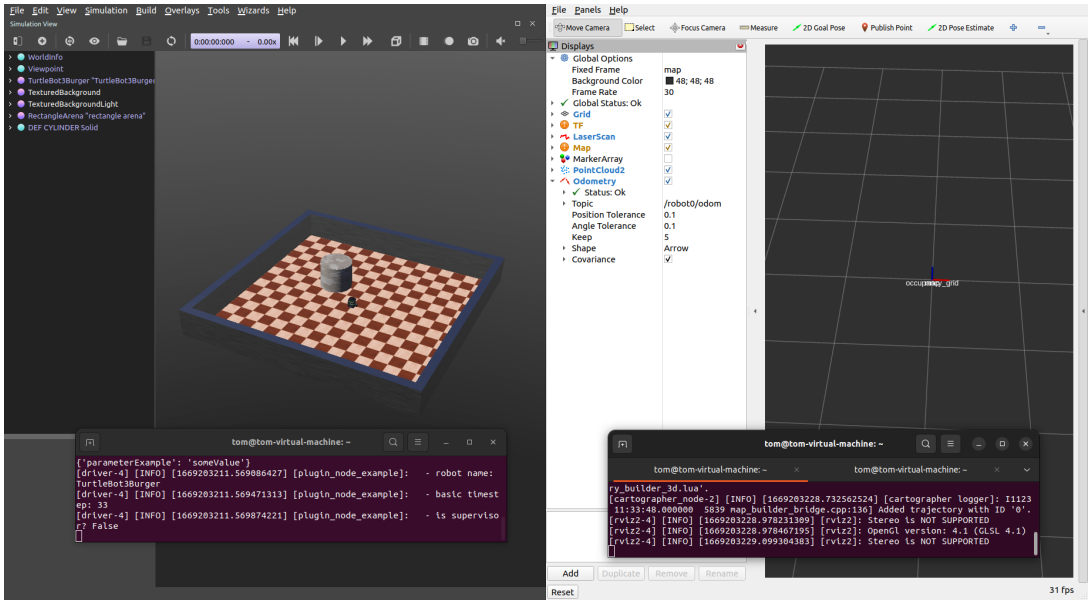

Figure 13: Webots and RViz setup

cost could be simply distance to node from the goal, but this means that the algorithm will spend much more time searching near the start position than near the goal because all of the costs will be higher. The effects of adding this arclength cost are shown in figure 12

4 Exercise 4

4.1 Part (a)

In order to combine navigation with the use of the RRT algorithm we must implement feedback linearization, this is done in the same manner as in exercise 2 part (a). We will use Webots and RViz to show the RRT navigation in action, this setup is shown in figure 13.

4.2 Part (b)

Motion primitives are pre-determined motions that the robot could take, representing the points that the robot could smoothly transition to[1]. An advantage of this is that when we compute the path using RRT we ensure the robot can follow it given our kinematics model. If we were not to use motion primitives, then for a non-holonomic robot we would need to solve differential equations with a two-point boundary problem, this would be considerably more computationally expensive. However in using motion primitives in this way, we are not fully exploring the full space of possible robot paths in $\mathcal{C}$ space. The problem is not probabilistically complete as we are making simplifications on the

[1] http://sbpl.net/node/48

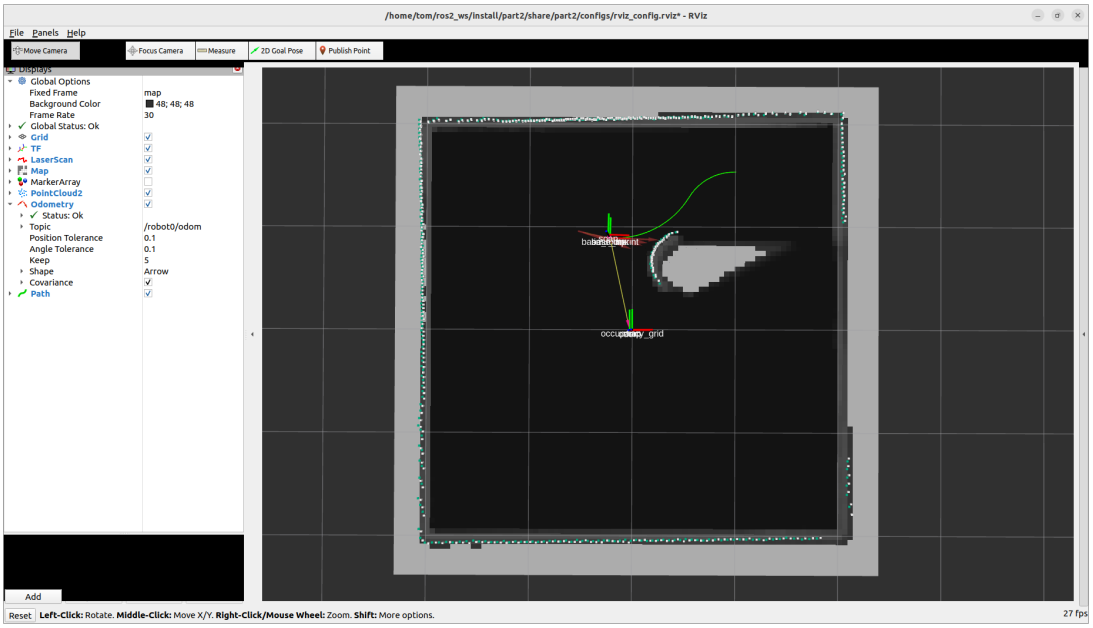

Figure 14: Navigation with RRT

dynamics of the robot, with the trade-off of not having to solve a two-point boundary problem. For our differential-drive robot, it would likely be feasible to solve this problem to give us a probabilistically complete navigation system, however for more complex systems with higher order differential equations governing the motion, it is clear why motion primitives are required.

4.3 Part (c)

Our RRT algorithm gives us a set of nodes for the robot to follow, we can control a holonomic point to move linearly between those points and then feedback linearization will cause the robot to follow the desired path. In order for the robot to follow the points we implement a `get_velocity` function which takes the robots current position and the positions of the nodes to follow. We can determine which path node the robot is closest to, and then determine if the robot is closer to the node after this node (in which case it is moving away from the closest node), or to the one before the closest node (in which case it is moving towards it). From here we can determine a line along which the robot should move.

In order to set a target, we manually generate it in RViz, this populates the `/move_base_simple/goal` topic.

4.4 Part (d)

We can visualize the path the robot takes in RViz, this is shown in figure 14. The robot updates the path to take in real time. This ensures that when the robot explores more of the map and the occupancy grid is updated, that this is considered in the path planning.

4.5 Part (e)

If we allow the robot to move, and then restart the SLAM script, we get the effect seen in figure 15. What is happening here is that we re-initialize the simultaneous localization and mapping (SLAM) algorithm, which assumes an initial odometry of $[0, 0, 0]$. As such the algorithm believes the robot is at the origin with zero yaw, and tries to interpret the data received from the scan to map the arena based on this assumption. As such we see that the unknown grey areas are not consistent with the particles from the scan based on the true robot position. We also observe an offset between the true robot position and the odometry.

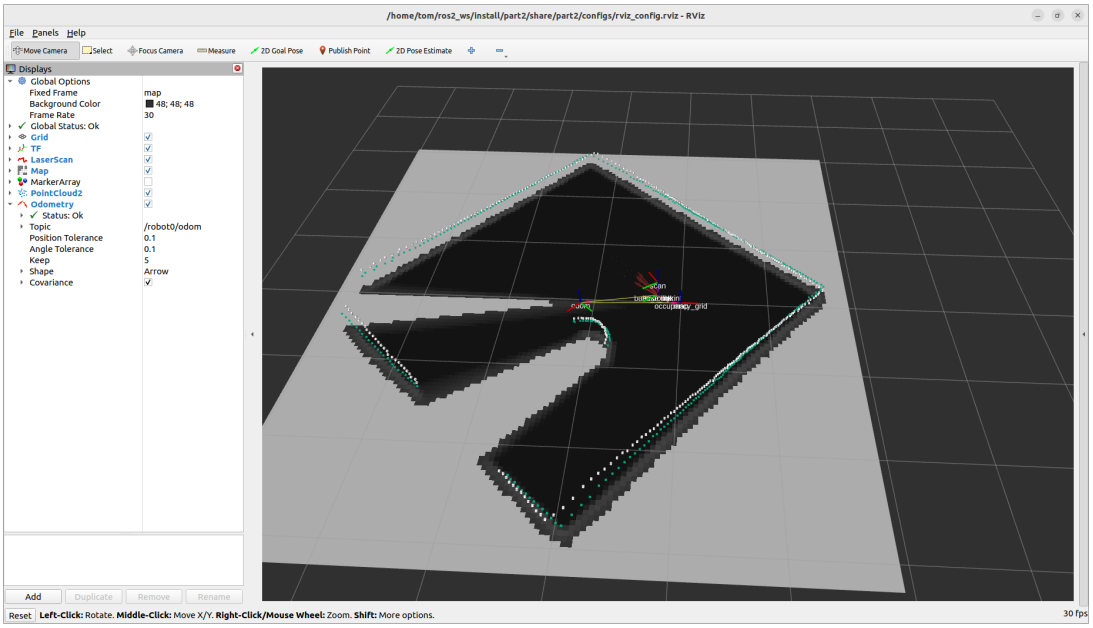

Figure 15: Effects of restarting SLAM once the robot has already moved

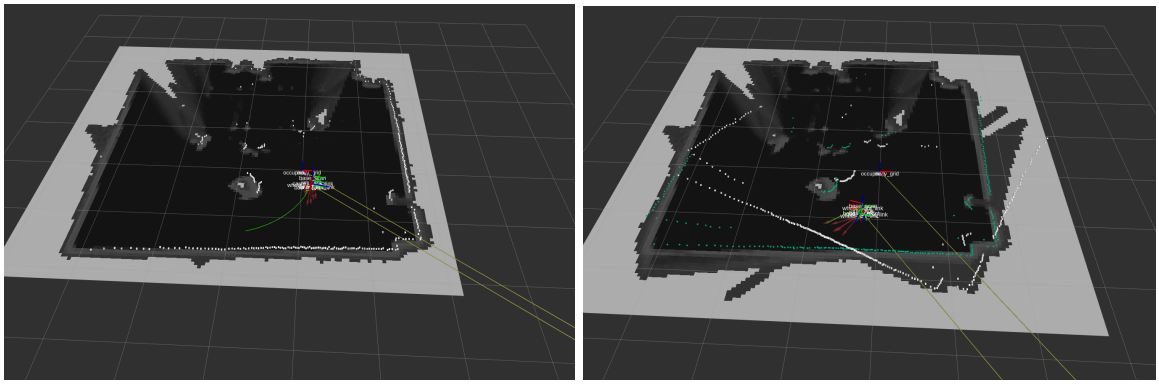

Figure 16: Path of real robot in RViz

5 Exercise 5

We can use our navigation with RRT algorithm on real robots. This is set up by connecting to the real robot using ssh, and publishing to topics accessible on a LAN. We can view the movement of the real robot in RViz similarly to in exercise 3, and this corresponds to the movement of the real robot. We see that when the RRT path is computed, the robot is exploring the arena and thus areas in the occupancy grid are revealed, this allows the SLAM algorithm to map more of the area than before and thus the RRT path is updated. This process repeats to build a full representation of the arena and thus the best path.

6 Exercise 6

6.1 Part (a) [bonus]

By initialising the multi-robot system, there are now five `cmd_vel` topics that can be published to, making each robot move. This can be achieved using the command
`ros2 topic pub -r 10 /cmd_vel geometry_msgs/Twist {linear: {x : 0.5, y : 0, z : 0}, angular: {x : 0, y : 0, z : 0}}`.

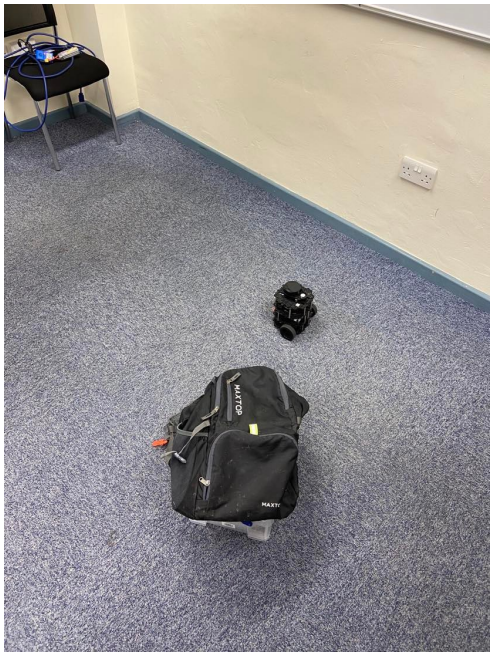

Figure 17: Final position of real robot

```python
def _match_destinations(self, robots, destinations):
    # we are happy with brute force here -> 5! = 120
    if len(robots) == len(destinations) == 1:
        return self._get_distance(robots[0], destinations[0]), {robots[0] : destinations[0]}
    else:
        cost = 9999
        best = {}
        for i in range(len(destinations)):
            c, m = self._match_destinations(robots[1:], [x for j,x in enumerate(destinations) if j!=i])
            m[robots[0]] = destinations[i]
            c += self._get_distance(robots[0], destinations[i])**2
            if c < cost:
                cost = c
                best = m
    return cost, best
```

Figure 18: Code snippet for labelling. Robots and destinations are lists of indexes for the robots and the destinations

6.2 Part (b) [bonus]

For our multi-robot problem we have a random initialisation of five robots which are unlabelled with respect to their destinations. We want to label them based on minimizing the summed squared distance between the robots and their destinations. But first we need the global positions for each robot, because we only have their local positions from the odometry (which is always 0). We can access the `/gps` topic to obtain global positions to be used in our labelling problem. For more robots, a greedy algorithm would be advisable for labelling destinations, however since we only have five robots, a brute force $\mathcal{O}(n!)$ complexity (5!=120) is fine. As such, we implement a recursive algorithm as described in 18. An implementation of this in simple python is shown in figure 19.

6.3 Part (c) [bonus]

We now look to creating an avoidance mechanism, we use a reciprocal velocity obstacle algorithm for this. We first need to ensure we have the global yaw of each robot, as well as their position, for this we subscribe to the `/imu` topic. We then want to define a reciprocal velocity obstacle region for each

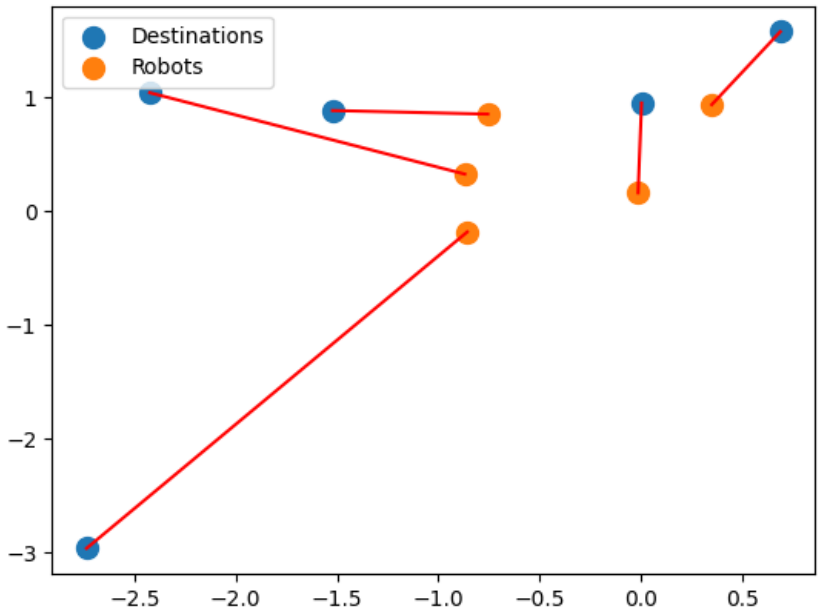

Figure 19: Labeling algorithm implemented in simple Python

robot, this is given using the following equations[2]

$$\gamma(\mathbf{p}_i, \mathbf{v}_i) = \{\mathbf{p}_i + t\mathbf{v}_i | t > 0\}$$
$$VO_B^A(\mathbf{v}_B) = \{\mathbf{v}_A | \gamma(\mathbf{p}_A, \mathbf{v}_A - \mathbf{v}_B) \cap B \oplus -A \neq \emptyset)\}$$
$$RVO_B^A(\mathbf{v}_B, \mathbf{v}_A) = \{\mathbf{v}'_A | 2\mathbf{v}'_A - \mathbf{v}_A \in VO_B^A(\mathbf{v}_B)\}$$

Where $\mathbf{p}_i$ is the position and $\mathbf{v}_i$ the velocity of a particular robot.

This gives us a region for a particular robot to not move into given the path of another robot. To account for all robots we need to consider the space

$$\bigcup_{j \setminus i} RVO_j^i$$

In practise, we can convert our space into a discrete grid, and for each robot we set a grid value to 1 if it is in the RVO space and 0 otherwise. Then we simply sum over grids and cap any values at 1. This gives us a discrete binary grid of RVO space. From here we set a robots velocity to the grid space that has a value of 0 with the closest Euclidian distance to the desired velocity (which always points directly towards the goal).

A key limitation of this solution is that we do not have holonomic robots. We must use feedback linearization to move the actual robots given our computed velocity. Since the robots are randomly initialized, it is not unlikely that the holonomic points velocity makes a large angle with the direction the robot is facing. This will lead to a large discrepancy between the holonomic movement and the actual robot movement. This is not taken into account by our RVO model, and the robots could collide nonetheless.

6.4 Part (d)

The approach implemented is a centralized controller for the multi-robot system. This approach boasts better performance than a decentralized approach, with a single controller managing syncing of messages to each robot, granting robustness to delays. A centralized approach is also more robust to

[2] https://gamma.cs.unc.edu/RVO/

noise in communications and sensing as there are generally several messages and sensor readings with redundant information across different robots and so this can be used to reduce the effects of any noise. This centralized control poses some disadvantages however, there is a singular point of failure as all the robots are reliant on a single control system. Reliable communication is essential, as the robots are completely reliant on the central control unit, drones performing searches in mountainous regions for example could not used centralized control because they would not be able to have a reliable communication channel to the central unit. Additionally a centralized control unit is not as scalable, you cannot add another robot to a fleet with centralized control and expect the control unit to be able to continue to perform optimally, the software would need to be edited to account for the additional robot.

A decentralized system offers an alternative solution with some advantages. Firstly a decentralized system is much more robust to failure, a single robot in a fleet can fail and the fleet will continue to function properly. Similarly it is scalable, as all the robots act independently adding another robot to the fleet means that it will still act optimally. Additionally a decentralized system only relies on localised communication, so can be used in environments where there would be no way to communicate with a central control unit. The issue with a decentralized system however is that its performance is generall sub-optimal compared to a centralized system, and the logic and programming is more difficult. Since the robots are communicating with each other asynchronously it is difficult to sync time-stamps and delays and overhead can arise. Since inputs are asynchronous, and there is no central system to utilise redundancy to reduce the effects of noise, rumour propagation can occur where a single noisy input is propagated and behaviour is based on this erroneous input. Care must be taken when programming decentralized systems to ensure robustness to these issues.