Mobile Robot Systems
Assignment 2

Zach Lambert Pembroke College March 2021

1 Potential Field Method

1.1 Exercise 1

(1a) Implementing a velocity field to reach a goal

The purpose of the potential field for the goal is to drive the trajectory towards the goal, so is an attractive potential field. For a goal $\boldsymbol{x}_g$, a sensible potential field is:

$$U_a(\boldsymbol{x}) = \frac{1}{2}|\boldsymbol{x} - \boldsymbol{x}_g|^2$$

The velocity field $v(\boldsymbol{x})$ should point down the gradient of the potential field, giving:

$$V_a(\boldsymbol{x}) = -\nabla U_a(\boldsymbol{x}) = \boldsymbol{x}_g - \boldsymbol{x}$$

hence the velocity field always points towards the goal and decreases in magnitude as the trajectory approaches the goal.

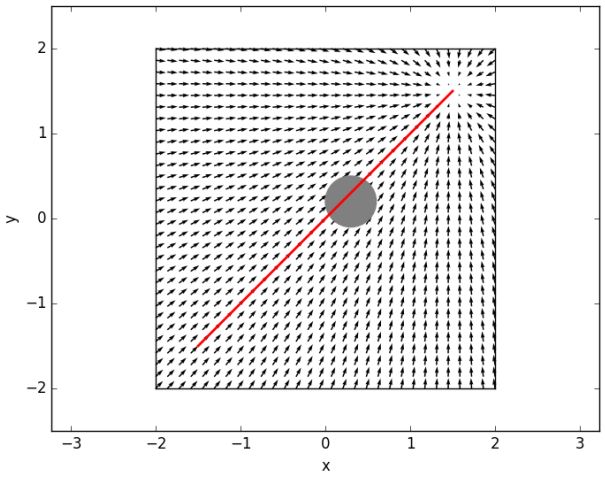

Figure 1: Trajectory returned using the potential field method with only an attractive field towards the goal.

Figure 1 shows the resultant trajectory, which moves in a straight line to the goal, ignoring the obstacle since this isn't taken into account in the velocity field yet.

(1b) Implementing a velocity field to avoid obstacles

The potential field for obstacles should be repulsive. The velocity field should always point away from the obstacles, where the magnitude tends to a maximum as the distance to an obstacle decreases to zero.

The potential function $U_r(\boldsymbol{x})$ is a sum of all the repulsive potential functions from each obstacle:

$$U_r(\boldsymbol{x}) = \sum_{i=1}^{N} U_{r,i}(\boldsymbol{x})$$

where an obstacle of position $\boldsymbol{x}_i$ and radius r_i has the potential function:

$$U_{r,i}(\boldsymbol{x}) = \begin{cases} U_{\max} \exp\left(-\frac{|\boldsymbol{x}-\boldsymbol{x}_0|-r_i}{\lambda}\right) & |\boldsymbol{x} - \boldsymbol{x}_i| \geq r_i \\ U_{\max} & |\boldsymbol{x} - \boldsymbol{x}_0| < r_i \end{cases}$$

where λ defines the scale of the potential field around the obstacle. An increase in distance $\lambda \ln 2$ will halve the value of the potential field.

Differentiating gives the velocity field:

$$\begin{aligned} V_{r,i}(\boldsymbol{x}) &= -\nabla U_{r,i}(\boldsymbol{x}) \\ &= \begin{cases} V_{\max} \frac{\boldsymbol{x}-\boldsymbol{x}_i}{|\boldsymbol{x}-\boldsymbol{x}_i|} \exp\left(-\frac{|\boldsymbol{x}-\boldsymbol{x}_0|-r_i}{\lambda}\right) & |\boldsymbol{x} - \boldsymbol{x}_i| > r_i \\ \boldsymbol{0} & |\boldsymbol{x} - \boldsymbol{x}_i| < r_i \end{cases} \end{aligned}$$

where $V_{\max} = \frac{U_{\max}}{2\lambda}$.

Therefore, the velocity field always points away from obstacle with maximum velocity $V_{\max}$ at the obstacle, which decays in magnitude as the distance to the obstacle increases.

These combine to give the total repulsive velocity field:

$$V_r(\boldsymbol{x}) = \sum_{i=1}^{N} V_{r,i}(\boldsymbol{x})$$

This velocity field should always overcome the attractive velocity field towards the goal when the robot is right next to the obstacle, at the very least. Therefore:

$$V_{\max} > \max_{\boldsymbol{x}}(|\boldsymbol{x}_g - \boldsymbol{x}|)$$

A factor of 2 larger was found to work well.

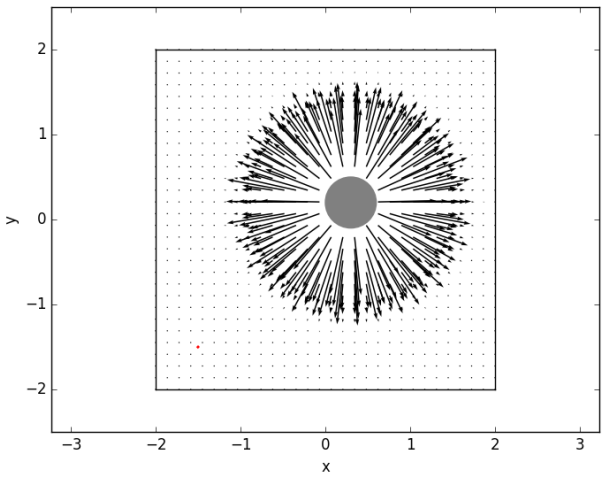

Figure 2: Velocity field and resultant trajectory with only the repulsive velocity field for avoiding obstacles.

Figure 2 shows the repulsive velocity field, which as expected, drives the trajectory away from the obstacles.

(1c) Combining the two fields

Combining the two fields gives the final velocity field:

$$V(\boldsymbol{x}) = V_a(\boldsymbol{x}) + V_r(\boldsymbol{x})$$

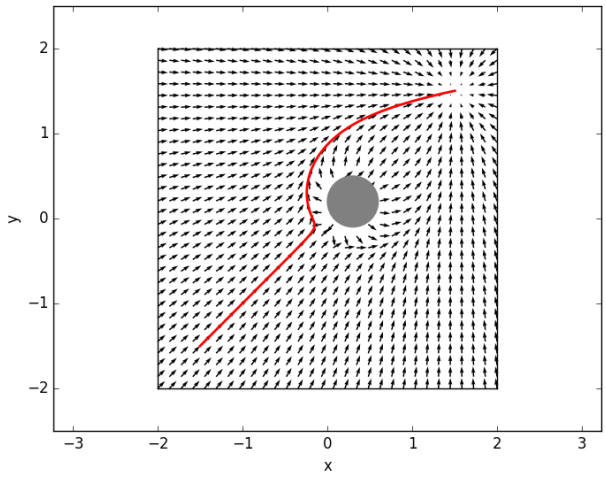

Figure 3: Combined velocity field and the resultant trajectory.

Figure 3 shows the velocity field and trajectory when combining the two velocity fields. The trajectory initially heads towards the goal in a straight line, but as it approaches the obstacle, the repulsive field from the obstacle pushes it away. This curves the trajectory around the obstacle.

(1d) Placing an obstacle at $[0, 0]$

When an obstacle is at $[0, 0]$, this creates a saddle point in the potential field in front of the obstacle. In this case, the start and end goal are in a straight line along the axis of the saddle point, so along this line, the trajectory enters a local minima, while it's at a local maxima along the perpendicular axis.

To avoid this, the trajectory simply needs to be perturbed slightly from the local maxima. This can be done by adding a small amount of noise to the velocity field.

(1e) Implementing the solution to the problem in (1d)

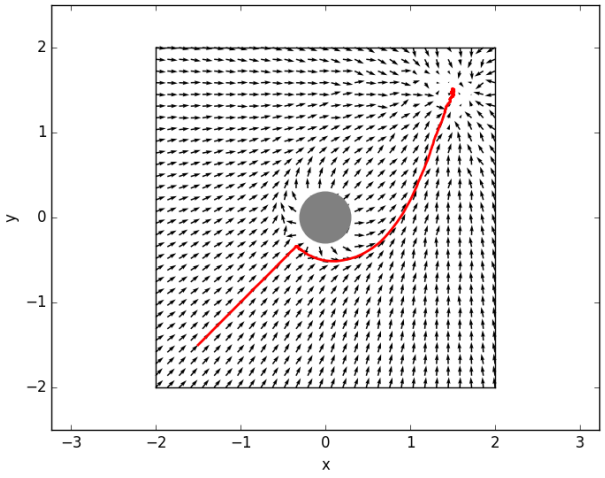

Figure 4: Trajectory when an obstacle is at $[0, 0]$, but with the issue in (1d) resolved by adding random noise to the velocity field.

A small amount of random Gaussian noise was added to the attractive velocity field, which perturbed the trajectory from the local maxima of the saddle point, allowing it to move around the obstacle, as shown in Figure 4

(1f) Adjusting solution to handle local minima

When two obstacle are placed near each other, a local minima is created between them. The solution from (1e) only handles saddle points, whereas local minima are a different problem.

The most robust solution to this problem is to combine gradient descent search and a random walk when local minima are encountered.[1]

This is implemented by having two operating modes: best first search (gradient descent) and random walk.

1. In best first mode, the trajectory follows gradient descent. When it reaches a local minima, characterised by a small velocity magnitude and distance to the goal exceeding a threshold, it will enter random walk mode.

2. In random walk mode, the trajectory randomly updates. When the trajectory moves back down the potential field, characterised by the current step having a positive component in the direction of the velocity field, it will return to best first mode.

Instead of using independent steps in random walk mode, a markov process was used. At each step, the trajectory would move with velocity:

$$V[t] = V_{\max}(\cos\theta[t]\boldsymbol{i} + \sin\theta[t]\boldsymbol{j})$$

and update the direction with Gaussian noise of standard deviation π:

$$\theta[t + dt] \sim \mathcal{N}(\theta[t], \pi)$$

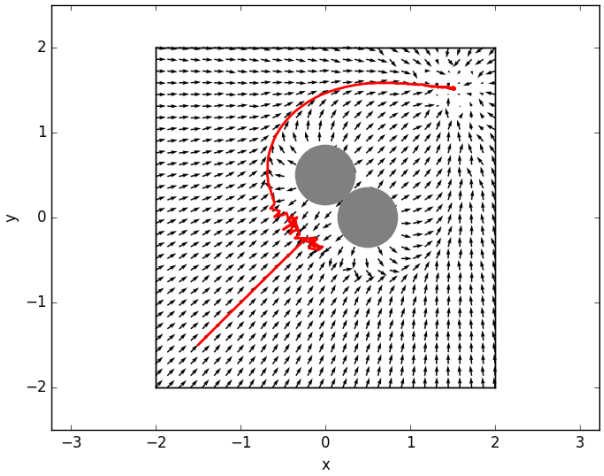

Figure 5: Trajectory returned when using the random walk method to escape local minima.

Figure 5 shows the trajectory returned by the random walk method, which becomes convoluted when trying to escape a local minima. To fix this, the final trajectory can be simplified by finding a subset of points along the trajectory that remain collision free between them.

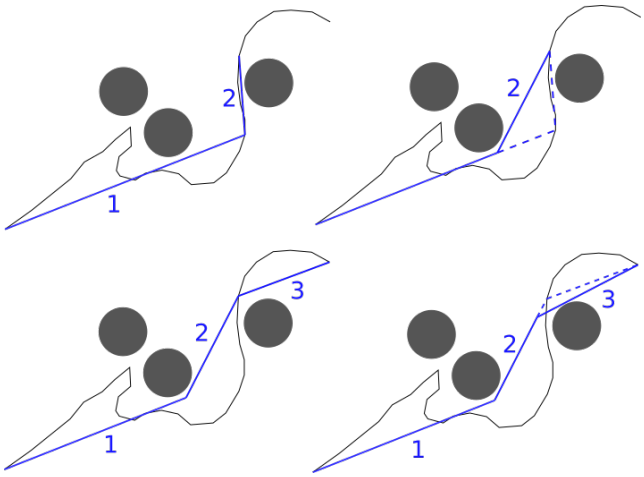

Figure 6: Algorithm used to simplify a trajectory.

Figure 6 shows the algorithm used to simplify the trajectory. In each iteration, it finds the next collision free segment from the original trajectory and adds this to the simplified trajectory. Following this, it back-tracks along the simplified trajectory to reduce the trajectory further. This continues until it reaches the end of the trajectory. This won't give the optimal trajectory, but gives a reasonable simplification and will remove the noise introduced by the random walk.

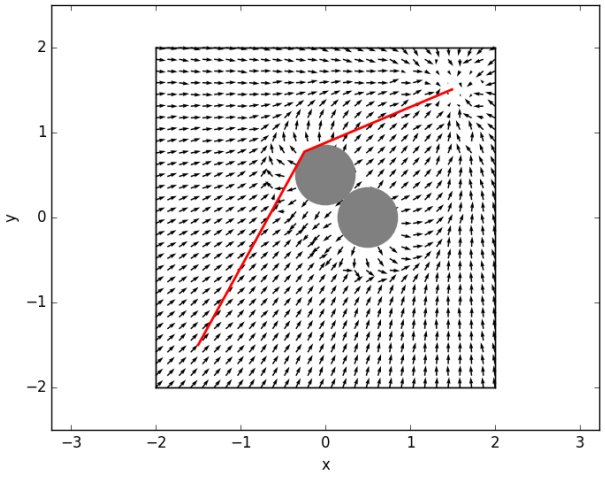

Figure 7: Trajectory returned using the random walk method to escape local minima after simplification.

Figure 7 shows the resultant trajectory after simplification. This escapes the local minima and gives a more optimal trajectory.

1.2 Exercise 2

(2a) Equations for feedback linearisation of a differential drive

For a robot at position $\boldsymbol{x}$ with orientation θ, the position $\boldsymbol{x}_p$ of a point P a distance ϵ in front is given by:

$$\boldsymbol{x}_p = \boldsymbol{x} + \epsilon \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

Differentiating gives the velocity of the point:

$$\begin{aligned} \dot{\boldsymbol{x}}_p &= \dot{\boldsymbol{x}} + \epsilon\dot{\theta} \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \\ &= u \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} + \epsilon\omega \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\epsilon\sin\theta \\ \sin\theta & \epsilon\cos\theta \end{bmatrix} \begin{bmatrix} u \\ \omega \end{bmatrix} \end{aligned}$$

This gives the forward kinematics equation:

$$\dot{\boldsymbol{x}}_p = J(\boldsymbol{x})\boldsymbol{u}$$

with control vector $\boldsymbol{u} = \begin{bmatrix} u & \omega \end{bmatrix}^T$.

(2b) The utility of feedback linearisation

With feedback linearisation, the control of the non-holonomic robot is implemented by control of the holonomic point P. By inverting the forward kinematics equation, this gives the inverse kinematics equation:

$$\begin{aligned} \boldsymbol{u} &= J(\boldsymbol{x})^{-1}\dot{\boldsymbol{x}}_p \\ \begin{bmatrix} u \\ \omega \end{bmatrix} &= \begin{bmatrix} \cos\theta & \sin\theta \\ -\frac{1}{\epsilon}\sin\theta & \frac{1}{\epsilon}\cos\theta \end{bmatrix} \begin{bmatrix} \dot{x}_p \\ \dot{y}_p \end{bmatrix} \end{aligned}$$

which depends on θ, defined by the differential equation $\dot{\theta} = \omega$.

(2c) Implementing the *feedback_linearized* function

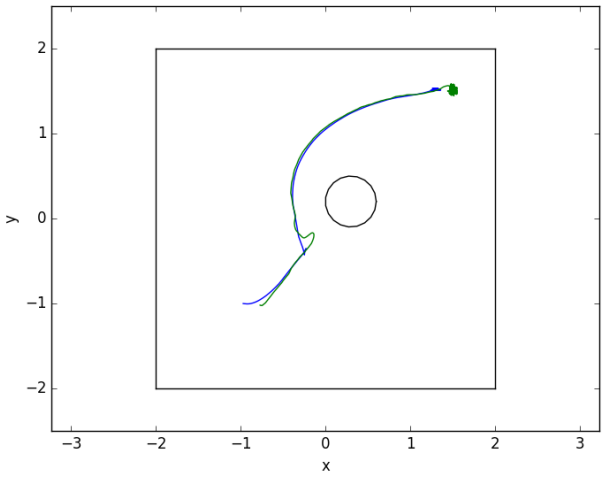

Figure 8: Trajectory followed by robot in Gazebo, following the velocity field from exercise 1.

Figure 8 shows the trajectory followed by the robot in Gazebo. It follows the velocity field from exercise 1, which combines the attractive field towards the goal and the repulsive field away from obstacles. This gives the desired velocity of the holonomic point P, set by u and ω using the inverse kinematics equation.

(2d) Adjusting the implementation to use relative pose

The robot has an absolute pose defined in the world frame, with absolute position $\boldsymbol{x}$ and orientation θ.

For an arbitrary point Q, this has an absolute position $\boldsymbol{x}_Q$ in the world frame. This position can be expressed in the robot frame R, denoted ${}^R\boldsymbol{x}_Q$, with the transformation:

$$ {}^R\boldsymbol{x}_Q = R(\theta)^T(\boldsymbol{x}_Q - \boldsymbol{x}) $$

where $R(\theta)$ is the standard rotation matrix, rotating a vector θ radians counter-clockwise.

When all positions are expressed in the robot frame, the feedback linearisation simplifies to:

$$ u = \dot{x}_p \quad \omega = \frac{1}{\epsilon}\dot{y}_p $$

2 Rapidly-Exploring Random Trees

2.1 Exercise 3

(3a) Implementing the *sample_random_position* function

Sample a position uniformly between the occupancy grid origin and the maximum position, found by adding (resolution × values.shape) to the origin.

The validity of the position is checked by the *is_free(position)* function provided by the occupancy grid. The position is sampled until a valid position is found.

(3b) Implementing the *adjust_pose* function

The current node has a pose consisting of position $\boldsymbol{x}_1$ and yaw θ_1. The final node has position $\boldsymbol{x}_2$ and yaw θ_2.

θ_2 is chosen such that a circular arc can connect the two positions. If $\phi = \text{atan2}(y_2 - y_1, x_2 - x_1)$, then:

$$ \theta_2 = \phi + (\phi - \theta_1) = 2\phi - \theta_1 $$

The *find_circle* function provided, returns the centre $\boldsymbol{x}_c$ and radius r of the circle. To check this arc is collision free, positions $\boldsymbol{x}[n]$ are checked along the arc, where:

$$ \boldsymbol{x}[n] = \begin{cases} \boldsymbol{x}_c + r \begin{bmatrix} \cos(\theta[n] - \pi/2) \\ \sin(\theta[n] - \pi/2) \end{bmatrix} & \theta_2 - \theta_1 > 0 \\ \boldsymbol{x}_c + r \begin{bmatrix} \cos(\theta[n] + \pi/2) \\ \sin(\theta[n] + \pi/2) \end{bmatrix} & \theta_2 - \theta_1 < 0 \end{cases} $$

Initialise $\theta[0] = \theta_1$.
Step $\theta[n+1] = \theta[n] + \Delta\theta$
End when $|\theta[n] - \theta_1| \geq |\theta_2 - \theta_1|$

$\Delta\theta$ is chosen such that the distance stepped along the arc is equal to the occupancy grid resolution, and has the correct sign, giving:

$$ \Delta\theta = \begin{cases} \frac{\text{(occupancy grid resolution)}}{r} & \theta_2 - \theta_1 > 0 \\ -\frac{\text{(occupancy grid resolution)}}{r} & \theta_2 - \theta_1 < 0 \end{cases} $$

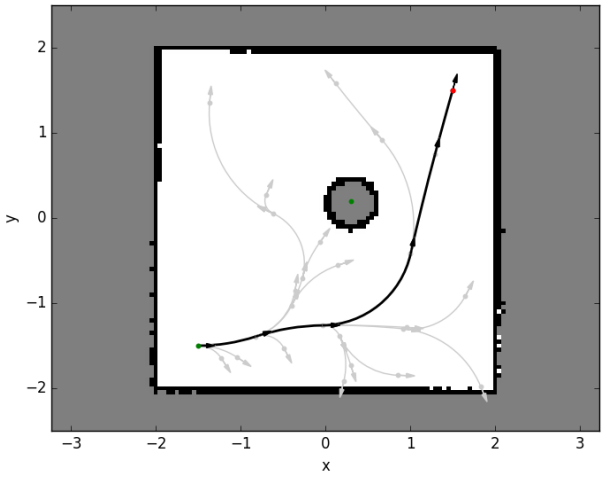

Figure 9: Trajectory found using RRT.

Figure 9 shows the trajectory found using the RRT implementation using this function. It avoids collisions and connects nodes with circular arcs.

(3c) Drawback of RRTs in the current implementation

The current RRT implementation is probabilistically complete and can guarantee that a solution will be found with enough iterations. However, it doesn't provide any guarantee on the optimality of the solution. This can be seen in testing the current implementation, which usually gives a sub-optimal solution.

One solution is to adapt the implementation to give the RRT* algorithm. Previously, when a new node is randomly sampled, a path is connected from the nearest collision-free node and added to the graph. RRT* is similar, but changes how the tree is expanded on sampling a new node.

Instead, the new node is steered towards the randomly sampled position. Following this, other nearby nodes are checked to find which gives the minimum cost for the new position, and the best is chosen as the parent instead. Finally, nearby nodes have their parents changed to the new node if this gives a lower cost.

(3d) Implementing RRT*

The implementation is based off of Algorithm 4, $Extend_{RRT^*}$, from "Incremental Sampling-based Algorithms for Optimal Motion Planning"[2].

One difference is that when changing the parent of the new node, and when changing the parent of nearby nodes, only the positions are kept the same. The yaw of nodes are adjusted to give a valid path between positions.

The algorithm is also adjusted such that instead of stopping when a solution is found, the tree is continually extended for the maximum number of iterations. When a new solution is found, it only replaces the old solution if the cost is lower.

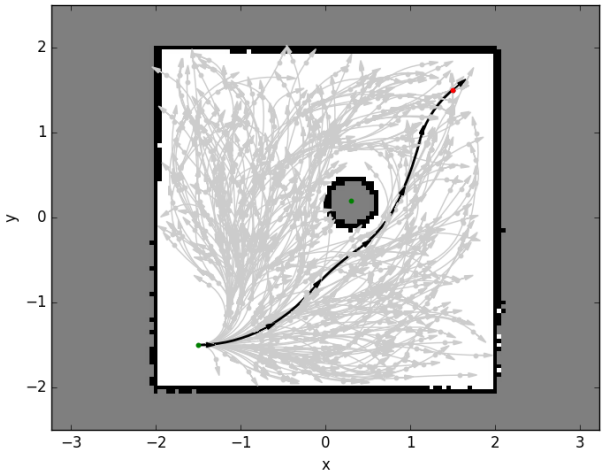

Figure 10: Trajectory returned using RRT*.

Figure 10 shows the trajectory found using RRT*. It repeatedly gave close to optimal solutions. This is much better compared to the simpler RRT implementation used previously. The drawback is the greatly increased computation time, but this could be improved by making use of more efficient data structures and algorithms.

2.2 Exercise 4

(4a) Implementing the *feedback_linearized* function and using with navigation

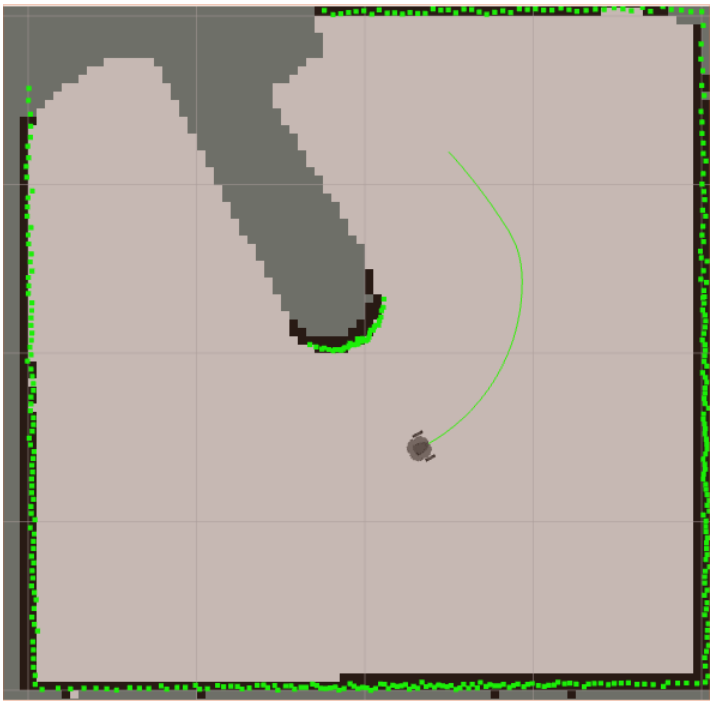

Figure 11: Screenshot of rviz with a trajectory planned to a 2D nav goal using RRT.

Figure 11 shows a trajectory returned when using the RRT implementation within ROS for navigation.

(4b) The advantages and disadvantages of motion primitives for path generation

RRT connects new nodes using motion primitives, connecting a circular arc from the previous node to a new node, satisfying yaw requirements. With this, it inherently models the non-holonomic motion constraints of the robot, and any generated path will be feasible.

For other sampling-based algorithms, determining whether two nodes can be connected requires more work.

Additionally, this allows more complex constraints such as minimum turning radius to be taken into account.

There are no obvious disadvantages, although other methods may be computationally cheaper.

(4c) Implementing the *get_velocity* function

For the current position $\boldsymbol{x}$, the two closest points on the trajectory $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are found, by stopping searching when $\boldsymbol{x}$ is between them, which occurs if:

$$\text{sign}(\boldsymbol{n} \cdot (\boldsymbol{x}_2 - \boldsymbol{x})) \neq \text{sign}(\boldsymbol{n} \cdot (\boldsymbol{x}_1 - \boldsymbol{x}))$$

where

$$\boldsymbol{n} = \frac{\boldsymbol{x}_2 - \boldsymbol{x}_1}{|\boldsymbol{x}_2 - \boldsymbol{x}_1|}$$

The velocity of the holonomic point P, $\boldsymbol{v}$ is set as:

$$\boldsymbol{v} = v_{\text{nom}}\boldsymbol{n} + kd\boldsymbol{n}_\perp$$

where $\boldsymbol{n}_\perp$ is perpendicular to n and d is the perpendicular distance to $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, along $\boldsymbol{n}_\perp$:

$$d = \boldsymbol{n}_\perp \cdot (\boldsymbol{x}_1 - \boldsymbol{x}) \quad \text{or} \quad \boldsymbol{n}_\perp \cdot (\boldsymbol{x}_2 - \boldsymbol{x})$$

v_{nom} is the nominal speed along the path, and k is the gain from the perpendicular distance d to the velocity along $\boldsymbol{n}_\perp$ to correct for this.

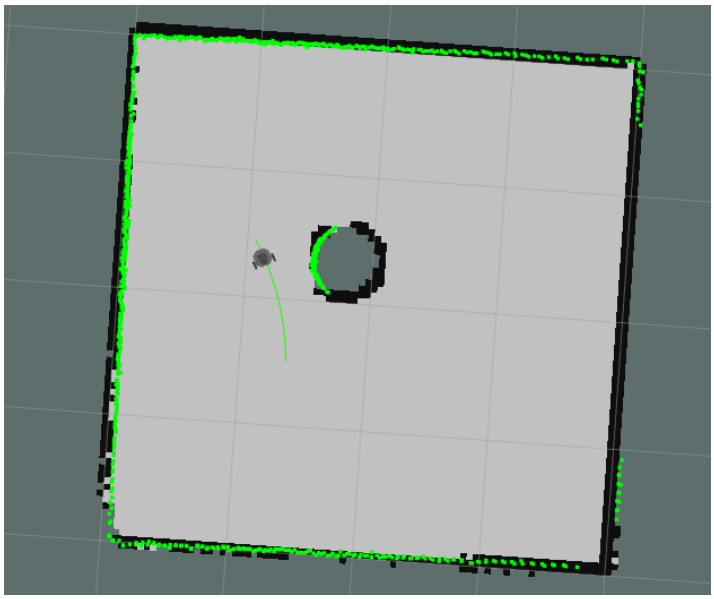

Figure 12: Screenshot of rviz with a trajectory planned to a 2D nav goal using RRT, where the robot is following the trajectory using the *get_velocity* function.

(4d) The purpose of *roslaunch exercises slam.launch*

This starts the launch file *slam.launch*, which loads parameters and starts nodes responsible for performing SLAM. This includes loading the robot description, doing forward kinematics with *robot_state_publisher* and starting SLAM with the *slam_gmapping* node in the *gmapping* package.

The SLAM algorithm simultaneously builds a map from the laser scan data and localises the robot within this map frame. Particle filter methods are used to combine estimates of pose from odometry and map data, to form the final pose estimate.

The RRT algorithm uses SLAM to provide the occupancy grid and estimate of the current robot pose. Then, when planning to a navigation goal, this starts at the current pose and avoids collisions with the provided occupancy grid.

3 References

[1] LaValle S., (2006) *Planning Algorithms*. Section 5.4.3, Randomized Potential Fields.

[2] Karaman S., Frazzoli E., (3 May 2010) *Incremental Sampling-based Algorithms for Optimal Motion Planning*. arXiv:1005.0416 [cs.RO].