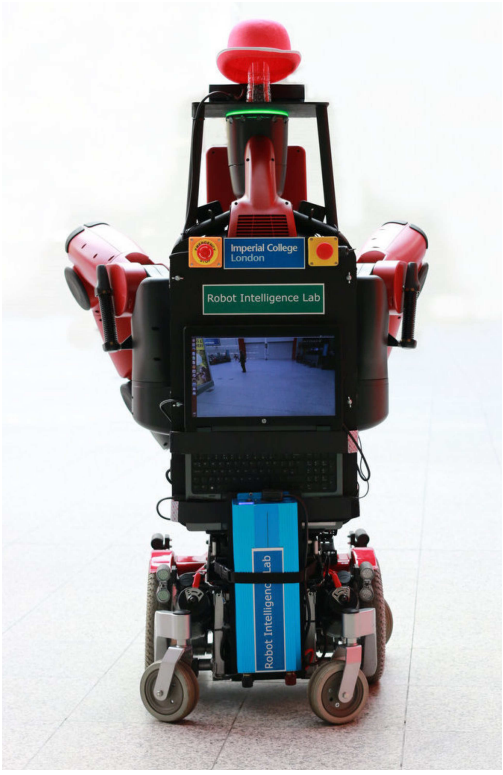
DE3 Applied Robotics **Coursework 2 - PLAN**

Module Leader: Dr Petar Kormushev Robot Motion Planning

1 Introduction

A well-implemented Motion Planning algorithm is essential for a robot to behave intelligently. From driverless cars to robot arms, every robot must perform motion planning of some sort. Previously, in Coursework 1 - MODEL, no motion planning was required because there were no obstacles or joint limits to hinder the robot arm's motion. In this coursework, we will look at motion planning for a mobile robot - specifically, for our very own Design Engineering's Natural Interaction Robot (Robot DE NIRO):



You can find more information about our platform, Robot DE NIRO, here: https://www.imperial.ac.uk/robot-intelligence/robots/robot_de_niro/

This coursework assumes you have already completed Coursework 1 - MODEL, which covers the getting started guide to the Virtual Machine. To save effort on your side, and avoid having to do repeated large downloads, you will continue using the same Virtual Machine as in Coursework 1 for this coursework.

Two new files will be used in this coursework. The files will be downloaded and placed inside the coursework 2 folder located in '/home/de3robotics/Desktop/DE3Robotics/src/coursework_2'. A description of the files is given in the table below.

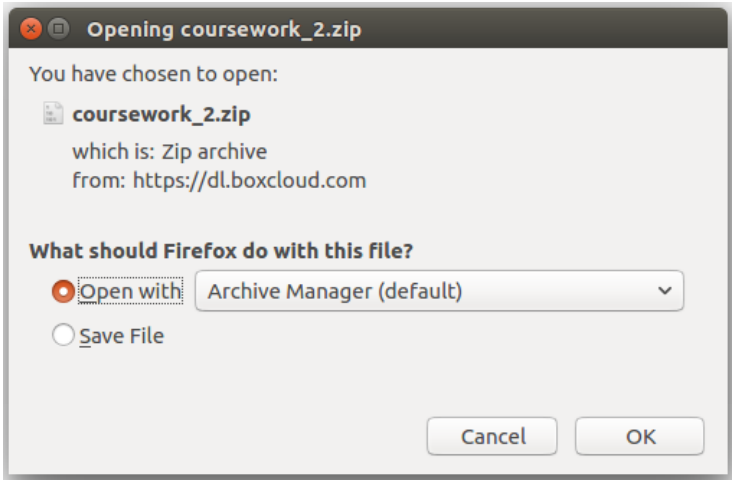
File	Description	Relative Path
`motion_planning.py`	Computes the desired motion plan of the robot, sends command velocity to the simulator.	src/motion_planning.py
`map.py`	Generates a map of the environment to use in motion planning algorithms.	src/map.py

2 Getting Started

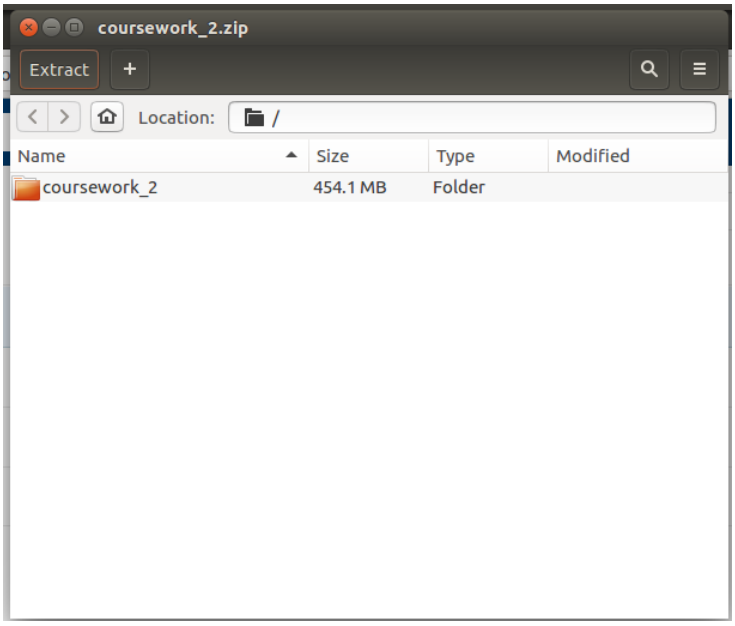
Before we can run anything, we need to download the `coursework_2` files for us to use. To do this, power on the Virtual Machine, open Firefox web browser (inside the virtual machine), and go to the following link:
https://imperialcollegelondon.box.com/s/w3j518onxc6lmjngih44utay6o0lhzg0

Go ahead and download the file `coursework_2.zip` (inside the virtual machine).

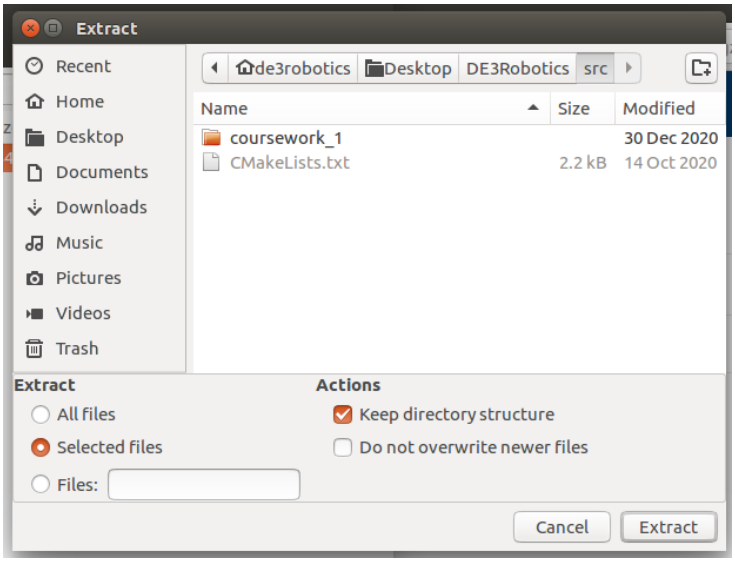
Once you have downloaded the zip file, open it with Archive Manager (the default program):



After the Archive Manager opens the zip file, press Extract:



You will now have to select a directory to extract the files to. Navigate to `Desktop/DE3Robotics/src`



Press Extract. When this has completed you will have all the files you need for coursework 2 in the correct folder! Now, we need to prepare these files to work with ROS. Open a new terminal, and navigate to the DE3Robotics folder:

```
cd Desktop/DE3Robotics
```

Now, run the following commands to update permissions of the workspace:

```
find -type f -iname "*.cfg" -exec chmod +x {} \;
find -type f -iname "*.py" -exec chmod +x {} \;
```

Now, run the following command to 'build' the workspace:

```
catkin_make
```

This will take a few seconds - if everything has completed successfully, the progress counter will reach 100%.

Finally, we will install `pandas` - a Python library for data handling. To do this, run the following command from terminal:

```
pip install pandas
```

Everything is now ready to run!

In a new terminal, run the following command: `roscore`

Open another terminal, and run the following commands:

```
cd Desktop/DE3Robotics
source devel/setup.bash
```

Then run the following command to load a Gazebo world populated with obstacles for DE NIRO to navigate:

```
roslaunch deniro_gazebo deniro_world.launch
```

Note: it could take a while for this to complete - up to a couple of minutes. It is important to be patient while this loads; running other commands (such as spawning DE NIRO) will cause issues!

Hint: Using autocomplete in terminal

You can press the Tab key to autocomplete a command you are writing in the terminal. For example: `cd Des` + Tab autocompletes to `cd Desktop`.

You should see a world with various obstacles in Gazebo:

Now, we'll add Robot DE NIRO to this simulation. Open a new terminal and run the following commands:

```
cd Desktop/DE3Robotics
source devel/setup.bash
```

Then run the following command to add Robot DE NIRO to the simulation:

```
roslaunch deniro_gazebo deniro_spawn.launch
```

You should see DE NIRO added to the simulation:

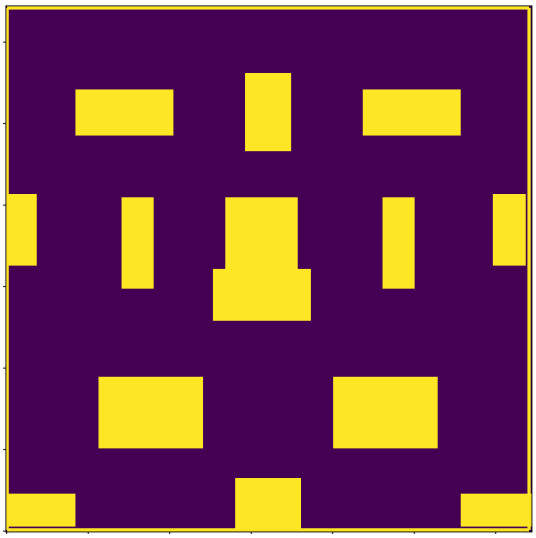
You can use the mouse scroll wheel to zoom in. To change the viewing angle, you can use the left, right, and middle/wheel buttons of the mouse to rotate and translate as needed.

Hint: Tucking DE NIRO's arms

When DE NIRO spawns, sometimes the function to tuck the arms out of the way might fail. If this is the case, run the following command:
`rosrun baxter_tools tuck_arms.py -t`

To load a map of the environment, run the following commands:
`cd Desktop/DE3Robotics/src/coursework_2/src`
`python3 map.py view`

The map should look like this:



Where yellow rectangles are obstacles to avoid. The map is scaled such that 1 m is 16 px wide. This means that a single pixel width is 0.0625 m. The centre of the world, coordinates (0, 0) m are located at the centre of the map.

Hint: Resetting DE NIRO's position

Instead of closing and relaunching Gazebo and DE NIRO each time you want to test your code, it is easier to run the following commands to reset DE NIRO's position. First, navigate to the coursework 2 source folder:

```
cd Desktop/DE3Robotics/src/coursework_2/src
```

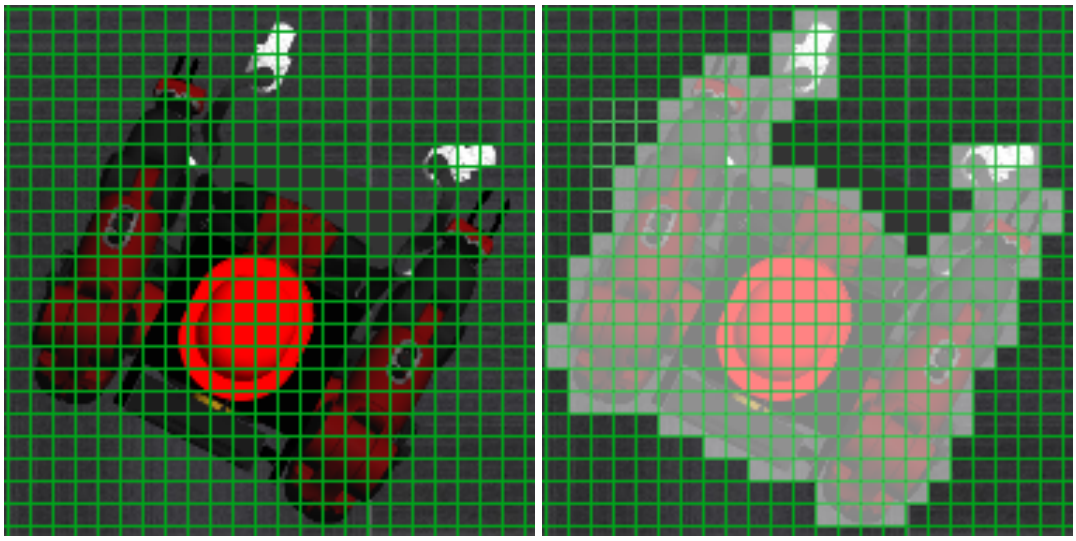
Then run the following Python script to reset DE NIRO to $(0, -6)$ m:

```
python3 set_deniro_pose.py
```

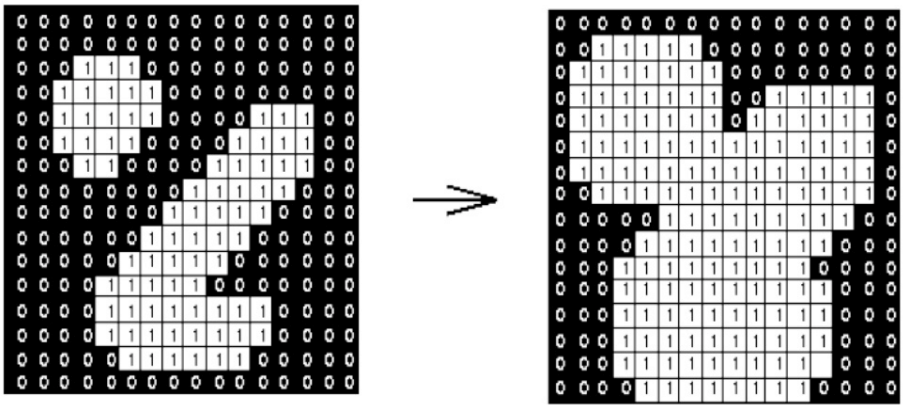
3 C-Space Map

Before we go any further, we need to produce a map that accounts for DE NIRO's size. This is the C-space map that we will use when planning motion.

To generate the C-space map, we will say that Robot DE NIRO is 1.0 m wide. For each pixel in the map, we will have to check if DE NIRO will hit any obstacles, and produce an updated map accordingly. To do this we will create a pixel 'mask' of DE NIRO:



The process of expanding the map based on a mask of the robot falls into the category of an image processing technique called 'dilation':



Effect of dilation using a 3×3 square structuring element

Source: https://www.javatpoint.com/opencv-erosion-and-dilation

Task A: C-Space dilation

Part i
Locate `Task A` in `map.py`. Fill in the code to generate a **square** mask to represent DE NIRO. Include the expanded map in your report - how has it changed compared to the original map?

You can view your generated mask by running the following command from Terminal:
`python3 map.py expand`

[Only for groups]

Part ii
Write some code to generate a **circular** mask to represent DE NIRO. Include this expanded map in your report - how does it compare to the map generated from a square mask? What advantages does a circular representation of DE NIRO have over a square representation?

Hint: Displaying images with `matplotlib`

You may wish to view your robot mask, and maybe even include it in your report. To view a visual representation of a two-dimensional array, use the `plt.imshow` function by adding the following lines to your Python code:

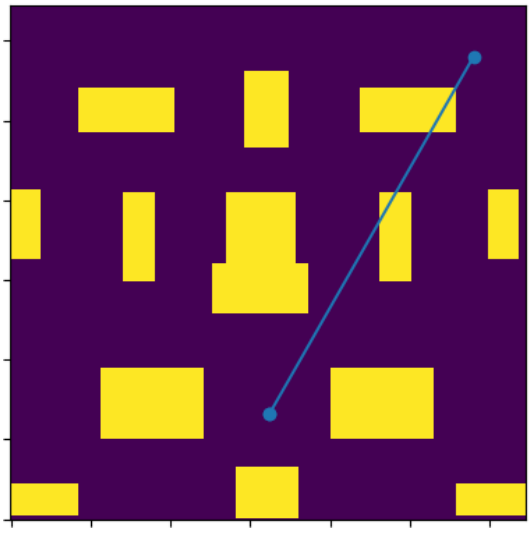
```
plt.imshow(your_array, vmin=0, vmax=1, origin='lower')
plt.show()
```

If you don't like the quite jarring yellow-on-purple map, you can also look at changing the colour map, by specifying `cmap`. See more about the colour maps available here:
https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html

4 Waypoint Navigation

A simple method of robot motion planning is to add waypoints by hand that the robot must travel to on the way to its goal. For all navigation tasks, we would like DE NIRO to navigate from a starting position of $(0.0, -6.0)$ m to $(8.0, 8.0)$ m.

You will notice that, in this case, we can't just walk directly to the goal:



We will have to perform some kind of motion planning to reach the goal!

Task B: Adding waypoints by hand

Part i
In `motion_planning.py`, find `TASK B`. Based on the expanded map you produced in Task A, calculate suitable waypoints for DE NIRO to pass through to reach the goal without hitting any obstacles. Add your waypoints to the code **in world coordinates** (not the pixel coordinates shown on your map).

With DE NIRO spawned in Gazebo (see Getting Started), open a new terminal window and run the following command to navigate to the correct folder:
`cd Desktop/DE3Robotics/src/coursework_2/src`
Then run the following command for DE NIRO to navigate to his goal via the waypoints you have added:
`python3 motion_planning.py waypoints`
This will also show you DE NIRO's planned route on the C-space map - include this in your report.

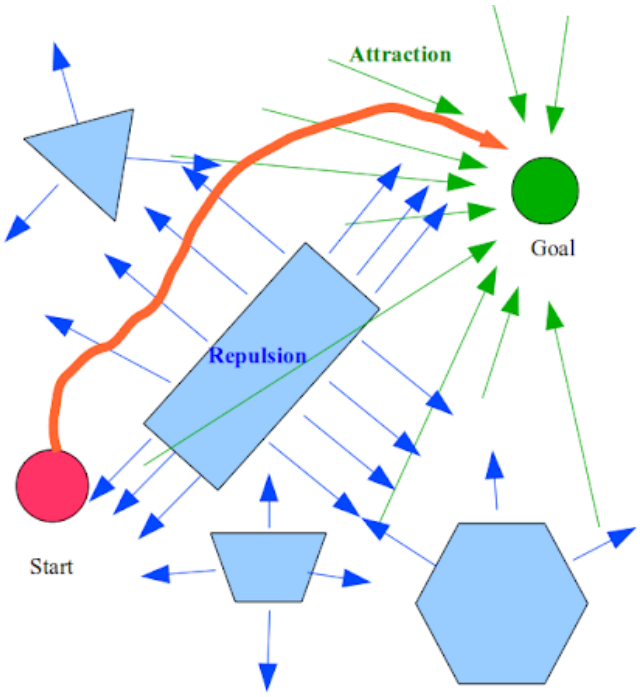
Explain how you chose your waypoints. If you calculated your waypoints in pixel coordinates then converted them to world coordinates, explain how you achieved this.

[Only for groups]

Part ii
Calculate the total path length of your planned path - try to find the shortest path from DE NIRO's initial position to the goal. What was your strategy for finding the shortest path?

5 Potential Field Algorithm

The potential field algorithm is a reactive method of path planning that represents obstacles as negative potentials, and the goal as a positive potential.



There are many strategies to assigning potentials, but they are all based on the robot's distance to the objects and the goal. A simple strategy for calculating the attractive force on the robot is to make it inversely proportional to the robot's distance from the goal:

$$\boldsymbol{f_{att}} = K_{att}\hat{\boldsymbol{x}}_g/d_g, \tag{1}$$

where $\hat{\boldsymbol{x}}_g$ is the unit vector from the robot to the goal, d_g is the distance of the robot from the goal, and K_{att} is a parameter to be tuned.

Similarly, this can be used for the repulsive force from each obstacle:

$$\boldsymbol{f_{rep}} = -K_{rep}\frac{1}{N}\sum_{i=1}^{N}\hat{\boldsymbol{x}}_i/d_i, \tag{2}$$

where $\hat{\boldsymbol{x}}_i$ is the unit vector from the robot to obstacle i, $d_i = |bmx_i|$ is the distance from the robot to obstacle i, and K_{rep} is a parameter to be tuned.

However, these strategies are problematic when there are many obstacles, or the goal is far away. Sometimes we must develop more complicated potentials to achieve our desired behaviour.

When the goal is very far away, it is sometimes beneficial to have an attractive forward towards the goal that has a constant magnitude:

$$\boldsymbol{f_{att}} = K_{att}\hat{\boldsymbol{x}}_g. \tag{3}$$

In general, all potential forces can be formulated in the following way:

$$\boldsymbol{f} = K\hat{\boldsymbol{x}} \cdot p(\boldsymbol{x}) \tag{4}$$

Where $p(\boldsymbol{x})$ is a function that determines the magnitude of the force. In the case of equation (1), $p(\boldsymbol{x}) = 1/|\boldsymbol{x}|$.

Task C: Implementing the potential field algorithm

Part i

Locate `Task C` in `motion_planning.py`. Start by writing an expression for the positive force (the force from the goal) and the repulsive force (the force from the obstacles) using a repulsive potential that is inversely proportional to distance, as in equation (2), and an attractive potential that is constant, as in equation (3).

With DE NIRO spawned in Gazebo (see Getting Started), open a new terminal window and run the following command to navigate to the correct folder:
`cd Desktop/DE3Robotics/src/coursework_2/src`
Then run the following command for DE NIRO to navigate to his goal using the potential field algorithm you have implemented:
`python3 motion_planning.py potential`

The parameters `K_att` and `K_rep` will require a lot of tuning for DE NIRO to successfully reach the goal!

WARNING! It is very difficult to achieve perfect tuning of the parameters and make the potential field algorithm work successfully. In fact, this is part of the learning outcome for this task. Therefore, we do NOT expect you to spend much time tuning. Instead, it is sufficient to just experiment with a few different values and explain how the behaviour changes. It is okay even if DE NIRO does not reach the goal, just explain why.

On an image of the map, draw the path that DE NIRO takes, and include this in your report. Is this the optimal path? Comment on any interesting features of the path taken.

What are the advantages and disadvantages of a reactive motion planner, such as the potential field algorithm, over other motion planners, such as the probabilistic road map?

[Only for groups]

Part ii

As mentioned, sometimes we require more complex potential fields to achieve better performance from our motion planner. Design and implement your own potential functions (different from equations (1-3). Discuss your reasoning behind your proposed potential function, and how you implemented it in your report.

On an image of the map, draw the path that DE NIRO takes, and include this in your report. Is this the optimal path? Comment on any interesting features of the path taken.

Estimate the length of the path taken by DE NIRO - is it better or worse than the length of the path taken when following your own waypoints? If it is better, why do you think it is better? If it is worse, why do you think it is worse?

Hint: Viewing your potential field

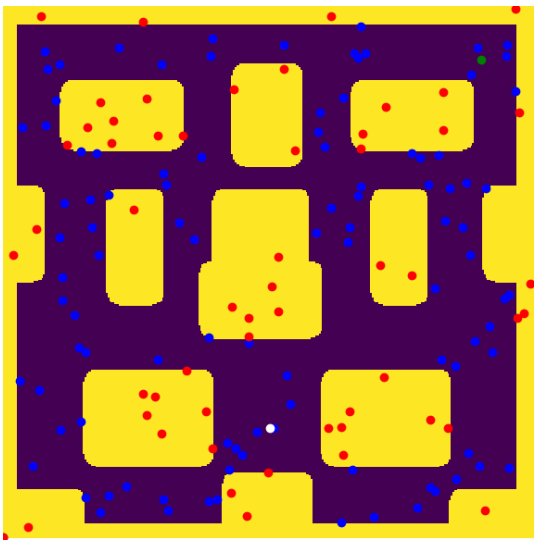
You may find it useful to view your potential field. You can do so by adding some lines of code we have prepared in `motion_planning.py`:

```
plotskip = 10   # only plots every 10 pixels (looks cleaner on the plot)
plt.imshow(self.pixel_map, vmin=0, vmax=1, origin='lower')

plt.quiver(obstacle_pixel_coordinates[::plotskip, 0],
obstacle_pixel_coordinates[::plotskip, 1],
obstacle_force[::plotskip, 0] * self.xscale,
obstacle_force[::plotskip, 1] * self.yscale)

plt.show()
```

6 Probabilistic Road Map

6.1 Random Point Generation

Probabilistic road map is an offline motion planning algorithm. It begins by randomly sampling points on a map of your environment:



Task D: Randomly sampling from the map

Part i

Locate `Task D` in `motion_planning.py`. We would like to generate N_{points} samples that don't collide with obstacles.

Complete the code to loop through the generated points and check if they are inside any obstacles. Remember, the value of an obstacle on the map is 1! If a generated point is inside an obstacle, set the corresponding entry in the rejection flag array to 1.

Spawn DE NIRO in Gazebo as before. When you run:

```
python3 motion_planning.py prm
```

A plot of the accepted and rejected points will be displayed on the map. Include this in your report.

[Only for groups]

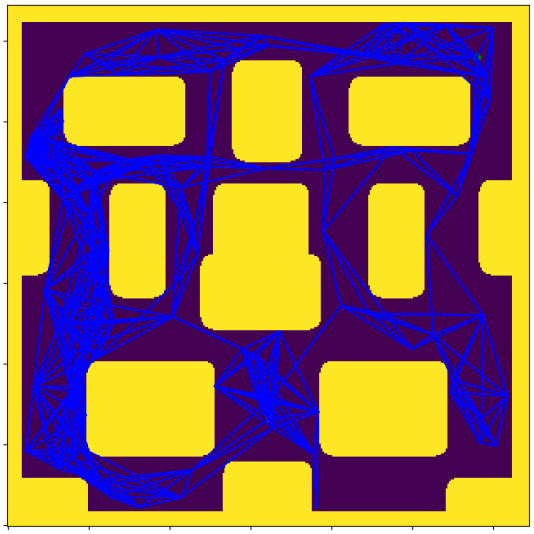
Part ii

If you inspect the code for generating samples, you will see that uniform sampling has been used. This might not be the best strategy to deal with regions of high obstacle density.

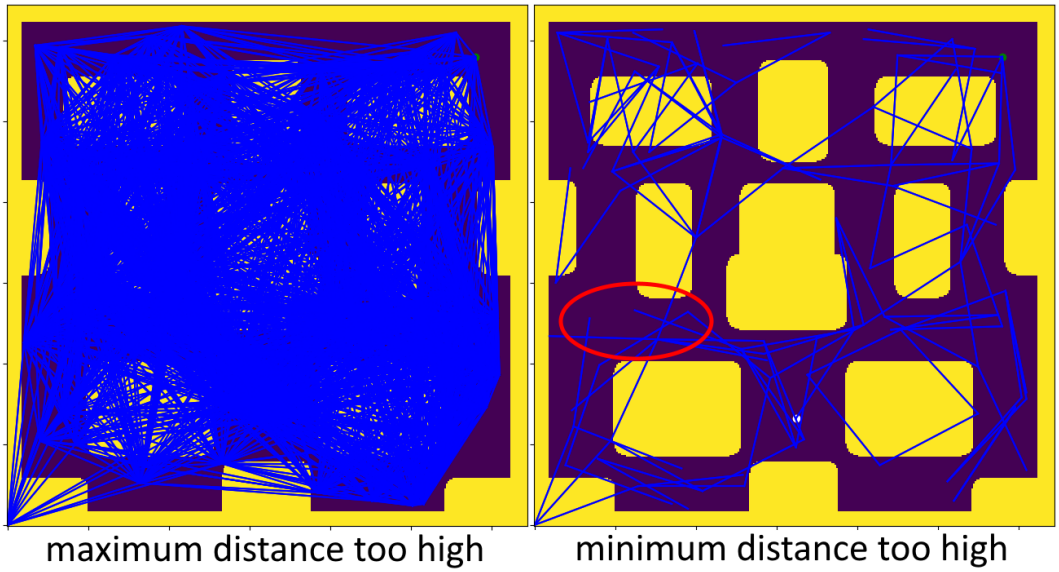
Propose your own strategy to deal with these problems (there is no need to implement it), and discuss this in your report.

6.2 Graph Creation

After sampling random points, we must generate a graph that uses these points as 'nodes':

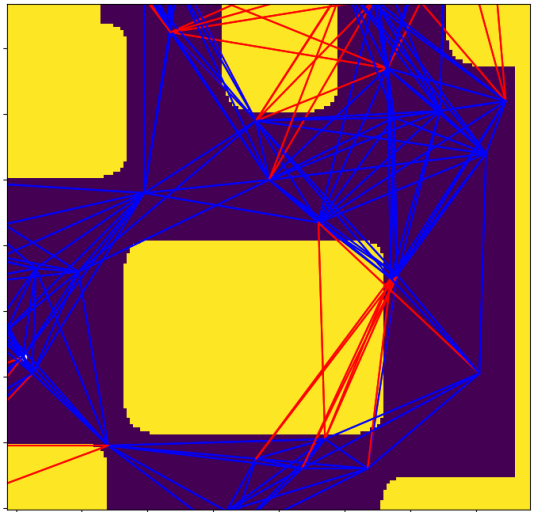


To achieve this, we must choose a minimum and maximum distance for two nodes to be given an edge. If two points are very far apart, it is likely that an obstacle will be between them. If two nodes are very close, there isn't any point in travelling between them.



Left - maximum distance too high, every node is connected unnecessarily. Right - minimum distance too high, nodes are disconnected unnecessarily (highlighted in red circle).

Even after choosing a minimum and maximum distance for edges, there's still a chance that the edge could collide with an obstacle. To avoid this, each edge must be checked for obstacles at some resolution - if the resolution is too low, edges could be kept even though the robot will collide with an object:



Task E: Creating the graph

Part i
Locate `Task E i` in `motion_planning.py`. Tune the minimum and maximum distance of the edges to produce a suitable graph. Discuss your reasoning behind your selected values and include the generated graph in your report.

Part ii
Locate `Task E ii` in `motion_planning.py`. Each edge that has been generated must be checked for collisions. Fill in the code to calculate the direction vector and distance between two nodes (`pointA` and `pointB`).

Tune the resolution of collision checking to make sure no accepted edges are in collision with an obstacles. Include your chosen resolution in your report, as well as the resulting graph (edges colliding with obstacles are red).

[Only for groups]

Part iii
A problem occurs when there aren't enough edges to make a complete graph from the initial node to the goal node. Propose a method to search near edges for suitable locations for new nodes, and discuss this in your report (you do not need to implement this).

6.3 Dijkstra's Algorithm

Once we have a suitable graph, we would like to find the optimal route to go from DE NIRO's initial position, to the goal position. They can be achieved using Dijkstra's algorithm.

We can used Dijkstra's algorithm to find the shortest path from the initial position to the goal position as follows:

1. Create two dataframes with headings 'Node', 'Cost', and 'Previous'. The 'Node' column tells us which node we are looking at, 'Cost' tells us the total distance travelled to reach this node, and 'Previous' tells us the path taken to reach that node.

   The first dataframe is the 'unvisited' dataframe, which initially contains every node in the graph. The cost of every node is initially set to a very high number, except the initial node, which has a cost of 0. This initial dataframe is shown in Table 1.

Node	Cost	Previous
$[x_{init}, y_{init}]$	0.0	''
$[x_1, y_1]$	1e6	''
$[x_2, y_2]$	1e6	''
...	...	...
$[x_N, y_N]$	1e6	''

Table 1: The initial unvisited dataframe

   The second dataframe is the 'visited' dataframe, which is initially empty, as shown in Table 2

2. Set the node with the lowest unvisited cost to be the current node.

Node	Cost	Previous
‘’	0.0	‘’

Table 2: The initial visited dataframe

3. Loop through every node connected to the current node that we haven't visited yet. For each connected node, check whether the cost of travelling from the current node to the connected node is smaller than the cost already recorded for the connected node. If it is, update the connected node's cost and its previous path to be the previous path of the current node + the current node.
4. Remove the current node from the unvisited dataframe, and place it in the visited dataframe.
5. Repeat steps 2-4 until the goal node is in the visited list.

When the algorithm is complete, the goal node row in the visited dataframe corresponds to the optimal distance from the initial position to the goal position, and the points DE NIRO will have to visit to get there. These points can then be used for waypoint navigation.

Task F: Dijkstra's algorithm

Part i
Locate `Task F` in `motion_planning.py`. You will see a Python implementation of Dijkstra's algorithm. Choose a suitable value for the initial cost of each node.

Complete the line that calculates `next_cost_trial` to calculate the cost of going from the initial node to the next node via the current node.

Once these are completed, you will have completed every component of the probabilistic road map motion planning algorithm! A map will appear with the proposed optimal route overlayed on it - include this in your report. Comment on any interesting features of the proposed route - how does it compare to the waypoint route you proposed in Task B?

DE NIRO will then navigate via these waypoints to the goal - comment on anything interesting you see.

Dijkstra's algorithm produces the optimal route **given a predefined graph**. Is the path taken actually optimal? If yes, justify your answer. If no, what would be needed to generate an optimal route?

[Only for groups]

Part ii
Dijkstra's algorithm is known as a one-to-many planning algorithm. This means it finds the optimal route from a starting node to all other nodes in a graph (if run to completion).

What are the advantages of a one-to-many planning algorithm? What would you need to do to travel from the first goal to a new goal node?

Find an algorithm that is more suitable for one-to-one planning, and explain it in your report. There is no need to implement this algorithm.