How to Run:

Run the following command to launch Gazebo.

```
$ roslaunch turtlebot3_gazebo turtlebot3_stage_4.launch
```

Run the following command to launch RVIZ.

```
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

This should open a configuration file containing two path markers as well as a camera. If it doesn't, the configuration can be loaded by selecting file->open config and choosing project_two_config.rviz.

Run the launch file using the following command.

```
$ roslaunch ros_course_project_two turtlebot3_navigation.launch
```

Part One

The following command will launch a node responsible for providing information about the robot's location on the map.

```
$ rosrun ros_course_project_two map_tf2_listener.py
```

Finally, you can send a command to the robot by starting the action client as shown below and telling the robot how far to move in the x and y directions.

```
$ rosrun ros_course_project_two moveBaseClient.py
```

Part Two

Considerations

In order to find the frontiers and grow obstacles, an occupancy grid was retrieved from the map topic using a subscriber. This returned a dictionary containing a header, an info section, and the actual data. It had the following form (note I didn't include all values):

```
{
    header: {seq, stamp, frame_id}
    info: {map_load_time, resolution: ~0.05, width: 384, height: 384, origin}
    data: [-1, -1, -1, 0, 100, ...]
}
```

Using this information, I converted the occupancy data (found using occupancy_grid.data) to a 2D grid. This essentially becomes a 2D list: [[row], [row], [row], . . .]. To do this, I used a numpy method called reshape. The original data is in row-major order (according to the documentation), which is the default option for reshape. In order to transform coordinates from the grid representation to the robot's physical frame, I used the map's resolution. The width and height

of the occupancy grid is 384 by 384, respectively. So, the width and height of the physical space can be found as follows:

```
width = height ~ 384*0.05 = 19.2
```

Using this information, the x and y coordinates in the physical world can be found from the occupancy grid using the following formula:

```
x = 19.2*(x_grid/384) - 10
y = 19.2*(y_grid/384) - 10
```

Note that we must subtract 10 because the origin of the occupancy grid is not the same as the origin of the map. It is located at (-10, -10) in the physical space. Taking the derivation of the physical width, these formulas can be rewritten as:

```
x = 0.05*x_grid - 10
y = 0.05*y_grid - 10
```

I decided to use a clustering technique called DBSCAN, or Density Based Spatial Clustering of Applications with Noise, because it removes outliers as a fundamental part of the algorithm. I thought this would be helpful in centralizing clusters around a location and removing unnecessary single points that could normally be classified as additional clusters. Instead, DBSCAN creates a single "noise" cluster. I decided to remove this additional cluster in order to ensure that the robot doesn't try to seek out the noise in the future. This algorithm works by finding clusters with a high density. It finds samples that have a higher density and uses those samples to search the surrounding area. Clusters are formed using a maximum distance value such, epsilon, that any point within the designated distance from a core sample is considered part of that core sample's cluster. Another reason I thought that this algorithm would be useful for clustering frontiers is because it is generalizable to any shape.

Using the DBSCAN() method from SciKit Learn, I created a pandas dataframe object containing the x,y location of each frontier point and the cluster each point belonged to. Using this, I could easily find individual clusters to create markers with different colors. I used a MarkerArray object in which every Marker was a series of points corresponding to a distinct cluster. To visualize this easily on the RVIZ map, I used a library called Seaborn to give each point in a given marker a color. This allowed me to create different colors for each marker in accordance with my desired color palette. I used a color palette from Seaborn, rainbow, that could contain any number of colors so that I didn't have to define a set number of clusters.

I also used a unique technique in order to grow the obstacles based on the initial occupancy grid. I used recursion to move along the x axis and y axis from a point containing an obstacle (represented as 100 in the occupancy grid). A limit can be specified to indicate how much the walls should be grown. I chose to use a value of 3 because I wanted the boundaries to be about half the width of the robot away from the original wall. I calculated this as follows. The physical dimensions of the waffle pi turtlebot are 0.281 m by 0.306 m. The value of this

based on the occupancy grid can be found by dividing by the resolution. The gives:

```
width ~ (0.306/0.05)/2 = 3.06 (divided by 2 for half the robot's width)
height ~ (0.281/0.05)/2 = 2.81
```

Since I'm using this value as a limit for an integer based value, I decided to simplify things by rounding both the width and height to 3. This essentially restricts the depth of the recursion so that obstacles are only grown to 3 spaces on all sides. I also decided to display this expansion in rviz using markers in order to ensure that it was working properly.

To Run

You can either run just this part, along with the three commands above, or it can be used jointly with part one to control the turtlebot via the terminal. To find the frontiers and grow the obstacles, run the following command.

```
$ rosrun ros_course_project_two frontier_detection.py
```

This, in combination with the saved RVIZ configuration, should bring up an overlaying map of all of the detected frontiers as well as markers indicating the way in which the obstacles were expanded.

Next, to segment the frontiers and locate the centroids, run the command below.

```
$ rosrun ros_course_project_two frontier_segmentation.py
```

This should show differently colored markers along with a centroid marker for each cluster.

#Part Three: ## How to Run:

```
rosrun ros_course_project_two frontier_evaluation.py
```

Considerations:

I decided to use a spin on a RRT inspired technique by selecting a random point around the initial inhabitable coordinate and resampling if the point is also an obstacle. This allowed me to remove the likelihood of the robot running into an obstacle it can't reach. Further, I used the action client's get_state method to determine when a goal is aborted so that I can give the robot a new location. I also used a main method to remove much of the functionality from callback so that updates could be made more seemlessly.

Part One Demo

https://youtu.be/sAWmcl47-TM

Part Two Demo

https://youtu.be/uwIDLPSOuNI

Part Three Demo

I'm really sorry, I had to fix an error at the last minute. It is working, but I don't have time to wait for the video to load before submitting. I will add a comment with the youtube link. I apologize for any inconvenience!