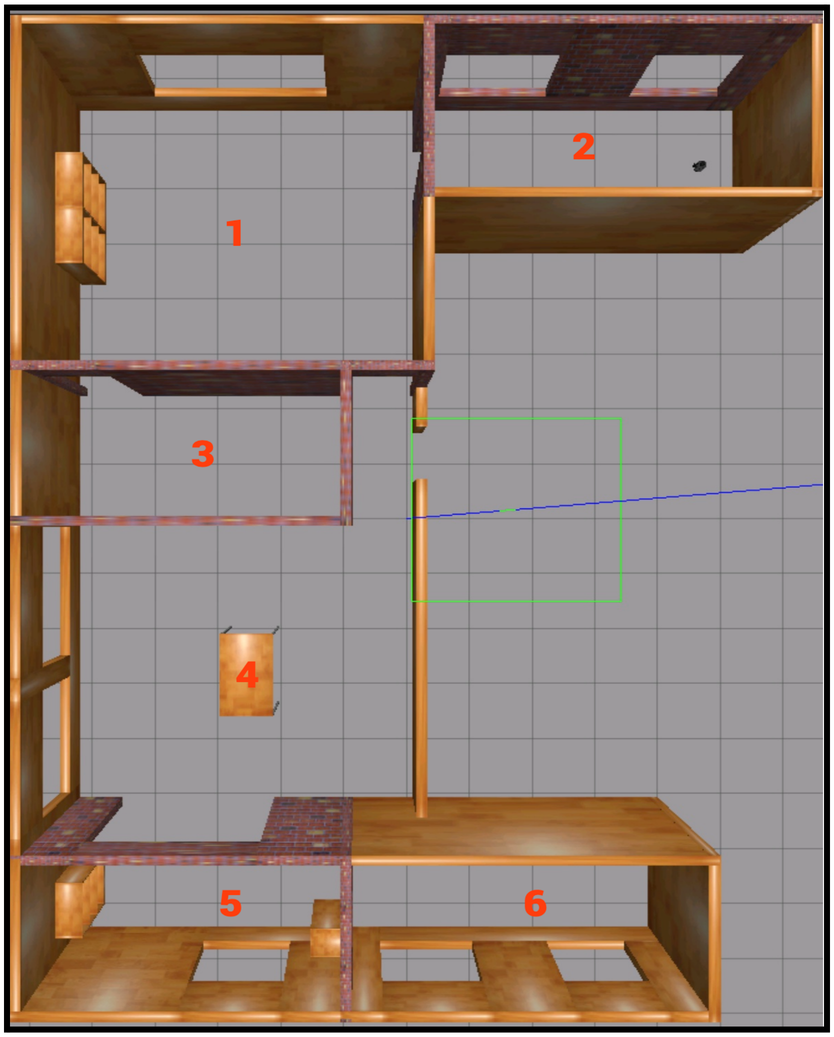
Alexy Skoutnev and Mary Stirling Brown

Final Project Report

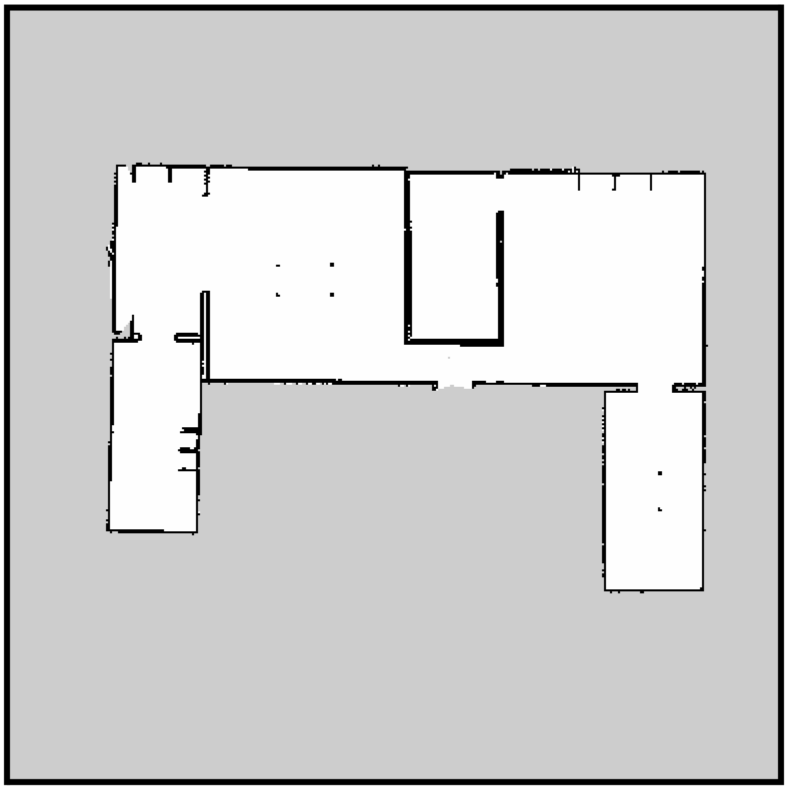
We implemented task 2 for the final project. Instructions for running our code can be found on our GitHub repo here. We first drove the turtle bot around in the house environment to create a map. The rooms of the house were labeled as follows:



SLAM Map Generation:

The navigation stack used in this project required a pregenerated map, and this was obtained from a ROS package named slam_toolbox. The slam_toolbox package uses a Simultaneous

Localization and Mapping (SLAM) algorithm to generate a map of the environment using sensor data from the robot. The robot was able to scan the environment utilizing laser reading and odometer measurements. We then loaded this map into the navigation stack and used it in RRT algorithm we developed. By using a pre generated map, we were able to significantly reduce the computation required for real-time navigation. The SLAM map of the environment is seen below.



RRT Planner Implementation:

The RRT planner plugin was implemented by editing the source code from an RRT* global planner on GitHub. This source code had the plugin already configured correctly, but we still had to change the turtlebot3's move_base.launch to include the parameters for the RRT plugin to be the "base_global_planner."

From the source code, we deleted the RRT* specific functions that assigned a node's parent based on a cost metric in rrt_star.cpp. In addition, we edited the majority of the code here to implement a simple RRT planner, such as adding a goal bias of 10%. The node struct was edited to assign each node a unique ID integer and their respective parent nodes' ID as well. The tree was built by connecting children nodes to their parent nodes based on the ID's. A new node's parent was assigned based on what node in the tree had the shortest distance between them. The distance metric used to compute the distance between two nodes was the Euclidean distance because the turtlebot3 only moves in the x and y positions within the house environment. With all of these changes to the source code, our turtlebot3 can successfully create a global plan from its initial position to the global position.

The main issue with creating the RRT planner dealt with the obstacle detector. The map of the house environment came as ROS's costmap_2d interpretation. Each cell in the cost map ranges from the value of 0 to 255. A freespace cost is 0 whereas an obstacle is in the higher range of about 254. The lower the value, the higher the desirability for the robot to go there. The higher the value, the lower the desirability for the robot to want to go there. So, narrower paths, such as the paths between 1 and 2 were given a higher cost. Therefore, we had to increase the cost tolerance to identify an obstacle to be greater than approximately 230. By increasing the cost to identify an obstacle, it may not recognize all obstacles or maneuver spaces around obstacles that are at a value of less than 230. The value to identify an obstacle greater than 230 seemed to be the most optimal solution after many trial-and-error with other cost values. Too low of a value would not allow the robot to go through narrow passages. Too high of a value would misclassify many of the walls and furniture as free spaces. Our RRT planner may not work perfectly all the time as there was a tradeoff in creating nodes in narrow paths and identifying the space around an obstacle that could lead to a collision.

U/I Interface:

The user interface, navigation_interface.cpp, was built on top of the RRT planner to navigate through the house environment. Many initial conditions such as, initialpose estimate and robot model state, had to be met for the base_move planner to work properly. The preprocessing step was done by utilizing ROS launch files and having processes evoked in a proper sequence. Within the proprocess stage, the file init_pose.cpp was executed to give the navigation stack a good initial pose estimation. Then the launch files that started the navigation planner was move_base.launch while navigation.launch configured all the initial states need for move_base and opened up a command line UI for giving navigation commands. A series of subscriber and publisher node were utilized to obtain the init state configuration and allowed the UI to pipe model states down to the planner.