3630 Intro to Robotics and Perception
Release Date: November 15th, 2024
Due Date: November 26th, 2024

Project 6: Warehouse Automation

Overview

In this lab, your robot will supervise a warehouse, mapping its environment and avoiding obstacles. If it hits an obstacle, the run ends. The robot knows its exact location on the map, so localization is not a concern. The goal is to visit all five landmarks within 6 minutes (360 seconds).

All the important files are present under the folder *controllers/exploration_controller/*. Specifically, focus on *controllers/exploration_controller/exploration.py* file.

Detailed descriptions of the code implementation

Your task is to program a robot to explore an unknown warehouse, collect markers, and map the area using its lidar sensor. The robot starts with no knowledge of the warehouse layout and can only detect its surroundings through lidar scans.

Here is what you will be implementing:

1. **Frontier Planning:** Identify the boundary (frontiers) between explored and unexplored areas. Choose a frontier to explore.
2. **Connected Component Function:** Group and manage frontiers for efficient selection.
3. **Exploration State Machine:** Calls the frontier planning function to select a frontier and then uses Rapidly-exploring Random Tree (RRT) to plan a path toward that frontier.
4. **PID Controller:** Calculate the robot's linear and angular velocities to reach a selected waypoint.

The robot explores by selecting a frontier, planning a path, moving to the first waypoint, and then updating its frontiers. This process repeats until all markers are collected.

Frontier Planning

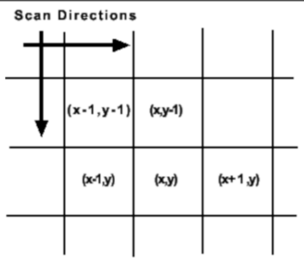
You may find these helper functions useful for implementing *frontier_planning():*
Grid.is_free(), Grid.is_in()
Robot_Sim.explored_cells
get_neighbors() [exploration.py]
find_centroid() [utils.py]
grid_distance() [utils.py]

Algorithm 1 Frontier Planning

```
1:  function FRONTIER_PLANNING(robbie, grid)
2:      Build a list of frontier cells [1]
3:      Group frontier cells (Using connected components labeling)
4:      Calculate centroids of each frontier group
5:      Sort centroids by distance from robot and frontier size
6:      Select best centroid based on distance and accessibility [2]
7:      if valid centroid found then
8:          Set robot's next destination to the selected centroid
9:      else
10:         Set robot's next destination to a random frontier cell
11:     end if
12:     return updated robot and next destination
13: end function
```

1. free, explored cells adjacent to unexplored ones.
2. util = $\alpha_1 \cdot$ distance $+ \alpha_2 \cdot$ frontier_length

Connected Component



Algorithm 6 Connected Component Labelling

```
Input: Labelling a particular pixel $(x, y)$
1:  if the pixel $(x, y)$ has '0' then
2:      Do nothing and proceed to next pixel $(x + 1, y)$
3:  else if the pixel $(x - 1, y - 1)$ has a label then
4:      Assign the label to the pixel $(x, y)$.
5:  else if neither pixels $(x - 1, y)$ or $(x, y - 1)$ is not labelled then
6:      Increment label numbering and assign the latest label to pixel
        $(x, y)$
7:  else if pixels $(x - 1, y)$ XOR $(x, y - 1)$ is labelled then
8:      Assign the label to the pixel $(x, y)$
9:  else if both pixels $(x - 1, y)$ and $(x, y - 1)$ are labelled then
10:     Assign the label of pixel $(x - 1, y)$ to the pixel $(x, y)$
11:     Record the equivalence if labels of pixels $(x-1, y)$ and $(x, y-1)$
        are not identical.
12: end if
```

Merges labels in single pass

You can test this function by running *test_connected_components.py*.

Exploration State Machine

In the exploration_state_machine function, you are expected to implement the robot's exploration logic. This involves sensing free cells in its field of view, planning a next target coordinate using frontier planning, handling collisions with obstacles by using RRT to replan paths, and setting the robot's wheel velocities to move toward the target.

Note: Using 'grid.markers' will lead to zero points on GraderScope.

As Project 6 will be released prior to the final due date of Project 5, and in Project 5 you are implementing RRT, you will copy your implementation of RRT to use when navigating to your computed frontiers (*grid.py*). However, you will not be submitting this file – it will be only to test locally. To ensure that all students can still get started on Project 6, we provide the RRT implementation on Gradescope for grading. Once Project 5 is **closed for everyone, taking late submissions into account**, we will provide you with this RRT implementation for local testing as well.

PID Controller

In the exploration_state_machine() function, you are expected to define a proportional-integral-derivative (PID) controller that adjusts the robot's linear and angular velocity. You are provided with error functions for both linear (compute_linear_error) and angular errors (compute_angular_error), which you will use to implement the PID logic. Increase Kp if the robot is too slow to respond. Add Ki if there is a persistent steady-state error. Use Kd to reduce overshooting or oscillations.

Hint: Use the helper functions in utils.py for calculations, such as grid_distance and rotate_point.

Local Testing

We provide 3 maps for your local testing under *controllers/exploration_controller/maps* that can be used for local testing. You can visualize the code locally, by running *exploration_controller/robot_gui.py* with the desired map specified. While there are 5 markers in each map, this file will list the total number of markers as greater than 5 due to the discretization of the world, as you can observe via the GUI. Note that each map is still worth a total of 20 points.
We also provide *autograder.py* file that you can run for local grading. Example usage of *autograder.py*: "*python3 autograder.py maps/maze1.json*".

Make sure to take advantage of functions in utils.py and remember that your run for either phase will terminate if you collide with an obstacle.

Grading

Rubric	
Found all five markers in the map	20 pts/map

Autograder will grade your exploration on 5 maps.

Submission Materials

Submit your exploration.py file and honor_code.docx to gradescope.

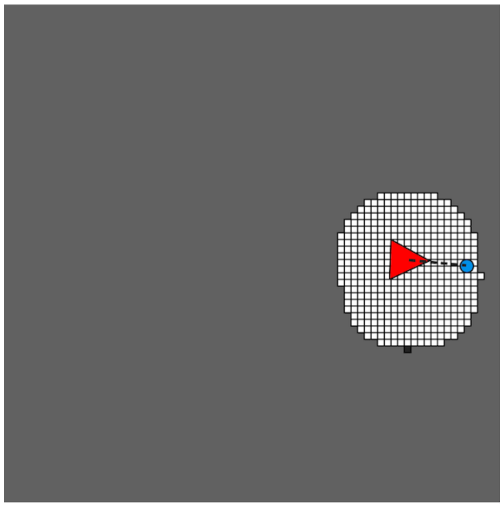
Running with Webots

Once the exploration.py is complete, you can validate that your algorithm works with the WeBots simulator.

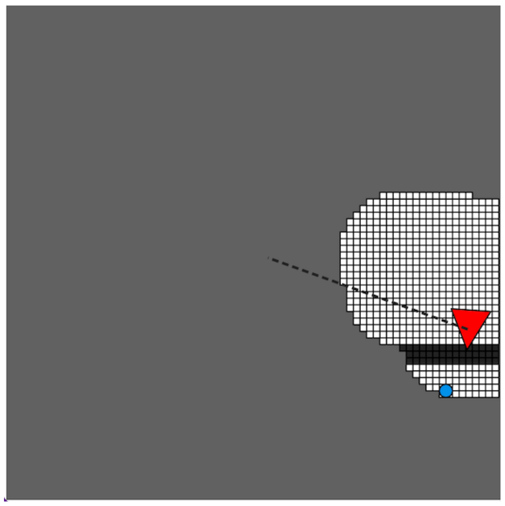
1. Open exploration_controller.py and **ensure that the maze_name (line: 255) is set to the map you would like to test on**. The map names are “maze1”, “maze2”, and “maze3”. A working implementation should work on all three maps.
2. Launch WeBots. Click File > Open World, then navigate to the “worlds” subfolder. Open the `.wbt` file which matches the **maze_name** from step 1.
3. A GUI window should appear which shows your exploration code running in a simple 2D approximation of the world to find a path. Then, the robot in WeBots should start moving along this path.
    a. We recommend selecting the “DEF e-puck Robot” object via the sidebar in the top-left of the WeBots screen. This will allow you to see the coordinate frame of the robot while it moves through the maze.
    b. Note that the 2D GUI window should automatically close after the robot is done moving.
    c. Also, the webots execution takes significantly longer time to run compared to GUI and autograder version.

Examples (Robot-GUI)

Robots should navigate towards the frontier



Robot can use RRT to get around the obstacle



Blue dot signifies the goal point
Dotted line signifies the path to the next waypoint in the RRT path