Lab 4: Mapping with LIDAR

CSCI 3302: Introduction to Robotics

Due 11/3/20 @ 11:59pm

The goals of this lab are to:

- Implement a basic discrete map representation
- Use coordinate transforms to map sensor readings into world coordinates
- Understand discrete and algorithmic aspects of mapping

You need:

- A working Webots Installation
- The Lab 4 base code from Canvas

Overview

Mapping and navigation are standard primitives in autonomous robots. Although the different map representations and planning algorithms vary drastically in complexity, the problems remain the same: localization and sensing is uncertain, maps are quickly corrupted, and both mapping and navigation have to deal with limited on-board resources. In this exercise you will use ePuck's new LIDAR sensor to map objects in the environment. You will initially display the objects' coordinates in the console and then implement data structures and helper functions that (1) allow you to store a map in ePuck's memory and (2) are suitable for path planning.

Instructions

Each group must develop their own software implementation, turned in by one team member with the lab report. Thus, each team should turn in a zip file including:

- The group's code (1 .py file)
- A single lab report (1 .pdf file)
- The Webots environment folder containing your world and code file.

If your group does not finish the implementation by the end of the class, you may continue this lab on your own time as a complete group.

Part 1: Preliminaries for using LIDAR

1. Initialize an array/list data structure to hold your LIDAR's incoming readings.
   You will be accessing the LIDAR's sensor readings with lidar.getRangeImage(). This will return an array containing a distance measurement from each angular 'bin'.

2. Before your controller's while loop, using the provided constants LIDAR_ANGLE_RANGE and LIDAR_ANGLE_BINS, programmatically compute the angle offset ϕ for each of your LIDAR's readings, using the knowledge that the middle bin (e.g., array index 10) is at ϕ=0 since it's pointing directly in front of the robot, and that the total horizontal spread is LIDAR_ANGLE_RANGE radians. Store the result in an array such that:
   lidar_offsets[index] = angle_offset_in_radians.

3. Near the bottom of your while loop, add code to get sensor readings from the robot's LIDAR. The sensor object is named *lidar* in the code, and this can be done with *lidar.getRangeImage()*, which returns an array of distance measurements corresponding to each laser being emitted from your robot.

Part 2: Coordinate Transforms

1. Write down the homogenous transform to convert readings from your LIDAR's frame of reference (x_{lidar}) into the robot's frame of reference (x_r, y_r) and then into world frame coordinates (x_{world}, y_{world}).
   Testing Tip: If the robot is at pose $(0,0,0^o)$ and the LIDAR beam is facing forward (i.e., the reading at index 10 which is offset by 0 degrees) and detects an object 5cm away, your homogenous transform should return $(.05,0)$ for the object's position. Gradually add complexity to your tests.

2. Fill in the Python function **convert_lidar_reading_to_world_coord** that will transform a reading from ePuck's LIDAR into world frame coordinates. Once you are confident that your code is working reasonably well, change the controller state from "stationary" to "line_follower" at the top of your code.
   Implementation Hint: You'll need to convert from (bin,distance) to (x,y) coordinates in the robot's frame of reference. You can assume the LIDAR is mounted at the robot's origin. Next, you'll need to take the (x,y) point in the robot's frame and convert it to (x,y) in the world frame, using a homogeneous transform.

Part 3: Map Construction

1. Choose a resolution for your map (e.g., 3cm per cell). Create a data structure outside of your while loop named world_map[#y_cells][#x_cells] to store and maintain it. Make sure you index by row first, and column second. You may also use a numpy matrix here!
   Tip: The Lab 4 square floor is 1m x 1m.

2. Fill in the function **transform_world_coord_to_map_coord** that takes float coordinates (x, y) as arguments and returns the (row,col) of the grid map location that it corresponds to as (i, j). Make sure to check whether world_coord is within the allowable coordinate range for your map – return None if it isn't.

3. Fill in the function **transform_map_coord_to_world_coord** that takes a grid location (i, j) and returns the (x, y) world coordinate at its center. Make sure to check whether (i,j) is valid or not, returning None if it's an invalid grid cell.

4. Add a call to **update_map** somewhere near the bottom of your controller's while loop, passing in the array of sensor readings from the LIDAR, resulting in an update to your map (which you'll actually implement next).

5. Fill in the function **update_map**, which takes an array of lidar sensor readings and updates world_map accordingly. Using the value "1" to indicate obstacles and "0" to indicate free, navigable space, use the functions from Part 2 and Part 3 with the LIDAR readings to populate your map. Make sure to ignore any LIDAR beams with distance >= LIDAR_SENSOR_MAX_RANGE or any beams with distance >= the longest possible diagonal given the square world size (1m x 1m)! Convert valid LIDAR readings into world coordinates, find the corresponding map grid cell, and if valid, mark the cell in your map as an obstacle.

Part 4: Visualization and Path Planning Prep

1. Add a call to **display_map** after your **update_map** function call. Then, fill in the function **display_map** that will visualize your map as your program runs. You will have to pick a resolution for each 'cell' from your map, drawing each part of the environment as empty or occupied. You may implement this as terminal output (e.g., using Python's print statements) or as a graphical visualization in a manner of your choosing. You should also visualize the current position of the robot on the map. Keep in mind that you'll probably want to iterate over your map's rows in **reverse order** since the terminal prints top-to-bottom (with (0,0) as top-left corner), and your map is probably oriented bottom-to-top (with (0,0) as bottom-left corner).
   Extra Credit Opportunity: If you produce a graphical (non-terminal) map visualization, you will be given up to **20 extra credit points** based on quality, at the grader's discretion.

Part 5: Lab Report

Create a report that answers each of these questions:

1. What is the drawback of a data structure that stores distances between every possible pair of nodes in the graph? How does the implementation in 4-2 address this problem?
2. What are the names of everyone in your lab group?
3. Roughly how much time did your group spend programming this lab?
4. Go to https://forms.gle/3LbtRx3o66mjEX347 and read the form text. If merited, please use it to confidentially indicate any difficulties or problems that you're experiencing with your team.