Lab 5: Path Planning

CSCI 3302: Introduction to Robotics

Due 28/03/25 @ 11:59pm

2pts/day Extra Credit for Early Submission (Max 10)

The goals of this lab are

- Understand the challenges of map construction and map-aided navigation.
- Implement a search algorithm (**Dijkstra's algorithm or A***) to find the shortest path on a grid map given a start point and goal point.
- Use path planning to generate a discrete set of waypoints for a robot to follow.

You need

- Lab 5 Webots starter code
- An understanding of Lab 3 (IK Controller), Lab 4 (Mapping), Dijkstra's Algorithm or A*, and State Machines
- A Python3 environment with numpy, matplotlib and scipy libraries

Dijkstra's algorithm takes a graph with N nodes as an input and returns the minimum cost of reaching every vertex when starting from a user-specified source vertex. A* adds a "heuristic", for example the distance as-the-crow-flies to the path cost, which usually leads to finding solutions faster. Using this information, we can find a path of vertices connecting a start location to a goal location. In order to make use of it, we have to convert the vertices on the path into coordinates we can have the robot travel to. Recall that with our inverse kinematics-based feedback controller from Lab 3, we can use a desired goal pose to figure out the wheel rotations that make the robot travel there! Therefore, combining these two ideas will allow us to use path planning on a graph to enable a robot to traverse a physical, continuous space.

Getting the map representation of the environment is the first challenge we will solve. The laser is noisy, meaning its readings cannot be relied upon entirely, requiring you to reject erroneous measurements. You then need to transform the map into a configuration space representation (by padding obstacles, allowing the robot to be treated as a point mass) to be able to compute collision-free paths. This lab consists of three parts:

1. Generate a map using a manual controller to teleoperate the robot around the environment, and save it to the filesystem.

2. Load the map from the filesystem and transform it into a configuration space representation. Use this representation to generate a path from a given start state to a given goal state, saving the path (list of waypoints) to the filesystem.
3. Load the list of waypoints in Webots and invoke an IK-based Feedback Controller to navigate the robot to its goal.

Assessment

1. We will test your map with our path planning algorithm. The map needs to include all relevant obstacles while also providing an opportunity for the robot to drive into each room in the apartment (except the laundry under the staircase).
2. We will test your planner with our map. We expect your results to come close to the shortest distance.
3. We will test your Webots controller with our waypoints to see whether your solution reliably navigates to the different rooms.

Part 1: Mapping

1. **Make sure you are in the *"manual"* mode** by checking the value of variable ***mode***. The Webots controller allows you to control the robot using the arrow keys on your keyboard (make sure you hit "play" on the simulation first!). We have already provided this manual mode controller for you. Use the arrow keys to control the robot. Pressing the "s" key saves the current map in the NumPy file format (map.npy). The controller uses the same approach that you developed in Lab 4 to plot LIDAR detections on the display. Perform some mapping and take note of the problems you observe, especially regarding noisy measurements.

2. To address this challenge of unreliable measurement, we will treat each measurement as *evidence* of an obstacle, rather than definitive proof of it. As we get more measurements on the same spot, we can be more confident that an obstacle exists there. Create a world map as a 2-dimensional NumPy array of floating-point values (the display is 360x360 pixels). Whenever a reading is found at a cell on the map, increment the corresponding entry by a small value (e.g., 5e-3) making sure it does not exceed 1. Keep in mind that your robot's control loop will be executing very quickly (every 32ms), so this number will grow rapidly.

3. Visualize your map by changing the color of your map display drawing routine. You can use the following equation to convert a gray value **g** in range [0,1] into the appropriate hexadecimal format:

   color = (g*256**2+g*256+g)*255

4. Once you are satisfied with your map, apply a threshold to reject all values that are below a certain value. In NumPy you can use *array>0.5*, to get an array of Booleans with "True"

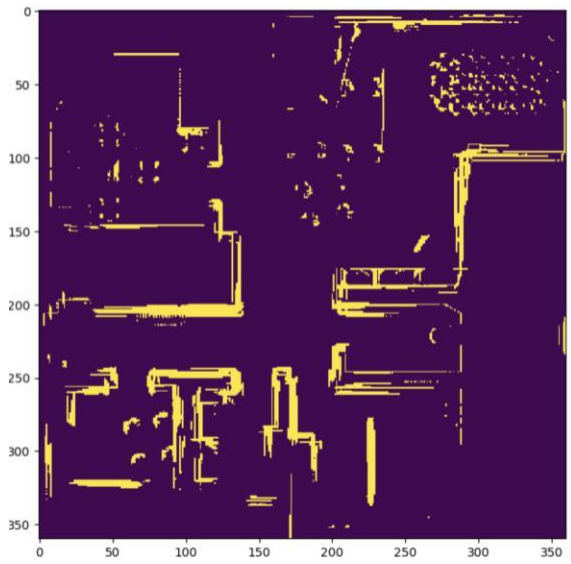
entries indicating entries in the original array that satisfy the provided criteria (e.g., all values greater than 0.5). You can then multiply this array (np.multiply) with 1 to convert it back to an integer array. Use NumPy's save method to store your data structure, making sure to name the file *map.npy*.

Part 2: Path Planning

1. **Set the mode to 'planner'**. This will prevent the robot controller from running its main *while loop* to allow your program to compute a path that the robot will follow in the last part of the lab. Load the map that you saved to the filesystem in Part 1. Visualize the map using a library of your choice, for example using matplotlib as shown below.

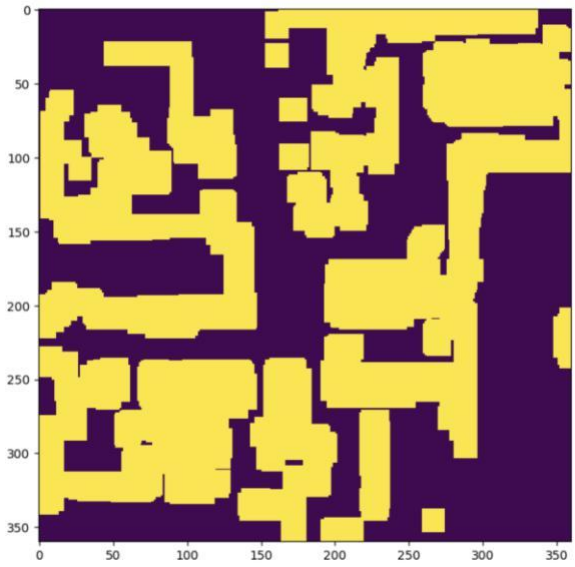
   Please comment out all the *plt.show()* statements before submitting. Note that all the following images are flipped horizontally. If you want to visualize them without them being flipped you can plot a horizontally flipped map using the np.fliplr function as shown here: plt.imshow(np.fliplr(map))

   ```
   from matplotlib import pyplot as
   plt import numpy as np
   plt.imshow(map)
   plt.show()
   ```

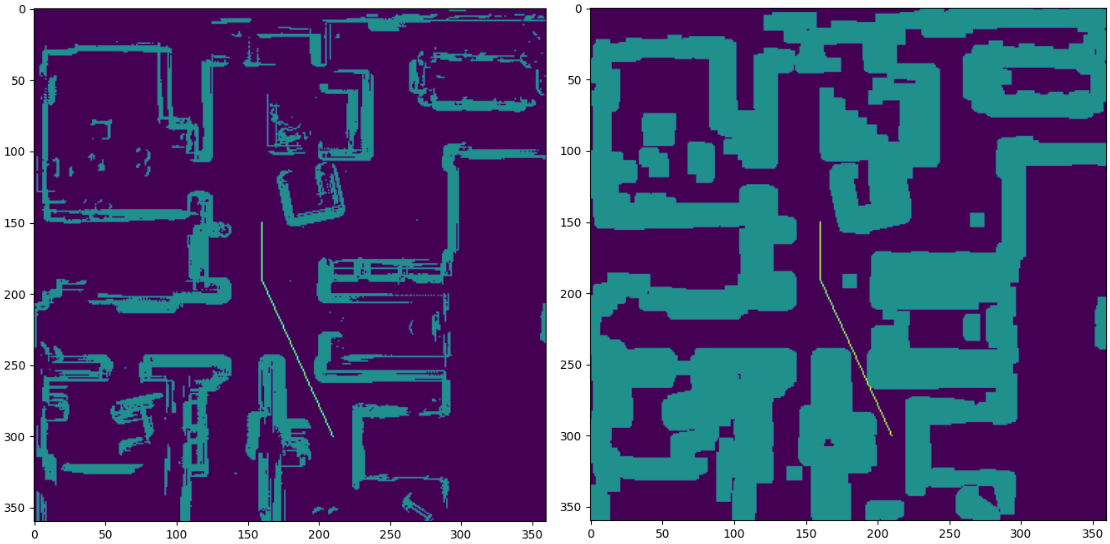
   

2. Compute an approximation of the "configuration space". You can do this by drawing squares that are centered on every pixel that contains an obstacle or by convolving the image with a quadratic kernel of ones. (This will be covered in the Computer Vision lecture.) There is no right or wrong approach for this so long as there is a collision-free path between rooms. Your goal is to create a map that leaves channels that are at least

one pixel wide to still allow the robot to enter every room while preventing the robot to collide with obstacles.



3. **Implement a path planning algorithm** (*if you use any external sources, cite them in your code*) that allows you to compute the shortest path (as a list of coordinate tuples) between two arbitrary points in free space using your configuration space map. You can use Dijkstra or A* to accomplish this. A possible result is shown below. Return an empty list if no such path exists. Use the ***start_w*** and ***end_w*** to provide start and end coordinates in the webot's coordinate frame. Set the ***start_w*** as the initial pose of the Tiago when you start the world. Try a few ***end_w*** once you have developed the planner.



4. Convert the list of path waypoints (which are expressed in the 360x360 map coordinate system) into waypoints in the **12m**x**12m** world coordinate system and save them to disk as ***path.npy***. Remember, the top-right corner of the horizontally flipped map is at (0,0) (in meters), the bottom left corner is at (-12, -12).

Part 3: Path Following

1. **Set the mode to 'autonomous'**. Load the path from the disk as goal waypoints and visualize them on your map.
2. Implement the controller from the IK lab to drive along your list of goals. You are encouraged to use a "state" variable to index into the goal coordinates, incrementing it as the robot approaches its current goal.
3. Test your solution by generating paths to different rooms in the environment.

Part 4: Robot Pick-up and Place

1. Import the robot movements from 'lab5_joint.py'. Edit 'picknplace' mode accordingly to move the robot to various locations based on path generated from the planner, perform a pick-up task, and place it 'start_ws' location in the environment.
2. Use the given start_ws, end_ws for finding the object 'orange', once your robot reaches the first end_ws, pick up the object recognized 'orange' and place on it the slab at the start_ws. Do it for next start_ws, end_ws.
3. Test your solution by generating paths to different rooms in the environment.

What to submit

1. Your controller file with name unchanged.
2. map.npy file.
3. Attach an image of your map and your c-space to the report pdf.
4. For ***end_w = (10.0, 7.0)*** *# Pose_X, Pose_Z in meters* and the ***start_w*** as the initial pose of Tiago when the world starts, attach the map showing the shortest path on the map to the report pdf.

5. Attach a video of at least one pick and place operation
   Note: The report pdf should have the above 3 images.

Part 4: Lab Report

1) What are the names of everyone in your lab group?
2) Why is sensor noise problematic when mapping?

3) How did you choose the value with which you increment the map entry? What happens if the value is too small, what happens if it is too large?
4) How did you choose the value to threshold your map? What happens if the value is too small, what happens if it is too large?
5) As Tiago is traveling along the path that the planner provided, suppose it detects an object in its way. How could you modify your solution to plan around/gracefully handle this unforeseen object in the robot's path?
6) Could we use an algorithm like RRT to generate a viable path instead of Dijkstra's algorithm/A*? If yes, how would the path look different? If no, why not?
7) Roughly how much time did you spend programming this lab?