16-350: Planning Techniques for Robotics
Homework 2: High-DoF Arm Planning
Due: Mar 12 (Wed), 11:59pm

Professor: Maxim Likhachev *Spring 2025 TA: Saudamini Ghatge*

1 Task

Write planner(s) for a planar arm to move from its start joint angles to the goal joint angles. Your planner must return a plan that is collision-free.

2 Code

Your code is within the folder `code`. Your planner must output the entire plan, i.e., all sets of joint angles from start to goal. The planner should reside in `planner.cpp` file. Currently, the planner function contains an interpolation-based generation of a plan. That is, it just interpolates between start and goal angles and moves the arm along this interpolated trajectory. It doesn't avoid collisions. As mentioned before, your planner must return a plan that is collision-free. The planner function (inside `planner.cpp`) is as follows:

```
static void planner(
     double *map,
     int x_size,
     int y_size,
     double *armstart_anglesV_rad,
     double *armgoal_anglesV_rad,
     int numofDOFs,
     double ***plan,
     int *planlength
     )
{ // Planner code }
```

3 Inputs

The `map` of size `(x_size, y_size)` contains information on what are obstacles and what are not. `armstart_anglesV_rad` and `armgoal_anglesV_rad` describe the start and goal configurations of the arm respectively (angles are all **clockwise** from the x-axis, in **radians**). `numofDOFs` is the number of joints characterizing the arm configuration.

You will not have to worry about accessing map indices or understanding angle specifications, as we have provided a tool that verifies the validity of the arm configuration: this is all you would need for planning. In the starter code, you will see how this function: `IsValidArmConfiguration(angles,`

`numofDOFs, map, x size, y size)` is being called to check for the validity of a configuration. Note: We do not check for self-collisions in this function (for simplicity), so do not worry about self-collisions (i.e. ignore them for this assignment).

4 Outputs

The planner function should return the entire plan and its length (see the current planner inside `planner.cpp` to understand how to interpret these variables).

When you run your code, you should be able to see the arm moving according to the plan you returned. If the arm intersects any obstacles, then it is an invalid plan. You might notice that the collision checker is not very rigorous and it might allow the arm to slightly brush through the obstacles sometimes (which is okay!).

5 Execution

The code folder contains two map files (map1.txt and map2.txt).

We are *not* using MATLAB for visualizations. Instead, the planner.cpp will produce a stand-alone executable that will be fed in arguments via command line. The planner takes in the input map file, number of degrees or freedom (numofDOFs), start angles in radians comma separated - **not space separated** (e.g. 1.4,3.12), goal angles comma separated - **not space separated**, planner id (integer), and output file path. The planner function should then call the appropriate planner based on the planner id and write a path into the output file.

Planner ids: $0 = RRT \; || \; 1 = RRT-Connect \; || \; 2 = RRT^* \; || \; 3 = PRM$

Note that the input parsing and output file are handled for you. You just need to focus on the planning part.

5.1 Running the Code

To compile on linux (mac or windows may require a substitute for g++, e.g. clang):

$ **g++ planner.cpp -o planner.out**

This creates an executable, namely planner.out, which we can then call with different inputs:

$ ./planner.out [mapFile] [numofDOFs] [startAnglesCommaSeparated] [goalAnglesCommaSeparated] [whichPlanner] [outputFile]

Example:

$ **./planner.out map1.txt 5 1.57,0.78,1.57,0.78,1.57 0.392,2.35,3.14,2.82,4.71 2 myOutput.txt**

This will call the planner and create a new file called `myOutput.txt` which contains the resultant path as well as the map it was run on.

Note: 1.57 = pi/2, 0.78 = pi/4, 4.71=pi*3/2, we are just inputting start and end positions in radians. grader.py has a function called convertPIs that can be helpful to create start & goal positions with respect to pi.

Visualizing your output: We have provided a python script that parses the output file of the C++ executable and creates a gif.

Example:

$ **python visualizer.py myOutput.txt --gifFilepath=myGif.gif**

This will create a gif `myGif.gif` in your project folder that visualizes the plan from `myOutput.txt`. See the comments in `visualizer.py` for more details, feel free to modify this file. This file is purely there for your benefit.

When you run it, you should be able to see the arm moving according to the plan you returned. If the arm intersects any obstacles, then it is an invalid plan. Since the collision checker might allow slight brushing through the obstacles sometimes, that is okay.

6 Submission

You will submit this assignment through Gradescope. You must upload one ZIP file named `<andrewID>.zip`. This should contain:

1. A folder `code` that contains all code files, including but not limited to, the ones in the homework packet. If there are subfolders, your code should handle relative paths.
2. A single executable named `planner.out` which we will directly use with `grader.py`
3. Your writeup in `<andrewID>.pdf`. This should contain a summary of your approach for solving this homework, results, and instructions for how to compile your code. Specifically discuss any hyper-parameters you chose and how they affected performance (this does not need to be too thorough, but we do want to see some thought). Do not leave any details out because we will **not** assume any missing information. There should be one line to compile the code.

   Undergraduate students:

   Implement both RRT-Connect and PRM for this assignment. For 5 randomly generated start and goal pairs on `map2.txt`, run each start/goal pair 4 times for each planner, since the planners are inherently stochastic (5 x 4 = 20 runs). Report the **mean** and **standard deviation** for each of the following statistics:

   (a) Planning times
   (b) Path costs
   (c) Number of vertices generated (in constructed graph/tree)
   (d) Success rate for generating solutions within 5 seconds

   Extra Credit:

(a) +5pts for implementing RRT

(b) +10pts for implementing both RRT and RRT*

Graduate students:

You must implement **all four** planners:

(a) RRT

(b) RRT-Connect

(c) RRT*

(d) PRM

For 5 randomly generated start and goal pairs on `map2.txt`, run each start/goal pair 4 times for each planner, since the planners are inherently stochastic (5 x 4 = 20 runs). Report the **mean** and **standard deviation** for each of the following statistics:

(a) Planning times

(b) Path costs

(c) Number of vertices generated (in constructed graph/tree)

(d) Success rate for generating solutions within 5 seconds

Note: Path cost can be determined by taking summation over the changes in joint angles from start configuration to goal configuration.

Note: You must randomly generate the start and goal pairs once and fix them for all planners.

Compile and explain your results. Justify your conclusions about which planner you think is the most suitable for the environment, giving reasons why. What issues does it still have, and how do you think it could be improved.

Extra Credit (+5): Present and compare additional statistics such as consistency of solutions (e.g., for different runs with similar start and goal, how much variance do you observe in solution paths and qualities), or time until first solution (for RRT* versus other planners). Thorough/insightful analysis of hyper-parameters is also eligible for extra credit.

7 Grading

The grade will depend on:

1. Correctness of your implementations (optimizing data structures, e.g., using kd-trees for nearest neighbor search, is **not** required) (50% weightage).

2. The speed of your solutions. We expect that you will achieve solutions within 5 seconds most of the time (for graduate students, this **must** be true for at least two of your planners). (20% weightage)

3. Relative cost of your solutions. Since these are arbitrarily suboptimal algorithms given a finite amount of time, we don't expect any particular solution values. However they should be "reasonable", and certain algorithms should have lower cost than others (e.g. RRT* should be cheaper than RRT). (20% weightage)

4. Results and discussion. (10% weightage)

5. Extra credit

Note: To grade your homework and to evaluate the performance of your planner, we may use a different map and/or different start and goal arm configurations. We will not test arm configurations with more than 7 degrees of freedom.

We have a `grader.py` where we will feed in our own set of maps, numDOFs, and start/goal locations. We plan to run `grader.py` on your executable and will assign points accordingly. To ensure your C++ executable is compatible, we have included `grader.py` with small mock data, as well as a `verifier.cpp/verifier.out` that `grader.py` calls. `verifier.cpp/.out` just collision checks the output path using the same `IsValidArmConfiguration` to ensure that it is collision free, and that the start and goal positions are consistent.

We will quite literally run `grader.py` as written with our own data, so please ensure your final solution is compatible with it. Test by running and see if the output csv gets created:

$ *python grader.py*

Feel free to create a copy and modify this code to run experiments!