Assignment 1: Planning for High-DOF Planar Arm

course code:-SC-627

January 2024

1 Description

In this Assignment, you will implement different sampling-based planners for the arm to move from its start joint angles to the goal joint angles.The planner should reside in the **planner.cpp file**. Currently, we provide a function that 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 does not avoid collisions. Your planner must return a plan that is collision-free. Note that all the joint angles are given as an angle with x-axis, clockwise rotation being positive (and in radians). So, if the second joint angle is $\pi/2$, then it implies that this link is pointing exactly downward, independently of how the previous link is oriented. Having said this, you do not have to worry about it too much as we already provide a tool that verifies the validity of the arm configuration, and this is all you need for planning. To help with collision checking we have supplied a function called **IsValidArmConfiguration**. It is being called already to check if the arm configurations along the interpolated trajectory are valid or not. So, during planning you want to utilize this function to check any arm configuration for validity. An example linear interpolation planner function inside **planner.cpp** is as follows:

Listing 1: Planner function

```
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)
{
    // Linear interpolation planner code here...
}
```

Inside this function, you can see how any arm configurations are being checked for validity using a call to IsValidArmConfiguration(angles, numofDOFs, map, x size, y size). You will also find code in there that sets the returned plan (currently to a series of interpolated angles).

2 TASK

The **planner.cpp** file will produce an executable that will be fed in arguments via the command line. The planner takes in the input map file, number of degrees of freedom (numofDOFs), start angles comma separated - not space separated (e.g. 1.4,3.12), goal angles comma separated, planner ID (integer), and output file path. The planner should then call the appropriate planning algorithm based on planner ID and write a path into the output file. Planner IDs: 0 = RRT
1 = RRT*
2 = PRM
Your assignment is to code up these three algorithms to return collision free paths. Note that the input parsing and output file are handled for you, you just need to focus on the planning part.

3 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 (optional but may
      be necessary: -std=c++11)
  ```

To enable debugging, add a -g tag:

- ```
  g++ planner.cpp -o planner.out -g
  ```

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

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

  Replace the placeholders with your specific values.
For 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.

4 Visualizing the planner

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
```

Replace `myOutput.txt` with the actual output file and `myGif.gif` with the desired GIF file name. This command runs the `visualizer.py` script, providing the output file as input and specifying the file path for the generated GIF.
This command will create a GIF (`myGif.gif`) that visualizes the plan from `myOutput.txt`. Refer to the comments in `visualizer.py` for more details, and feel free to modify this file. The `visualizer.py` script is provided for your benefit.

When you run the visualizer, 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. Please note that the collision checker may not be of very high quality and might allow slight brushing through obstacles sometimes, which is acceptable.

NOTE: We do NOT check for self-collisions inside **IsValidArmConfiguration** for simplicity. You are allowed to continue to ignore self-collisions for the assignment, but be aware that a real-robot collision checker needs to take them into account.

5 Report

Provide a table of results showing a comparison of:

- Average planning times.
- Success rates for generating solutions in under 5 seconds
- Average number of vertices generated (in a constructed graph/tree)
- Average path qualities

for each of the three planners with a brief explanation of your results. To generate the results, use map2.txt and run the planners with 20 randomly generated start and goal pairs (randomly generate the pairs once and fix those for all the planners). Compile the statistics and write a paragraph summarizing the results and make a conclusion for each of the following points:

- What planner you think is the most suitable for the environment and why
- What issues that planner still has
- How you think you can improve that planner

6 Grading

- The correctness of your implementations
- Finding solutions within a fixed reasonable budget of time (i.e. 5 seconds)
- The speed of your solution. At least one of your planners should achieve solutions within 5 seconds.
- 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).
- Result and discussion

We will grade your code using the `grader.py` file 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` that `grader.py` calls (please compile `verifier.cpp` using the line `g++ verifier.cpp -o verifier.out`). The `verifier.cpp` file collision checks the output path using the same `IsValidArmConfiguration` function to ensure that the path is collision-free, and that the start and goal positions are consistent. The `grader.py` file will output a CSV file called `grader results.csv` which summarizes the results of the tests. We will quite literally run `grader.py` as written with our own data, so please ensure your final solution is compatible with it.

7 Submitting the assignment

Submissions need to be made through teams and they should include:

- A folder named code that contains all relevant C++ files.
- A single executable named planner.out which we will directly use with `grader.py`.
- A PDF writeup named `< name-IITBID>.pdf` with results and high level details about your code.
- This PDF writeup, named `< name-IITBID>.pdf`, contains the results and high-level details about 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. Additionally include instructions on how to compile your program in case we need to compile it later on.