16-782, Fall'22, Planning and Decision-making for Robotics

Homework II:
Planning for a high-DOF planar arm

DUE: Oct 26 (Wednesday) at 11:59PM

Description:
In this project, you will implement different sampling-based planners for the arm to move from its start joint angles to the goal joint angles. As before, the planner should reside in *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 π/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:

```
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 configuration is 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).

Your task:
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, planner id (integer), and output file path. The planner should then call the appropriate planner based on planner id and write a path into the output file.
Planner ids: 0 = RRT | 1 = RRT-Connect | 2 = RRT* | 3 = PRM

Your assignment is to code up these four 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.

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
```
To enable debugging, add a -g tag, e.g. >> 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]
```
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 in 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 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. You might notice that the collision checker is not of very high quality and it might allow slightly brushing through the obstacles sometimes (that's okay).

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 note that a real-robot collision-checker needs to take them into account.

Your report:
Provide a table of results showing a comparison of:
1. Average planning times
2. Success rates for generating solutions in under 5 seconds
3. Average number of vertices generated (in a constructed graph/tree)
4. Average path cost

for each of the four planners with a brief explanation of your results.

To generate the results, use map2.txt and run the planners with 5 randomly generated start and goal pairs (randomly generate the pairs once and fix those for all the planners). Since these planners are inherently stochastic, run each start/goal configuration 4 times and report the average and standard deviation for each statistic (i.e. across all 5x4=20 runs). At the end, compile the statistics and write a paragraph summarizing the results and make a conclusion each of the following points:

1. What planner you think is the most suitable for the environment and why
2. What issues that planner still has
3. How you think you can improve that planner

Extra Credit (5 points): You are also encouraged to present additional statistics such as consistency of solutions (how much variance you observe in solution for different runs with similar start and goal for example), or time until first solution (for RRT* versus other planners). Thorough/insightful analysis of hyper-parameters is also eligible for extra credit.

Extra Credit (15 points): Modify the SAMPLE() function in RRT* to incorporate the Sampling Heuristics (Node Rejection (NR), Local Biasing (LB) and Combined (CO)) introduced in this paper [1]. Compare with vanilla RRT* and report the performance in terms of solution cost with a plot and a table as shown in Figure 8 and Table 2 in the paper.

Grading:
The grade will depend on:
1. The correctness of your implementations (optimizing data structures e.g. using *k*-d trees for nearest neighbor search is NOT required)
2. Finding solutions within a fixed reasonable budget of time (i.e. 5 seconds).
3. The speed of your solution. At least one of your planners should achieve solutions within 5 seconds.
4. 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).
5. Results and discussion.
6. Extra credit

How we will grade your code:
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. *Any submission that is not will receive 0's for this part of the assignment, we will not be lenient as we have provided everything now to ensure compatibility.* 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!

To submit:
Submissions need to be made through Gradescope and they should include
1. A folder named code that contains all relevant C++ files.
2. A single **executable** named ***planner.out*** which we will directly use with our own grader.py.
3. A PDF writeup named <Andrew ID>.pdf with results and high level details about your code. Specifically discuss any hyper-parameters you faced 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 if we need to compile it later on.

References
[1] Akgun, B. and Stilman, M., 2011, September. Sampling heuristics for optimal motion planning in high dimensions. In 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (pp. 2640-2645). IEEE.