16-782, Fall'18, Planning and Decision-making for Robotics

Homework II:
Planning for a high-DOF planar arm
DUE: Oct 17th (Wednesday) at 11:59PM

Description:
In this project, you are supposed to 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 inside the planner() function. Currently, this 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 does not avoid collisions. Your planner should 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.

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 here... }
```

Inside this function, you will 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). You will need to modify it to set it to the plan generated by your planners.

The directory contains a map file map1.txt. Here is an example of running the test from Matlab when planning for a 5-DOF arm:

To compile the C code:
>> mex planner.cpp

To run the planner
>>startQ = [pi/2 pi/4 pi/2 pi/4 pi/2];
>>goalQ = [pi/8 3*pi/4 pi 0.9*pi 1.5*pi];
>>runtest('map1.txt',startQ, goalQ);

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. You can ignore it or improve it for extra credit.)

NOTE: We do not check for self-collisions inside the `IsValidArmConfiguration`. You can choose to ignore self-collisions for the assignments, but should take them into account for the extra credit.

Also 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.

For this homework you will implement four algorithms:

1. RRT
2. RRT-Connect
3. RRT*
4. PRM

You have to provide a table of results showing a comparison of:

1. Mean planning times
2. Mean number of samples generated
3. Mean path qualities

for each of the four planners with a brief explanation of your results. For the results, you will use map2.txt and run the planners with say 20 randomly generated start and goal pairs (you will randomly generate the pairs once and fix those for all the planners.)

Extra Credit: Implement a sophisticated collision checker for the high-DoF planar arm. It should account for self-collisions. There is an inherent trade-off here between fidelity and speed. Extra credit is only to make up points lost elsewhere, you cannot get more than 100% on this assignment.

To submit:

Submissions need to be made through Gradescope and they should include

1. A folder named *code* that contains all MATLAB and C/C++ source files.
2. A PDF writeup named *<Andrew ID>.pdf* with instructions to compile code, results, and everything we need to know about your implementations and submission. Do not leave any details out because we will not assume any missing information.

Grading:

The grade will depend on two things:

1. The correctness of your implementations (optimizing data structures e.g. using kd-trees for nearest neighbor search is NOT required)
2. Results and discussion.
3. Extra credit