16-782, Fall'21, Planning and Decision-making for Robotics

# Homework II:
# Planning for a high-DOF planar arm
# DUE: Oct 20th (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 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 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.

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 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). 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];
>>planner_id = 0 % placeholder for now
>>runtest('map1.txt',startQ, goalQ, planner_id);
```

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.

NOTE 1: We do NOT check for self-collisions inside `IsValidArmConfiguration`. 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.

NOTE 2: 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 implement four algorithms:

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

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 qualities

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 20 randomly generated start and goal pairs (randomly generate the pairs once and fix those for all the planners). 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).

**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]. In your writeup, 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.

**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:

1. The correctness of your implementations (optimizing data structures e.g. using *k*-d trees for nearest neighbor search is NOT required)
2. The speed of your solution. At least one of your planners should achieve solutions within 5 seconds.
3. Results and discussion.
4. Extra credit

**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.