

Homework II:
High-DoF Arm Planning
DUE: March 26th (Friday) at 11:59PM

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.

Undergraduate Students - Implement either RRT-Connect or PRM

Graduate Students - Implement all 4 algorithms: RRT, RRT-Connect, RRT*, and PRM.

Code

The planner should reside in *planner.cpp* file. Your planner must output the entire plan, i.e., all sets of joint angles from start to goal. 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 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 *plan_length  
)  
{}
```

Check lines 305-309 in planner.cpp to get an idea of how to call different planners.

Inputs

The map of size (x_size, y_size) contains information on what are obstacles and what are not. Value of cell (x, y) in the map can be accessed as:

`(int)map[GETMAPINDEX(x,y,x_size,y_size)]`

If it is equal to 0, then the cell (x, y) is free. Otherwise, it is an obstacle. *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 angles characterizing the arm configuration.

We expect that 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, so do not worry about self-collisions.

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 of very high quality and it might allow slightly brushing through the obstacles sometimes.

Execution

The code directory two map files (*map1.txt* and *map2.txt*). Following is an example of running the test from within MATLAB (in the same directory where all source files are placed).

To compile the cpp code:

```
>> mex planner.cpp
```

```
>> 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; % change for different planners  
>> runtest('map1.txt', startQ, goalQ, planner id);
```

Submission

You will submit this assignment through Gradescope. You need to upload one ZIP file named *<Andrew ID>.zip*. This should contain:

1. A folder named *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. We will only compile your MEX code (according to instructions in the writeup) and execute the *runtest.m* script.
2. A PDF file named *<Andrew ID>.pdf*. This should contain a summary of your approach for solving this homework, results, and instructions for how to compile your code. This should be one line for us to execute in MATLAB of the form “*mex <file 1> <file 2> ... <file N>*”.
 - For 20 randomly generated start and goal configurations on *map2.txt*, report:
 1. Min, max, std, and mean planning times
 2. Min, max, std, and mean path qualities
 3. Min, max, std, and mean number of vertices generated (in constructed graph/tree)
 4. Success rates for generating solutions within 5 seconds
 - For undergraduate students, these stats will be reported for 1 planner. For graduate students, these are reported for all 4 planners.
 - Descriptions and analysis of your results – why are some planners faster or slower? What issues do they have? How can they be improved?

Please **do not** include the map text files in your submission.

Grading

The grade will depend on:

1. Correctness of your implementations.
2. The speed of your solutions. We expect solutions are found within 5 seconds most of the time (for graduate students, this must be true for at least two of your planners).
3. Results and discussion.
4. Note: To grade your homework and to evaluate the performance of your planner, we may use a different map, different start and goal arm configurations, and different arms. We will not test on arm configurations with more than 7 degrees of freedom.

Extra credits

For undergraduate students:

1. +10pts for implementing both RRT-Connect and PRM
2. +15pts for implementing all 4 planners

For both undergraduate and graduate students:

+10pts for testing >100 random start and goal pairs and generating an *area under the curve plot*, where the x-axis is time it takes to find a solution and the y-axis is percentage of solutions found

within a give time. For example, if all solutions were found at exactly 3 seconds, this plot would have a horizontal line $y = 0$ in the x range $[0, 3]$, then immediately jump to another horizontal line $y=1$ in the x range $[3, \text{upper } x \text{ lim}]$.