| **16-782: Planning and Decision-Making in Robotics** |
| :---: |

**16-782: Planning and Decision-Making in Robotics**

**Homework 2: Planning for High-DoF Arm**

**Due: October 22, 2024, 11:59pm**

*Professor: Maxim Likhachev*                    *TAs: Yorai Shaoul, Itamar Mishani*

# 1  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 in homework 1, 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 $\frac{\pi}{2}$, then it implies that this link is pointing exactly downward, independently of how the previous link is oriented. Having said that, 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:

```cpp
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 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-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.

## 3   Building and Running the Code

Open a terminal and navigate to the `code` directory. Then:

```
>> mkdir build
```

```
>> cd build
```

### 3.1   Building with Cmake

```
>> cmake ..
```

```
>> cmake --build .  --config Release
```

To build in debug mode change `Release` with `Debug`. Now, you can run the code with:

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

Example:

```
>> ./planner 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
```

### 3.2   Building with g++

To compile on Linux (Mac or Windows may require a substitute for g++, e.g. clang):

```
>> g++ ../src/planner.cpp -o planner
```
(optional but may be necessary: `-std=c++11`)

To enable debugging, add a `-g` tag:

```
>> g++ ../src/planner.cpp -o planner -g
```

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

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

Example:

```
>> ./planner 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 Output

We have provided a python script that parses the output file of the C++ executable and creates a gif.

Example:

```
>> python scripts/visualizer.py myOutput.txt --gifFilepath myGif.gif
```

This will create a gif `myGif.gif` that visualizes the plan from `myOutput.txt` within the `output` directory. See the comments in visualizer.py for more details, and feel free to modify this file. This file is purely there 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. 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.

# 5   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 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). Compile the statistics and write a paragraph summarizing the results and make a conclusion for each of the following points:

1. What planner do you think is the most suitable for the environment and why.

2. What issues does the planner still have?

3. How do 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 the solutions for different runs with similar start and goal points, 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 report, compare against 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.

# 6 Grading

The grade will depend on:

1. The correctness of your implementations (optimizing data structures, e.g., using kd-trees for nearest neighbor search, is **not** required).

2. Finding solutions within a fixed reasonable time budget (i.e., 5 seconds).

3. The speed of your solutions. At least one of your planners should achieve solutions within 5 seconds.

4. Results and discussion.

5. Extra credit.

We will grade your code using the `grader.py` file, where we will feed in our own set of maps, numD-OFs, 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++ ../src/verifier.cpp -o verifier` from within `build` directory). 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.

To run `grader.py`, do the following:

From `code` directory:

```
>> cd scripts
```

```
>> python grader.py
```

# 7 Submission

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` within `build` directory which we will directly use with `grader.py`.

3. A PDF writeup named `<AndrewID>.pdf` with 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.

## References

[1] Baris Akgun and Mike Stilman. Sampling heuristics for optimal motion planning in high dimensions. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2640–2645, 2011. doi: 10.1109/IROS.2011.6095077.