Middle East Technical University | Department of Computer Engineering

CENG 460

Introduction to Robotics

Fall 2020-2021

Programming Assignment II - Configuration Agnostic RRT*

Due date: 04.02.2021, Thursday, 23:55

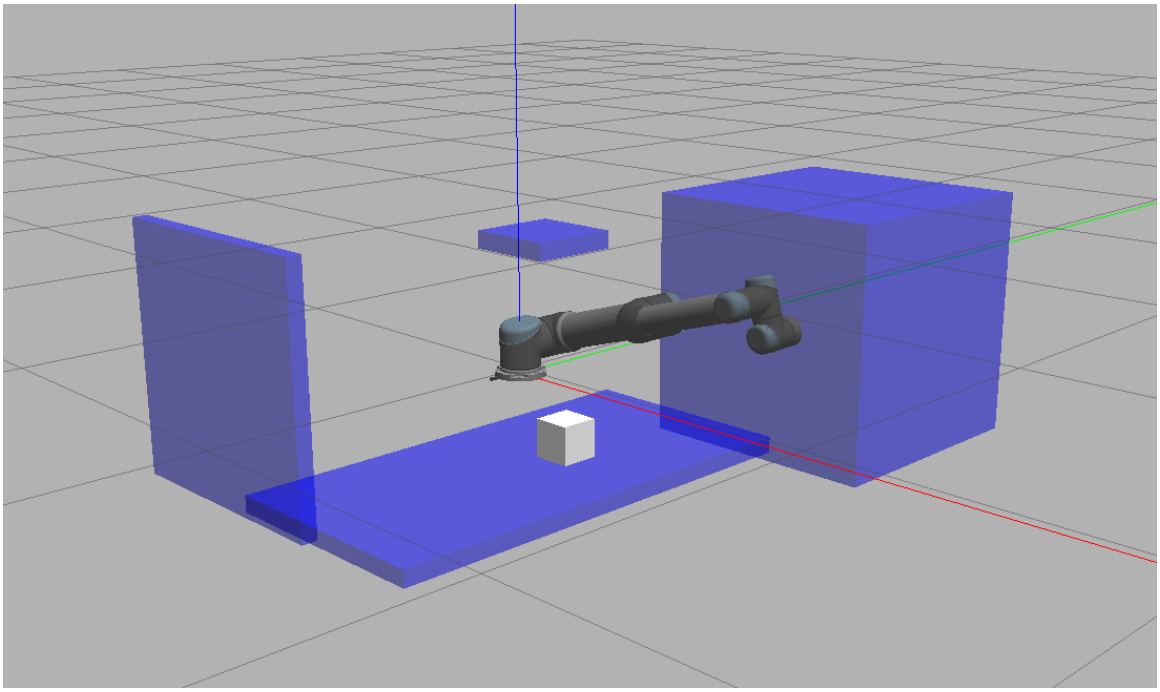

Figure 1: A sneak peek of part 2 of the assignment.

The purpose of this assignment is to increase your intuition of sampling based planning algorithms and assess your ability to implement them.

1 Overview

In this assignment, you will implement the RRT* algorithm for a 6-DOF robot. Since visualisation is not possible in higher dimensions, it is difficult to debug the problems within your algorithm. Therefore, you will implement a configuration agnostic RRT* where the algorithm itself does not know what kind of configuration space it operates. You will then first test this algorithm in a basic 2D environment and then transfer it to the 6-DOF robot case after you are satisfied with the performance.

2 Preliminaries

Most of the ROS heavy parts are already implemented for you about the 6-DOF robot simulation, if you have done the necessary readings for PA1, you should be able to understand the big picture. However, you can review the following to have a deeper understanding the implementations:

- `actionlib` package: a sophisticated protocol over topics that acts like a service but with intermediate results delivered to the client.
- `ros_control` package: an interface that decouples low-level controllers and high level ROS nodes.
- MoveIt!: a very popular planning library. It also provides a `move_group` node that other nodes can make planning requests.

3 The Setup

The base part of the RRT and the 2D test does not require ROS. A Python 2.7 installation with `numpy` and `matplotlib` packages is enough.

For the ROS part, unfortunately Riders option is not available yet.

ROS in Local Machine

- Install MoveIt!, following the instructions on the webpage. Make sure you install it for ROS-Melodic!
- Build `universal_robots` package from source in your workspace for the UR5 robot that you will plan for and its controllers.
- It is the same as PA1 after this. In your workspace:

```
> git clone https://gitlab.com/cemonem/ceng460_hw2.git
> cd ..
> catkin_make
> source devel/setup.sh
> roslaunch ceng460_hw2 ceng460_hw2.launch
```

You should see your robot moving for a while and then stop. After everything is set, start working on your assignment by modifying `ceng460_hw2/scripts/cube_placer.py`.

4 Specifications

In order to give you priorities in your implementation, most of the tasks are seperated and given **tentative** percentages. Again, these percentages **are subject to change** and are just given to you so that you can deduce which task should be attacked first. **Note that 6-DOF Robot part will not be graded if your algorithm fails in 2D case, no matter what the final percentages are!**

RRT* has many variants. We will follow the implementation in the proof paper of Karaman et. al., p. 16, Algorithm 6, with some twists:

- **Goal Biased Sampling:** We will try to find a path to a single configuration in this assignment. Since the probability to land on a single point in the space is 0, we will modify the sampling such that the goal configuration is sampled with probability p, and uniform sampling will happen with $1 - p$.
- **Greedy Steering:** Instead of taking a single step towards the sampled point from the nearest configuration in RRT, the algorithm should take as many steps it can take without colliding. This is a staple to goal biased sampling in the case of goal region being a single point. The steering should assume a linear trajectory between two configurations (c_1, c_2) with $c_1 \oplus \lambda(c_2 \ominus c_1),\ \lambda \in (0, 1]$ (addition $\oplus$ and subtraction $\ominus$ nature may

differ with respect to configuration space). The step size ss should be constant in terms of distance $d(c_1, c_2)$ (definition differs with respect to configuration space). If the distance between last point tested and c_2 is less than ss or $d(c_1, c_2) < ss$, c_2 is tested against collision. The first point to be tested is $c_1 \oplus \frac{ss}{d(c_1,c_2)}(c_2 \ominus c_1)$.

- **Collision-Free Checks:** should be done in the same fashion with greedy steering.
- **k-neighborhood:** The *Nearest* function in the original algorithm should return the closest configuration in RRT in terms of $d(c_1, c_2)$ in this assigment. Ties should be broken randomly. The *Near* function returns k closest configurations in RRT. Ties should be broken randomly for *Near* as well.

4.1 Base RRT* (50%)

We will implement the configuration agnostic parts of the RRT class in `rrt_base.py`. Except `distance, steer, allclose, sample, collision_free, valid` implement all of the methods according to their descriptions (you may use the unimplemented methods inside the other methods). Leave the rest empty.

4.2 2D RRT* (15%)

It is time to test your algorithm. Override `steer` and `collision_free` in `RRTStar2D` class in `rrt_2d.py`. The others are implemented for you. Assume the configuration space is a rectangle of 2D `numpy` vectors, which represent position of a mobile robot. This robot is only able to draw line segments towards the points it wishes to go.

After you fill in the descriptions, run the script and check for faults in your implementation in the animations (for example, the goal cost should decrease as more nodes are added for RRT*). If you are convinced your algorithm works, plot extreme values (too high, too low) for each setting `p, k, step_size` with the others held constant. The most important one is shown as an example in Figure 2: if `k` is 1, the algorithm reduces to RRT, therefore the goal path never changes and it is suboptimal. Add your plots in a short report with explanations regarding the effects of each setting.

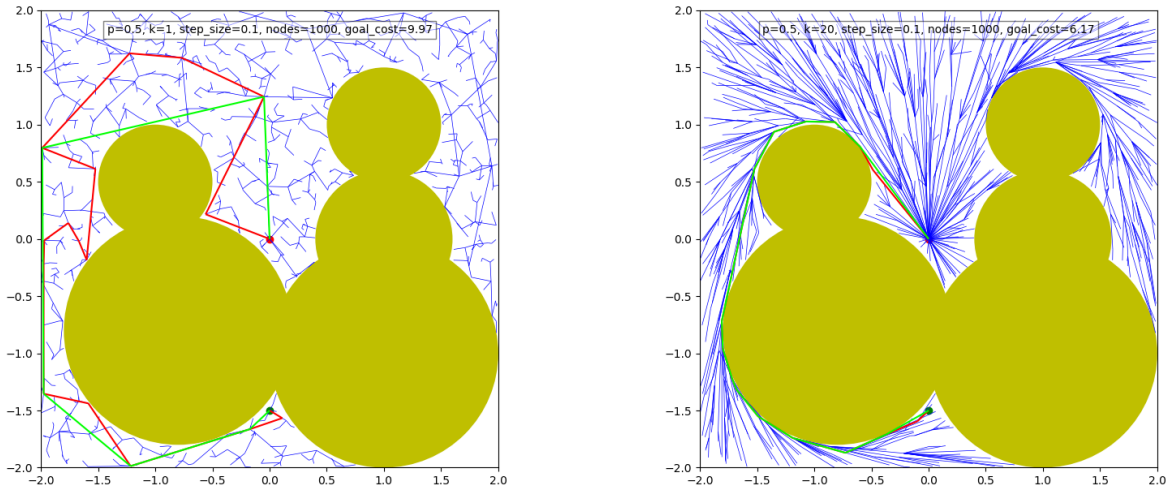

Figure 2: RRT and RRT* compared. The red line is the goal to path, the green line is the simplified path. Observe how RRT* wraps around the obstacle optimally. In contrast, even the simplified path is suboptimal in RRT.

4.3 RRT* for 6-DOF Robot - No Obstacles (15%)

After you are convinced your algorithm operates correctly, do the same procedure for `RRTStar6D` class in `cube_placer.py` as you did for 2D case (this time you will implement the sampling, distance and collision checking too). You may want to use `Ceng460Hw2Environment` methods in `ceng460_hw2_environment.py` during your implementation.

Attack the cube placing task after you are done with your implementation with no obstacles:

```
>roslaunch ceng460_hw2 ceng460_hw2.launch spawn_obstacles:=false
```

- The cube is spawned with identity global orientation, with the indicated in `object/init_pose` parameter. The end effector (with the tf label `"ee_link"`) can only attach to the object with a specific relative pose. This relative pose is hard-coded in the plugin source file provided to you, `ceng460_hw2_world_plugin.cpp`. Attach to this cube using `Ceng460Hw2Environment.attach` method.
- Place the object in the position indicated in `object/final_pose` with identity global orientation.
- Detach, and call `Ceng460Hw2Environment.declare_success` as your last procedure.

4.4 RRT* for 6-DOF Robot, Custom Ad-Hoc Functionalities (10%)

Attack the cube placing task with obstacles this time. You might have noticed how slow collision checking is from the easier version of the task. This is because Moveit! does not provide any interface for collision checking other than a ROS service, and that is very slow compared to collision checking within the same process. Since it will take forever to find a plan with how slow collision checking is, utilize an ad-hoc addition to speed up the procedure. The optimality of the path is not important, just find a path! An ad-hoc addition could be:

- Revert to RRT, since optimality is not necessary and it is too costly to check connections for neighbors too.
- Try to connect the new node of the tree to the goal at every iteration. (the author of this assignment used this).
- growing two RRTs instead of one, one from goal, one from initial configuration. Try to connect to the new node of one tree from the nearest of other at each iteration.
- Change your sampling strategy such that it is resolution based (like grid sampling).

You can refer to book by Choset et. al. for more ad-hoc algorithms. Comment diligently on your ad-hoc procedure and make it easier for the grader to understand.

4.5 Computational Complexity (5%)

The efficiency of your algorithm does not matter too much in this assignment, but you should keep an eye. The original algorithm reports $O(nlogn)$ for n nodes with optimized data structures for $Nearest$ and $Near$ such as k-d trees. Assuming the number of edges in the tree is asymptotically same with the nodes, you should be able to obtain $O(kn^2)$ complexity with k neighbors with simple python lists and objects.

4.6 Neatness of Code (5%)

This homework will be graded with whitebox method, therefore you should pay special attention to comments etc. within your code, especially if you are doing some complicated calculation, hard to figure out ROS magic, or doing something ad-hoc with your planner. Encapsulate your code well.

5 Other Regulations and Submission

- You must use Python 2.7 with ROS Melodic for this assignment.
- All of the transforms in this assignment respect the right-hand rule and all of the angles are in radians. Distance values are in meters.
- The helper code communicates with `move_group` node for collision checking and IK. However, Moveit! also provides API (in terms of ROS topics and other) to directly plan and go to a specific point (check out the tutorials for `move_commander` if you are interested). You may use these when you are debugging, but having them in your final submission is not allowed. You should implement your own planning algorithm.
- The IK computation returns stochastic results and sometimes picks very hard to reach/colliding configurations. To aid that problem, your algorithm will be run several times if it fails to find a goal. Other than this, please use seeds for random number generators to make your own part deterministic. The value of the seeds are not important. This is especially important in 2D code, since you have all the control there.
- You are not allowed to hard-code specific configurations to visit in your code (no, that will not count as an ad-hoc method).
- If you want to use any other Python module in yoor want to use another ROS package you must first ask publicly on the course forum before using it, we may or may not allow it.
- When you are done, put your `report.pdf`,`rrt_base.py`, `rrt_2d.py`, `cube_placer.py` in `pa2_eXXXXXXX.zip` and submit it to ODTUCLASS.
- **This is an individual assignment. Using any piece of code that is not your own is strictly forbidden and constitutes as cheating. This includes friends, previous homeworks, or the Internet. The violators will be punished according to the department regulations.**
- Please submit your questions publicly on the course forum if your question does not include parts of your solution. If your question does contain some of the solution, send an email to onem@ceng.metu.edu.tr.