Lab 3: Motion Planning with a 6-DOF Manipulator

(72 points total)

1 Main Implementation

Parts 1-2: See team Github for RRT implementation.

Parts 3-4: See team Github for trajectory screen captures: question-3.mp4, question-4-default.mp4, question-4-top.mp4, question-5-default.mp4, and question-5-top.mp4

Part 5: See team Github for `_get_random_sample_near_goal` implementation, and trajectory captures textttquestion-5-default.mp4 and `question-5-top.mp4`.

Observations: The time elapsed was usually greater than 3.0 seconds (ex. 3.4, 3.5) once we reduced epsilon to 0.2 and implemented `_get_random_sample_near_goal`. This is larger than the time elapsed without these changes, which was usually less than 1.0 second. We believe this change is due to a greater number of extensions of the RRT in order to create a node that must now be even closer to the goal node.
The final configuration of the trajectory ended up much closer to the soda can, though. Previously with epsilon of 1.0 the end-effector was often in the general area of the soda can but not very close to it. With epsilon of 0.2 the end-effector is moved to a pose in which it could feasibly grasp the soda can. This lines up with our expectations of how changing epsilon would affect final configuration state of the robot in relation to the goal (lower epsilon resulting in final configuration state which is closer to goal state).
Although the computation time increased, this trade-off is justified by the significant improvement in solution quality. While the previous configuration ($\epsilon = 1.0$) was faster, it rarely resulted in a successful grasp; the stricter tolerance ($\epsilon = 0.2$) ensures a much higher success rate for generating a viable grasp configuration, making the robot's path functionally useful rather than just approximate.

Part 6: It is not a good idea to call `_get_random_sample_near_goal` with probability 1.0 because the RRT would then likely only be extended in a single direction in the configuration space, namely the direction pointing from the start state to the goal state. While in some scenarios this directed growth could help the RRT reach the goal state very quickly, it makes the RRT very inflexible due to having only one possible growth direction.

An example of where this is problematic can be seen whenever there is an obstacle that prevents a direct path from start state to goal state. Image a horseshoe shaped obstacle (in configuration space) which surrounds this start state, having an opening only on the end that is away from the goal state. In order for the RRT to explore from start to goal, it would have to grow in the direction opposite from the goal state, and thus must sample a point in this opposite direction. Yet because we are sampling near the goal with probability 1.0, this will never happen, and the RRT will certainly get 'stuck' on this obstacle, never reaching the goal state.

In-person lab: Once you are confident in your simulation results, you are ready to run it on the real robot. This can be done on the lab workstations with -

```
$ python adarrt.py --real
```

Refer to Piazza for more instructions on scheduling time in the lab.

2 Additional Questions

As a very rough guideline, we anticipate that for most teams, the answers to each question below will be approximately one paragraph long (or about 4-5 bullet points). However, your answers may be shorter or longer if you believe it is necessary.

2.1 Resources Consulted

We implemented the core logic for the AdaRRT class by strictly following the pseudocode and terminology presented in the CSCI 545 lecture slides (specifically the definitions for q_{rand}, q_{near}, and the EXTEND operation). We utilized OpenAI's ChatGPT-5 as a debugging assistant to screen our Python implementation for logical errors. Additionally, we used ChatGPT-5 to verify our theoretical understanding of RRT behavior, confirming that the 'jerky' nature of the initial trajectory was an expected result of the randomized EXTEND steps before smoothing.

2.2 Team Contributions

Nitzan, Mansi, and Nathan set up the Docker container and started the coding. Nathan finished the implementation, wrote the report, and recorded the videos. Nima tested the code and reviewed the final videos and submission to make sure everything was correct. All of us worked together during the in-person lab session.
Nathan, Mansi, Nitzan, Nima: 25% contribution each