

# Lab 3: Motion Planning with a 6-DOF Manipulator

CSCI 545

**Team Name:** Markov Misfits

**Group Members:**

Gerardo Huerta  
Haley Miguel  
Rasika Ramanan  
Arianna Yuan

**(72 points total)**

## 1 Main Implementation

In this lab assignment, you will implement the RRT algorithm for a 6-DOF robotic manipulator. To perform the assignment, you will need to have installed the AIKIDO infrastructure. We provide for you the file **adarrt.py**, which contains several methods you will fill in. During development, you will run your code in simulation with -

```
$ python adarrt.py --sim
```

The python file contains the following classes and functions:

- **AdaRRT:** This is the main class. It initializes the start and goal node, the number of iterations, the step\_size  $\delta$  when extending a node in the tree, the desired goal precision  $\epsilon$  and the joint limits of the robot. It also includes information about the environment, which includes a table, a soda can that we want to grasp, the robot and a set of constraints that check for collisions. This implementation will be very similar to the RRT you built in HW4, with a few modifications discussed below.
- **AdaRRT.Node:** A Node object should contain a copy of the state, a pointer to the parent node in the tree, and a list of pointers to all its child nodes in the tree. A state in the provided code is a 6D np.array that contains the robot's configuration.

- `main`: this function specifies the start and goal configurations, sets up the RRT planner and computes a path. It then calls the AIKIDO function `compute_joint_space_path`, which generates a trajectory for the robot to follow.

Steps to complete the lab:

1. **Implement an RRT algorithm** by filling in the code in the provided file. For starting configuration  $q_S$  and goal configuration  $q_G$ , and parameters  $\epsilon$  and  $\delta$  use:

$$\begin{aligned} q_S &= [-1.5, 3.22, 1.23, -2.19, 1.8, 1.2] \\ q_G &= [-1.72, 4.44, 2.02, -2.04, 2.66, 1.39] \\ \delta &= 0.25 \\ \epsilon &= 1.0 \end{aligned}$$

Make sure you have `roscore` running before starting your RRT!

2. **Visualize the trajectory in rviz.** First, execute your AdaRRT implementation, but don't execute the trajectory. Then, open rviz from the command line using `rosrun rviz rviz`. In the bottom left module, click the "Add" button and navigate to the "By Topic" tab. You should see a `InteractiveMarkers` topic under `/dart_markers`. Add the topic before executing your trajectory generated by AdaRRT.

- (a) Accept the GitHub Classroom invitation  
(GitHub Classroom)
- (b) Download the docker image (we will not use the same docker images as in Lab1 and Lab2).

If use **ARM-64** architecture:

- If use Python2:  
Download and directly load this docker image:  
(Google Drive)  
`docker load -i cs545-lab3-melodic-arm64.tar.`
- If use Python3:  
Download and directly load this docker image:  
(Google Drive)  
`docker load -i cs545-lab3-noetic-arm64.tar.`

If use **X86-64** architecture:

- If use Python2:  
Download and directly load this docker image:  
(Google Drive)  
`docker load -i cs545-lab3-melodic-x86-64.tar.`

If build your own Docker image:

- Follow this instruction:  
(Google Drive)

- (c) Build the provided workspace (already in the docker image, you can skip this step): (Google Drive)

Unzip it to your home directory. Before running the assignment codes, make sure you have run source /ros\_ws/devel/setup.bash in the same terminal as your lab script. Remove build, devel, and log from your ros\_ws and rebuild the workspace by running

```
catkin\_\_make\_\_isolated -j4 # Use 4 parallel jobs
```

- (d) Create Docker Container

- (e) Run lab3 code

If use **ARM-64** architecture:

- Terminal 1: Run roscore

```
Run roscore
source /opt/ros/melodic/setup.bash
roscore
```

- Terminal 2: Run your script

Put your script in '/ros\_ws/src/lab3/adarrt.py'.

Run your script:

```
source ~/ros_ws/devel_isolated/libada/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/root/ros_ws/src/libada
python ~/ros_ws/src/lab3/adarrt.py --sim
```

- Terminal 3: Run Rviz visualization

```
source /opt/ros/melodic/setup.bash
source ~/ros_ws/devel_isolated/libada/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/root/ros_ws/src/libada
rosrun rviz rviz
```

If use **X86-64** architecture:

- Terminal 1: Run roscore

```
Run roscore
source ~/.bashrc
roscore
```

- Terminal 2: Run your script

Put your script in ~/lab3/adarrt.py.

Run your script:

```
source ~/.bashrc
python ~/ros_ws/src/lab3/adarrt.py --sim
```

- Terminal 3: Run Rviz visualization

```
source ~/.bashrc
rosrun rviz rviz
```

3. **(40 points)** Use an off-shelf screen capture software (e.g., <https://itsfoss.com/kazam-screen-recorder/>) to **record a video** of the trajectory. Include the video in the root of your GitHub repo as a file named question-3.mp4.
4. **(10 points)** The RRT trajectory is typically jerky. Typical planners use shortcircuiting algorithms to make the path smoother. Replace the function `ada.compute_joint_space_path` with `ada.compute_smooth_joint_space_path`. Capture the new trajectory with two videos – one showing the default isometric view, and another showing the top view. Include the videos in the root of your GitHub repo as files named `question-4-default.mp4` and `question-4-top.mp4`.
5. **(10 points)** The goal precision  $\epsilon$  of 1.0 in the previous question is too large. In order to avoid collisions, we need to improve the precision. However, this dramatically increases the time to compute a solution. To improve computation, **add a method** `_get_random_sample_near_goal` that generates a sample around the goal within a distance of 0.05 along each axis of the search space. Then, change the `build` method so that it calls `_get_random_sample_near_goal` with probability 0.2 and `_get_random_sample` with probability 0.8. Reduce  $\epsilon$  to 0.2.

Write down your observations in the PDF file. Also capture the new trajectory with two videos – one showing the default isometric view, and another showing the top view. Include the videos in the root of your GitHub repo as files named `question-5-default.mp4` and `question-5-top.mp4`.

Although an increase in computation is expected, by adding a 20% goal biasing, we observed that the computation time was kept reasonable. For our `question-5-default.mp4`, the elapsed time was 4.078 seconds, whereas in `question-4-default.mp4`, the elapsed time was 3.618 seconds, which was not too much of a difference.

With  $\epsilon$  set to 0.2, the RRT algorithm is guaranteed to find a path that gets much closer to the final goal configuration before it finishes.

In addition, as seen in `question-5-top.mp4`, the path is still very smooth and direct. The goal-biasing did not reintroduce the "jerky" motion from the initial RRT implementation. The path clearly plans a motion around the table to get to the soda can, showing that the RRT is still capable of exploring to avoid obstacles while also being biased toward the goal.

6. **(10 points)** Explain why it is not a good idea to call `_get_random_sample_near_goal` with probability 1.0. Also present an example where this could be problematic.

Write your answer in the PDF.

It's not a good idea to call `_get_random_sample_near_goal` with too high of a probability, because this over-emphasizes the robot's trajectory towards its goal state at the detriment of navigating a map with any type of obstacle. The RTT algorithm may not explore states that are "too far" away from the goal, and thus, if there's a wall or colliding obstacle very close to the goal state, the RRT cannot "see" its way around the obstacle by branching out new nodes in places that aren't pinholed near the goal. An example of where this could be problematic is a scenario where the goal is located inside a box with a single path available through a narrow passageway, where the entrance is located near the robot.

**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

**Question: (1 points)** Please describe which resources you used while working on the assignment. You do not need to cite anything directly part of the class (e.g., a lecture, the CSCI 545 course staff, or the readings from a particular lecture). Some examples of things that could be applicable to cite here are: (1) did you get help from a classmate *not* part of your lab team; (2) did you use resources like Wikipedia, StackExchange, or Google Bard in any capacity; (3) did you use someone's code (again, for someone *not* part of your lab team)? When you write your answers, explain not only the resources you used but HOW you used them. If you believe your team did not use anything worth citing, *you must still state that in your answer* to get full credit.

Our team's work was primarily based on the provided course materials and lectures. For specific technical implementation details, we consulted the official online documentation for Python, NumPy, and the Robot Operating System (ROS) to understand the usage of certain functions and libraries. We did not use any code from outside sources or receive assistance from individuals outside of our team.

## 2.2 Team Contributions

**Question:** (1 points) Please describe below the contributions for each team member to the overall lab. *Furthermore, state a (rough) percentage contribution for each member.* For example, in a team of 4, did each team member contribute roughly 25% to the overall effort for the project?

Contributions:

- **Gerardo Huerta, 25%**: Wrote functions `build`, `_get_random_sample`, and `_get_nearest_neighbor` for base RRT code.
- **Haley Miguel, 25%**: Implemented the path smoothing for Question 4. Recorded the videos for `question-3.mp4`, `question-4-default.mp4`, and `question-4-top.mp4`. Answered Question 6.
- **Rasika Ramanan, 25%**: Implemented the goal-biasing logic for Question 5, including the `_get_random_sample_near_goal` functions and modifications to the `build` loop.
- **Arianna Yuan, 25%**: Organized and distributed the tasks, wrote functions `_extend_sample`, `_check_for_completion`, and `_trace_path_from_start` for base RRT code. Merged smoothing and goal-biasing branches, ran final merged code, and recorded the final `question-5` videos. Answered Question 5.