

ENEE 467: Robotics Project Laboratory

Collision-Free Kinematic Motion Planning in ROS 2

Lab 5 — Part I

Learning Objectives

This lab provides a hands-on introduction to core motion planning concepts using minimal Python-only kernels and custom ROS 2 packages. Students familiar with the higher-level MoveIt framework may find this approach unorthodox, but it is intentional. In Part II, we will revisit these concepts using MoveIt and `ros2_control`, linking the low-level methods developed here to standard planning tools. After completing Part I of this lab, students should be able to:

- Write ROS 2 nodes implementing collision-free joint-space sampling-based path planning algorithms for kinematic chains
- Write ROS 2 nodes implementing polynomial trajectory generation algorithms for via-point paths in configuration space
- Understand how collision detection, C-space sampling, graph search, and heuristic C-space tree exploration integrate in a high-performance planning stack.

Motion planning computes collision-free, time-parameterized trajectories in joint or Cartesian space that move a robot from an initial to a goal configuration while respecting obstacles, joint limits, and dynamics. Unlike pure kinematics, it explicitly considers the robot's workspace and dynamic constraints and is thus critical for robots operating in semi-controlled or unstructured settings. Together, the two parts of this lab aim to equip you with the skills and tools to use modern motion planning frameworks (e.g., *MoveIt*) and to develop your own ROS 2-compliant motion planning programs.

Related Reading

- S. M. LaValle, “*Planning Algorithms*,” Cambridge Univ. Press, 2006. Available: [Free PDF](#).
- K. Lynch and F. Park, “*Modern Robotics*, Cambridge University Press, 2nd ed., 2019.
- M. Spong et. al., “*Robot Modeling and Control*”, John Wiley & Sons, Inc., 2006.
- G. Russi, “*Robotics Part 32 — Trajectory Generation*,” Available: [Robotics Unveiled](#).
- S. Castro, “How Do Robot Manipulators Move?” Available: [Robotic Sea Bass](#).

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab [Questions](#) (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab [Exercises](#) at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

| | |
|-------------|----|
| Question 1. | 5 |
| Question 2. | 7 |
| Question 3. | 8 |
| Question 4. | 9 |
| Question 5. | 12 |
| Question 6. | 15 |
| Question 7. | 19 |
| Question 8. | 19 |


List of Exercises


| | |
|---|----|
| Exercise 1. Efficient Path Generation via Informed Sampling (25 pts.) | 16 |
| Exercise 2. Via-Point-Path Trajectory Generator (25 pts.) | 20 |

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

Question 0. A question for you.

Boxes with a  symbol (like the one below) are extra theoretical details provided for your enlightenment or just general information about the lab. We recommend that you read them during the lab, but they may be skipped during the lab and revisited at a later time (e.g., while working on the lab report):

 Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text Python commands or snippets will be set in blue teletype font, e.g., `a = [0, 0, 0]`, while Python type hints (and occasionally XML tags) will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and [cyan text](#) highlights a link to an external online reference.

1 Lab Procedure

1.1 Prerequisites

Basic knowledge of URDF-based robot modeling and ROS 2 is assumed for this lab. Rigid-body simulation, coordinate frame transforms, and forward/inverse kinematics should all be familiar to students. It is beneficial but not necessary to be exposed to motion planning topics such as sampling-based search and polynomial trajectory generation.

1.2 Getting this Lab's Code

To begin, in a new terminal session, navigate to the `~/Labs` directory and then clone the lab repo:

```
cd ~/Labs
git clone https://github.com/ENEE467-F2025/lab-5.git
cd lab-5/
```

Verify that the lab-5 image is built on your system

```
docker image ls
```

Make sure that you see the lab-5 image in the output:

```
lab-5-image    latest    <image_id>    <N> days ago  <image_size>GB
```

From within the container, build and source your workspace:

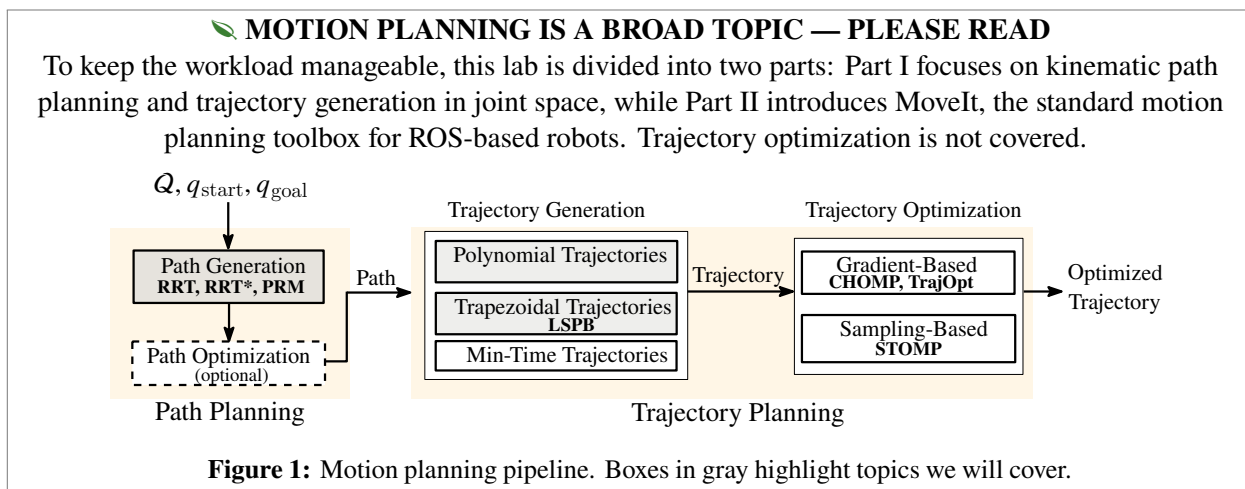
```
cd ~/ros2_ws/
colcon build --symlink-install
source install/setup.bash
```

Then test your build by running the following command from an active container:

```
cd ~/ros2_ws/src && python3 test_docker.py
```

This should print the following output to the terminal (if the message doesn't appear, stop and contact your TA, otherwise proceed with the lab procedure):

```
All packages for Lab 5 found. Docker setup is correct.
```

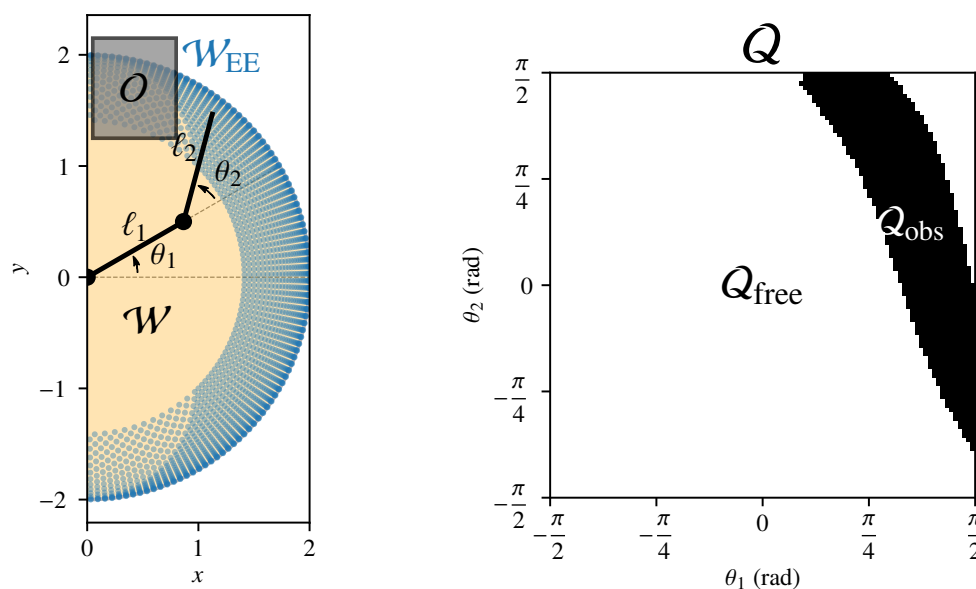


2 Motion Planning Foundations

Motion planning often splits into two stages: **path planning** (time-free configuration or pose sequence) and **trajectory planning** (assigning timing, velocities, and possibly accelerations). The resulting trajectory may be jerky or violate limits, so a final **trajectory optimization** step may usually be required to smooth or adjust it.

2.1 Configuration Space Obstacles

To plan collision-free motions, it is usually convenient to express workspace obstacles in terms of the robot's configuration using configuration-space obstacles (C-obstacle), which is any configuration in which the robot intersects an obstacle (see Figure 2b). C-obstacles partition the configuration space or C-space, Q , into a set of colliding configurations, Q_{obs} , and the collision-free space Q_{free} .



(a) Workspace of the planar 2R robot arm

(b) Approximate C-space obstacle representation

Figure 2: Example workspace (W) and C-space (Q) of a planar 2R robot arm.

Running Example (Section 2) — 2R Robot

The 2R robot in Figure 1 has link lengths $\ell_1 = \ell_2 = 1$ m and joint limits $(-\frac{\pi}{2}, \frac{\pi}{2})$. It is shown at the configuration $q = [\frac{\pi}{6}, \frac{\pi}{4}]^\top$. The rectangular workspace obstacle O has its bottom-left corner at $(0.05, 1.25)$ and area $0.75 \text{ m} \times 0.9 \text{ m}$ (width \times height). Because the robot's workspace W intersects O , the corresponding configuration-space obstacle set Q_{obs} is non-empty.

To estimate the set of C-space obstacles, we need three modules:

- (1). **Uniform sampling** of the C-space (necessary since it contains infinitely many configurations)
- (2). Computing all points on the robot via **forward kinematics**, for each sample
- (3). **Checking for collisions** between the robot and obstacles.

Having covered forward kinematics in Lab 2, we'll focus on (1) and (3) in what follows.

2.1.1 Random Sampling in C-Space

A common and simple approach to generating random C-space samples entails sampling from a uniform distribution, where each joint variable is equally likely to assume any value within its limits (see [Continuous Uniform Distribution](#)). Thus, for an n-degrees-of-freedom robot, a sample configuration is an array of length n whose components are independently drawn from intervals contained within the joint bounds. NumPy facilitates uniform sampling via the `numpy.random.uniform` function. Let's generate random configurations for a 2R arm with limits $(-\frac{\pi}{2}, \frac{\pi}{2})$. Create a file `random_sampling.py` in the `~/ros2_ws/src/concepts` directory, and add the following code block to it:

```
import numpy as np
n = 2 # define the number of DoFs
joint_limits = [-np.pi/2, np.pi/2] # joint limits

def sample_config(n=n, joint_limits=joint_limits):
    """Sample a random configuration within joint limits."""
    q_samp_cand = np.random.uniform(joint_limits[0], joint_limits[1], size=n)
    try:
        assert (joint_limits[0] < q_samp_cand[0] < joint_limits[1]
                and joint_limits[0] < q_samp_cand[1] < joint_limits[1]
                ), (f"Sample config out of bounds: "
                   f"[{joint_limits[0]:.2f}, {joint_limits[1]:.2f}]!")
    except AssertionError as e: # Re-sample if out of bounds
        print(f"\033[91m{e}\033[0m\nResampling...")
        return sample_config(n, joint_limits)
    return q_samp_cand

if __name__ == "__main__":
    for _ in range(5):
        q_samp = sample_config(n, joint_limits)
        print(f"Sampled config: {np.round(q_samp, 4)}")
```

Run the script and verify that it prints five joint configurations within the specified joint limits.

Question 1. Why is it important to introduce additional guards when generating random joint samples, e.g., the assertion logic inside the try-except block in `random_sampling.py`?

Hint: Recall that the joint limits we're enforcing define an open interval.

2.1.2 Collision Detection

The C-space sampler may not always return samples from the free C-space, so C-space sampling is generally followed by a collision-checking step. Collision detection is usually framed as a distance problem, but in the basic case, we typically only ask a [binary question](#): does any part of the robot intersect with the obstacle or not? We'll focus on the planar case for simplicity. Spatial collision checking is similar but more involved. Here, each link is a line segment in the XY-plane, and obstacles are simple shapes like circles or polygons (see [Figure 3](#)).

The `coll_det.py` file in the `concepts` folder implements collision detection for rectangular and circular obstacles. Rectangular collision checks return only *binary* results, while circular checks also provide the *minimum distance* to the obstacle. Let's try a few examples.

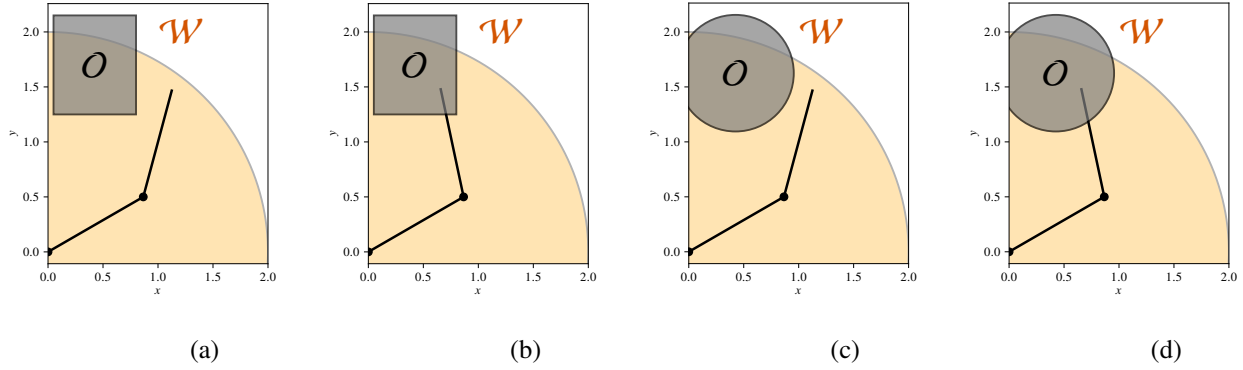


Figure 3: Collision detection for a 2R robot with rectangular and circular obstacles, showing non-colliding ($q = [\frac{\pi}{6}, \frac{\pi}{4}]^T$) and colliding ($q = [\frac{\pi}{6}, \frac{2\pi}{5}]^T$) cases.

Change directory to the `~/ros2_ws/src/concepts` folder:

```
cd ~/ros2_ws/src/concepts
```

Then try querying the collision detector for the configuration and obstacle in Figure 3 (a) by passing the appropriate `np.pi` expressions to the `config` parameter and "rect" to the `obstacle_type` parameter using the format `--<parameter_name> <parameter_value>`:

```
python3 coll_det.py --link_lengths 1.0 1.0 --config np.pi/6 np.pi/4 \
--obstacle_type "rect" --rect_params 0.05 1.25 0.75 0.9 --verbose true
```

Verify that the above command yields the following output and a plot of the robot's workspace:

```
No collision with rectangular obstacle at q=[0.5236 0.7854]
```

Next, let's try checking for circular obstacles by setting the `obstacle_type` parameter to "circle" and specifying the center (first two values) and radius (third value) of the circular obstacle using the `circ_params` parameter. Retaining the parameters provided in the 2R robot example, to check whether the robot at the configuration $q = [\frac{\pi}{6}, \frac{\pi}{4}]^T$ collides with a circular obstacle centered at (0.425, 1.625) with radius 0.5303 m (see Figure 3 (b)), run:

```
python3 coll_det.py --link_lengths 1.0 1.0 --config np.pi/6 np.pi/4 \
--obstacle_type "circle" --circ_params 0.425 1.625 0.5303 --verbose true
```

In this case, we also obtain the minimum obstacle distance along with collision information:

```
No collision with circular obstacle at q=[0.5236 0.7854]
Min. dist to obstacle: 0.7172 m
```

For the circular obstacle, a colliding configuration such as $[\frac{\pi}{6}, \frac{2\pi}{5}]^T$ can be checked using the following command (note the angle format passed to the `config` parameter):

```
python3 coll_det.py --link_lengths 1.0 1.0 --config np.pi/6 2*np.pi/5 \
--obstacle_type "circle" --circ_params 0.425 1.625 0.5303 --verbose true
```

Verify that this yields a minimum distance smaller than the obstacle radius (0.5303 m):

```
Collision with circular obstacle at q=[0.5236 1.2566]!  
Min. dist to obstacle: 0.2755 m
```

Lastly, try passing a configuration that exceeds the joint limits, e.g., $[\frac{\pi}{6}, \frac{\pi}{2}]^T$:

```
python3 coll_det.py --link_lengths 1.0 1.0 --config np.pi/6 np.pi/2 \  
--obstacle_type "circle" --circ_params 0.425 1.625 0.5303 --verbose true
```

Verify that the script raises the following `AssertionError` in this case, since, by definition, the boundaries of the C-space are defined by the joint limits:

```
Joint angles out of bounds: [-1.57, 1.57]!
```

Note that the collision-free C-space may be further restricted by task constraints. Try to imagine a 6-DoF robot arm carrying an object rigidly attached to its end-effector that must remain upright throughout motion. In this case, any configuration that causes the end-effector to tilt (even if it is collision-free) is not task-feasible and should be discarded during sampling.

Question 2.

- (a). Run the collision detector (`coll_det.py`) for the following two configurations, assuming a circular obstacle with the default parameters used in the examples:

(i). $q = [\frac{\pi}{6}, \frac{\pi}{4}]^T$ (ii). $q = [\frac{5\pi}{12}, -\frac{\pi}{4}]^T$

Report whether a collision is detected and the minimum obstacle distance in each case.

Note: To pass in the fractional and negative arguments of (ii). above, specify `config` as:

```
--config 5*np.pi/12 " -np.pi/4"
```

- (b). For the 2R robot in [Figure 3](#), if two configurations q^A and q^B yield the same collision-free end-effector position $x_{ee} \notin \mathcal{O}$, but link 2 is in collision for q^B while both links are free for q^A , should both configurations be in \mathcal{Q}_{obs} ? Provide a yes/no answer with justification, and optionally, a conceptual plot.

2.1.3 Estimating the Set of C-Obstacles

Now that we have a C-space sampler and a collision detector we can query to determine if a sampled configuration is in collision, we can estimate the set of C-obstacles. We've provided a working implementation for determining this set in the file `config_space.py` within the `concepts` folder that's specialized to a planar 2R robot. The script is parameterized, allowing you to explore different link lengths, joint limits, and obstacle configurations. For instance, to visualize the C-obstacles for the setting in [Figure 2b](#), with the parameters specified in the [2R robot example](#), run the command:

```
python3 config_space.py --link_lengths 1.0 1.0 --thetas np.pi/6 np.pi/4 \  
--joint_limits " -np.pi/2" np.pi/2 --rect_params 0.05 1.25 0.75 0.9
```

The `thetas` argument sets the initial configuration of the 2R robot, specified as fractional factors of π and passed using the `np.pi` format in the Bash command, e.g., for the configuration $[\frac{\pi}{6}, \frac{\pi}{3}]^T$, you would pass `--thetas np.pi/6 np.pi/3`; for $[-\frac{\pi}{3}, \frac{\pi}{4}]^T$, pass `--thetas " -np.pi/3" np.pi/4`. The `link_lengths` specifies the arm's link lengths, and `rect_params` specifies the bottom-left xy

position (first two values), and the width and height of the rectangular obstacle in meters (last two values). Notice the double quotation marks around the negative lower limit in the `joint_limits` argument and the space before the minus sign, and carefully type out (or copy-paste) the commands.

Question 3.

- (a). Consider the setting in Figure 2a. With the 2R robot's link lengths and joint limits fixed, run `config_space.py` for each scenario below by adjusting the relevant arguments. In your report, comment on how Q_{obs} changes, and include plots showing Q (see Figure 2b):

- (i). The bottom-left corner of O stays fixed; width and height increased by 50%
- (ii). The area of O stays the same; bottom-left corner moved to $(0.05, -2.15)$.

Note: The script automatically saves a plot named `cspace_obs_rect.png` in the current working directory, so you can copy the file directly to your report draft after each run.

- (b). Suppose we are instead given a planar 2R robot arm with the following link lengths: $\ell_1 = 0.5 \text{ m}$, $\ell_2 = 0.45 \text{ m}$. If O 's parameters remain unchanged from Figure 2a, is it always the case that Q_{obs} is the empty set? You can confirm your intuition by plotting a few configurations within the arm's joint limits $(-\frac{\pi}{2}, \frac{\pi}{2})$. You only need to provide a yes/no answer and a small comment. No plots are required.

2.2 Configuration Grids & A* Search

Because obstacle geometry can be complex, the collision-free C-space is often hard to characterize exactly. A common workaround is to approximate it with a *discretized grid* of M^n cells (C-grid), where each cell represents a joint configuration (see Figure 4 for $M = 100$).

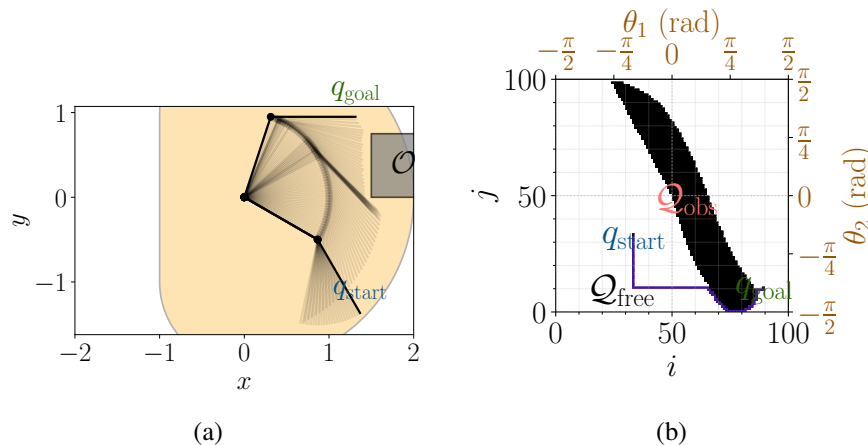


Figure 4: (a) Workspace with path. (b) C-grid with path in purple.


Once the C-space has been discretized into a grid, finding a path, i.e., a set of collision-free grid cells connecting two collision-free cells on the grid, is simply a *search problem*, and many planners often employ the [A* search algorithm](#), which finds the minimum-cost path between defined indices. A common cost function or heuristic is the [Manhattan distance](#) to the goal cell along collision-free cells. Each grid cell is typically taken to be a square of length 1.


The file `astar_search.py` within the `concepts` folder implements A* on a binary occupancy C-grid. It is parametrized, so we can test it on various start–goal pairs. For the first example, let’s try performing A* search to find a path between grid cells at the cell indices (33, 33) and (89, 10), corresponding to configurations $[-\frac{\pi}{6}, -\frac{\pi}{6}]^\top$ and $[\frac{2\pi}{5}, -\frac{2\pi}{5}]^\top$ in C-space (see Figure 4) by passing the cell indices to the `start_idx`s and `goal_idx`s parameters using the following command (note that the default joint limits set within the script are $(-\frac{\pi}{2}, \frac{\pi}{2})$ and not the canonical $(-\pi, \pi)$):

```
python3 astar_search.py --start_idxes 33 33 --goal_idxes 89 10
```

Verify that this yields the intermediate cells, configurations, as well as the path cost, i.e., the sum of the side lengths of all grid cells contained in the path (except the start cell).

```
The route found covers 100 grid cells.
Cell: (i, j) = (33, 33) -> (theta1, theta2) = (-0.52, -0.52) rad
...
Cell: (i, j) = (89, 10) -> (theta1, theta2) = (1.25, -1.25) rad
Minimum cost from start to goal: 99
```

You should also see a plot (similar to Figure 4 (a)) that shows the generated path. By default, the generated path avoids obstacles, but this is only a safety add-on to A*. See the  box below.

 Collision checking is not part of standard A* search, as the default assumption is that the cells in the grid to be searched are already collision-free. However, it can be incorporated as an extension when applying A* to path planning in Cartesian or joint space. In such settings, the cost function must include a collision cost, e.g., infinity.

As a second instance, pass a colliding goal cell (e.g., $(48, 68) \approx [-\frac{\pi}{63}, -\frac{10\pi}{53}]^\top$) to the script using:

```
python3 astar_search.py --start_idxes 33 33 --goal_idxes 48 68
```

Observe that A* terminates with “Goal cell (48, 68) is in collision! No path found!”. However, setting the script’s `check_collision` parameter to `False` allows the search to proceed:

```
python3 astar_search.py --start_idxes 33 33 --goal_idxes 48 68 \
--check_collision False
```

Now A* “succeeds” yielding an invalid path with a finite cost of 50 units. **Note:** Disabling collision checking is not advised in practice, but it can be useful in simulation for debugging or task setup.

Question 4. So far, we’ve assumed that the C-grid supplied to the A* search routine contains only indices pointing to reachable free C-space configurations. However, the goal cell might be *unreachable*, i.e., collision-free but isolated from the start cell by obstacles. Run the following command to simulate this scenario:

```
python3 astar_search.py --start_idxes 10 89 --goal_idxes 99 10 \
--rect_params 0.25 0.0 1.75 0.75
```

Why does A* fail even when free space exists? Provide a brief explanation in your report; no plots are needed. **Hint:** Recall that we’ve assumed joint limits of $(-\frac{\pi}{2}, \frac{\pi}{2})$.

2.3 C-Space Random Trees

Computing a full C-grid is expensive for large dimensions ($O(M^n)$ in time and memory). Instead, many planners incrementally grow a C-space tree of random collision-free configurations rooted at the start (see Figure 5) configuration by a combined process of random sampling, collision-checking, and tree search. Each node is a configuration $q \in Q$, and edges connect nearby nodes within a fixed C-space radius. The tree generation routine proceeds in discrete steps, expanding the tree with new collision-free nodes until the tree reaches the goal region, or until a maximum number of iterations is reached. Let's grow a C-space tree to make things concrete.

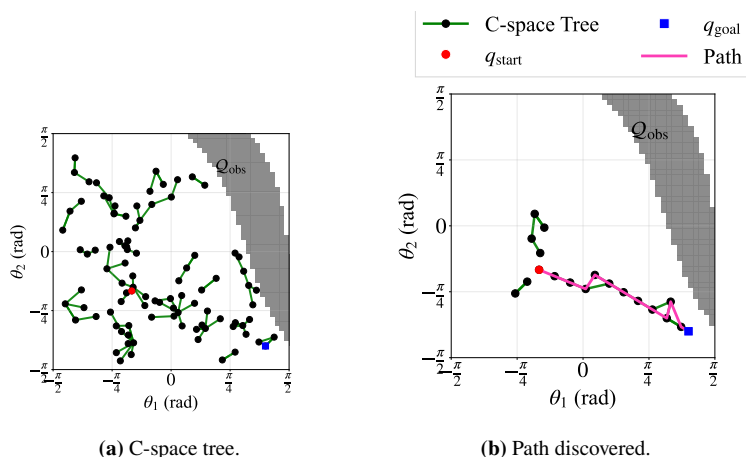


Figure 5: C-space tree & path generated via incremental sampling and A* search.

The `cspace_tree.py` script implements a random C-space tree generator for the 2R robot. Generate a tree rooted at the two collision-free configurations from Section 2.2 using the command below, where `max_iter` sets the iteration limit:

```
python3 cspace_tree.py --qstart " -np.pi/6" " -np.pi/6" --qgoal 2*np.pi/5 \
" -2*np.pi/5" --max_iter 150
```

You should see the following output (truncated for brevity) and a plot of the C-space tree:

```
--- Tree Metadata ---
Total nodes in tree: 96
Maximum tree depth: 17
```

Note that the maximum tree depth is 17, meaning the deepest node in the tree is 17 edges away from the root. Since path length counts nodes rather than edges, this corresponds to a path of 18 nodes from the root to that leaf. Path planning requires tracking path quality, typically assessed by the sum of node costs along the path. A common cost function is the Euclidean distance (or 2-norm) between two nodes in the tree

$$\|q_1 - q_2\|_2 = \sqrt{\sum_{i=1}^n (q_{1,i} - q_{2,i})^2}, \quad (q_{\star,i} \text{ is the } i^{\text{th}} \text{ joint variable of } q_{\star}), \quad (1)$$

a quantity easily computed using NumPy's `linalg.norm` function. Create a file `tree_cost.py` under the `concepts` folder and add the following code block to it:

```
import numpy as np
n = 2
joint_limits = [-np.pi/2, np.pi/2] # joint limits

q_1 = np.random.uniform(joint_limits[0], joint_limits[1], size=n)
q_2 = np.random.uniform(joint_limits[0], joint_limits[1], size=n)

cost = np.linalg.norm([q_1, q_2])
print("Cost:", cost)
```

Run the file a few times to see how the cost variable changes with each C-space sample. To display each node's cost in the C-space tree, run the generator with the `--show_costs` flag:

```
python3 cspace_tree.py --qstart " -np.pi/6" " -np.pi/6" --qgoal 2*np.pi/5 \
" -2*np.pi/5" --max_iter 150 --show_costs
```

Observe that this prints a version of the tree metadata that includes the total tree cost and average node cost; the plot now also shows the cost for each node:

```
Total tree cost: 178.485
Average node cost: 1.859
```

For a final check, try generating a C-space tree rooted at the colliding configuration, $q_{\text{start}} = [\frac{\pi}{4}, \frac{\pi}{4}]^T$:

```
python3 cspace_tree.py --qstart np.pi/4 np.pi/4 --qgoal " -np.pi/6" " -np.pi/6" \
--max_iter 150
```

Observe that the tree generation process fails, and the script returns the following output:

```
Start configuration is in collision!
```

This behavior is expected, since, by definition, both the start and goal configurations must be collision-free for tree construction or path generation to even make sense.

2.4 Generating Paths by Exploring Random C-Space Trees

Once a C-space tree is built, A* search (Section 2.2) can retrieve the lowest-cost path using stored node costs as the heuristic. However, building the *entire* tree before searching is inefficient. Sampling-based planners instead grow a single tree *incrementally* by sampling a random configuration and connecting it to the *nearest* node, adding only cost-effective nodes (unlike grid methods, which enumerate all cells). Among the class of sampling-based path planners, we'll focus on the optimal variant of the Rapidly-Exploring Random Trees algorithm specialized to the 2R robot (see [RRT on Wikipedia](#) and the [original paper](#)), since it is also implemented in the Open Motion Planning Library (OMPL), the planner backend for MoveIt.

The `cspace_tree.py` script already supports path generation via the `--generate_path` flag. Try running it on the start and goal pair from the Section 2.3:

```
python3 cspace_tree.py --qstart " -np.pi/6" " -np.pi/6" --qgoal 2*np.pi/5 \
" -2*np.pi/5" --max_iter 150 --generate_path
```

Verify that this yields the tree and path metadata, and the configurations along the path (waypoints):

```
--- Tree Metadata ---  
Total nodes in tree: 18  
Maximum tree depth: 13
```

Check the tree, node, and path costs by appending the `show_costs` flag to the previous command:

```
python3 cspace_tree.py --qstart " -np.pi/6" " -np.pi/6" --qgoal 2*np.pi/5 \  
" -2*np.pi/5" --max_iter 150 --generate_path --show_costs
```

Observe that the tree and average node costs (22.438 and 1.247, respectively) are lower than those of the previous example. The tree also has a maximum depth of 13.

Question 5.

- Why are the tree cost and maximum depth smaller when `generate_path` is enabled than when only the C-space tree is built? (**Hint:** Path planning searches for a minimum-cost route to the goal, so it only explores a subset of the full tree.)
- Run the path generator with increasing values of `max_iter`. Test for `max_iter` set to 10, 20, and 30 using the following command (replace `<num>` with the test number):

```
python3 cspace_tree.py --qstart " -5*np.pi/12" " -np.pi/6" \  
--qgoal 2*np.pi/5 " -2*np.pi/5" --generate_path --max_iter <num>
```

How does the planner's likelihood of finding a path change as you increase this parameter? In your report, provide a simple statement of your observation; no plots are required.

Completeness and Optimality

Path planners are assessed by *completeness* and *optimality*. A complete planner always finds a path if one exists; an optimal planner returns the least-cost path. Randomized planners like RRT are *probabilistically complete* (solution probability $\rightarrow 1$ over time), while RRT* is *asymptotically optimal*, converging to the least-cost path as samples increase.

3 Path Planning for Spatial Kinematic Chains in ROS 2

So far, you have explored path planning through C-space grids, uniform sampling, and collision checking for a 2R robot. We now transition to ROS 2, applying the same ideas to a 3R robot using RRT*. Rather than covering RRT* in full, we will follow an implementation and focus on manageable modifications such as goal-biased sampling and basic parameter tuning.

Running Example (Section 3 & Exercise 1) — 3R Robot

The 3R (or RRR) spatial open chain in Figure 6 has configuration $q = [q_1, q_2, q_3]^T = [1.51, 0.61, 1.41]^T$ radians. We take $q \equiv \theta = [\theta_1, \theta_2, \theta_3]^T$.

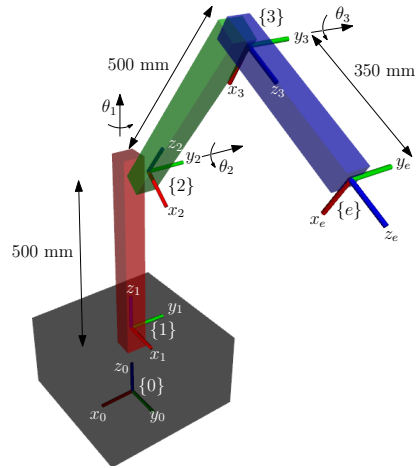


Figure 6: 3R spatial open chain with link frames and rotation axes.

3.1 Test the RRT* Joint-Space ROS 2 Path Planner

From a running Docker container, start up a new `tmux` session. Then in the `tmux` terminal, run the following commands sequentially

```
cd ~/ros2_ws/
colcon build --symlink-install
source install/setup.bash
```

Verify that the package is installed by listing its executables using: `ros2 pkg executables robot_3r_planners`, which should print the following: `informed_rrts_planner`, `robot_geom_publisher`, `sampling_based_planner`, and `simple_obstacle_publisher`. Then split your `tmux` session using the `Ctrl + A` and `B` (or `Ctrl + A` and `V`) macro to create a second pane. In one `tmux` pane, start the Gazebo, control, and planning ancillary nodes using:

```
ros2 launch robot_3r_bringup robot_3r_bringup.launch.py \
launch_planning_nodes:=true run_headless:=true is_dense:=false
```

This should bring up an RViz window showing the robot in an environment with two orange cuboidal obstacles. Note that the `run_headless` parameter controls Gazebo's launch mode: `run_headless:=false` starts the GUI (client mode), while `run_headless:=true` runs it in the background (server mode). In the second `tmux` pane, run the sampling-based path planning node, setting the goal parameter (`goal_config`) to the desired goal configuration of the 3R robot:

```
ros2 run robot_3r_planners sampling_based_planner --ros-args -p \
goal_config="[1.51, 0.61, 1.41]" -p check_collision:=True
```

Monitor the active `tmux` session for planner logs. If a path is found for the given goal, the node prints the waypoints (truncated here for clarity) and stops searching, though it remains running:

```
Found a path on attempt 1!
Waypoint: [0. 0. 0.]...
Waypoint: [1.51 0.61 1.41]
Num waypoints: 20
Path cost: 3.827
```

You should also see RViz CUBE markers that show the start configuration (in purple), goal configuration (in blue), as well as the intermediate configurations (in black) along the computed path. By inspecting the waypoints, observe that the robot starts from the zero configuration and that the terminal configuration is exactly the `goal_config` parameter we specified.

We can visualize the path the 3R robot traces in Cartesian space by setting the `show_ee_path` parameter to `True` at runtime (**Note:** you will need to terminate both the RViz and planner nodes and rerun them in the same order as before, making sure to pass the new parameter to the planner):

```
ros2 run robot_3r_planners sampling_based_planner --ros-args -p \
goal_config:="[1.51, 0.61, 1.41]" -p check_collision:=True -p \
show_ee_path:=True
```

Now, once the planner finds a path, you should see the Cartesian poses of the arm, visualized as a green LINE_STRIP RViz marker (see [RViz Markers](#)) in addition to the intermediate arm configurations. Terminate the planner node by pressing `Ctrl + C`. Leave the RViz node running. Now rerun the planning node with a deliberately invalid (colliding) goal:

```
ros2 run robot_3r_planners sampling_based_planner --ros-args -p \
goal_config:="[1.38, 1.91, 0.51]" -p check_collision:=True
```

Verify that the following output is printed to shell indicating that the planner could not find a valid path to the goal configuration. In fact, no path will be valid, since by definition, the goal configuration is part of the path and must be collision-free to be valid.

```
Goal configuration [1.38 1.91 0.51] is in collision!
Could not find a path. Goal configuration in collision.
Planning failed on single attempt. Stopping further planning.
```

Having sampled the ROS 2 planner, let's try to make some upgrades to make it more efficient.

3.2 Goal-Biased Random C-Space Sampling

The planner instance from [Section 3.1](#) simply sampled uniformly from the C-space at each planning iteration. However, this is naive and inefficient if we want to reach the goal in a timely fashion, since, intuitively, we'll require more iterations to sufficiently explore the C-space for a distant goal. A more efficient approach, however, is to bias the sampling process to generate samples from a region around the goal with some pre-computed probability, $0 < \epsilon < 1$ ($\epsilon = 1$ is not recommended):

$$q_{\text{samp}} = \begin{cases} [q_{i,\text{samp}} \mid i = 1, 2, \dots, n]^\top, & q_{i,\text{samp}} \sim \mathcal{U}_{[q_{i,\text{min}}, q_{i,\text{max}}]}, \text{ with prob. } 1 - \epsilon \\ q, & \text{where } \|q - q_{\text{goal}}\|_2 \leq \delta, \text{ with prob. } \epsilon \end{cases} \quad (2)$$

(δ is a very small +ve number < 1).

This means that with goal-biased sampling, the planner generates random C-space samples $100(1 - \epsilon)\%$ of the time and goal-directed samples $\epsilon\%$ of the time to accelerate path finding. Test this using our custom ROS 2 path planner by setting $\epsilon = 0.1, 0.3, 0.5$, and 0.7 via `rrts_goal_sample_rate` and enabling `use_goal_biased_sampling:=True`:

```
ros2 run robot_3r_planners sampling_based_planner --ros-args -p \
goal_config:="[1.51, 0.61, 1.41]" -p check_collision:=True -p \
use_goal_biased_sampling:=True -p rrts_goal_sample_rate:=0.1 # <- change this
```

Restart only the planning node each time (if packages were built with `-symlink-install`). Notice that the planner finds a path, with decreasing waypoints: 18, 14, 13, 12 for the respective ϵ values.

Question 6. What happens when you sample from the goal region all the time, i.e., when $\epsilon = 1.0$? Try for the goal configuration $[1.01, 1.1, 1.41]^T$ and two robot workspaces by setting the `is_dense` parameter in the RViz bring-up command (see [Section 3.1](#)) as follows:

Case 1: Simple (`is_dense:=false`);

Case 2: Cluttered (`is_dense:=true`)

```
ros2 launch robot_3r_bringup robot_3r_bringup.launch.py \
launch_planning_nodes:=true run_headless:=true is_dense:=false # modify
```

For each case, run the planning node for $\epsilon = 1.0$ and simply report your findings. No figures.

3.3 Optimizing Planner Performance by Parameter Tuning

Sampling-based planners are highly parametric, so understanding the role of each parameter is essential for tuning performance. In ROS 2, a node's parameters can be listed via `ros2 param dump <node_name>`. Launch RViz and the planner in separate (sourced) tmux panes (with the commands provided in [Section 3.1](#)). Then in a third pane, run:

```
ros2 param dump /sampling_based_planner
```

You should see a list of parameters that the planner node declares and their default values. To obtain descriptive information about a specific parameter, we can use the following command:

```
ros2 param describe <node_name> <param_name>
```

Retrieve the description of the `rrts_expand_dist` parameter for the `/sampling_based_planner` node (verify the node name using `ros2 node list`). You should obtain:

```
Parameter name: rrts_expand_dist
Type: double
Description: Maximum distance (in radians) between nodes in the RRT* tree.
Controls how far the tree grows in a single iteration.
Recommended value: 0.1 - 0.5 radians.
```

Terminate all running nodes, and run the RViz bringup command from [Section 3.1](#). Then run the planner node for `rrts_expand_dist` $\in \{0.01, 0.05, 0.5, 1.0\}$ using the following command. Observe the waypoint counts recorded in each case: 0, 25, 6, and, 4.

```
ros2 run robot_3r_planners sampling_based_planner --ros-args -p \
goal_config="[1.51, 0.61, 1.41]" -p check_collision:=True -p \
use_goal_biased_sampling:=True -p rrts_expand_dist:=0.01 # modify
```

Indeed, large `expand_distance` values yield faster exploration with fewer, longer edges, but at the cost of precision and a higher risk of grazing or colliding with obstacles. Smaller values produce slower yet denser searches, though excessively small settings may prevent finding any path. Other parameters, such as the minimum obstacle clearance `min_obs_dist`, the neighborhood radius `rrts_connect_circle_dist`, and the discretization step `rrts_path_resolution`, also influence performance but are omitted for brevity.

Exercise 1. Efficient Path Generation via Informed Sampling (25 pts.)

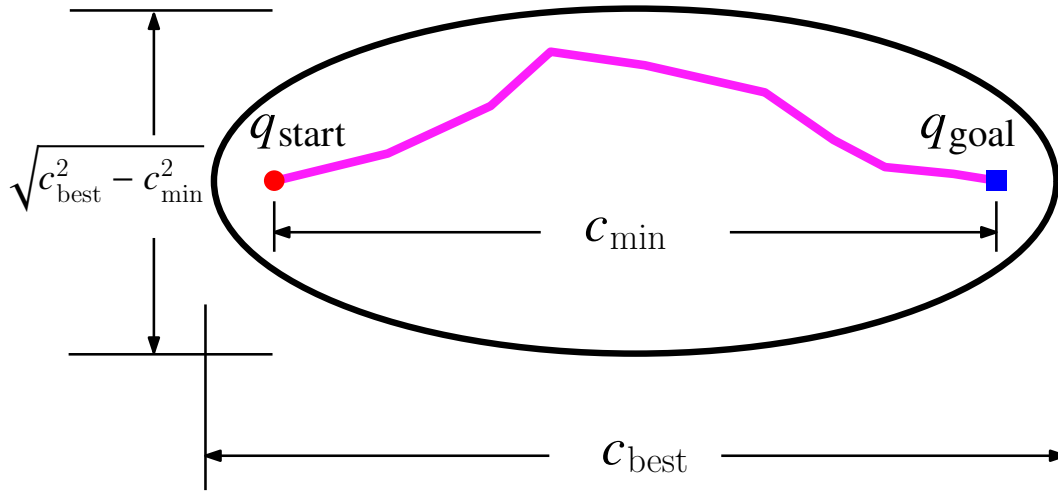


Figure 7: Ellipsoidal sampling region used by the Informed RRT* algorithm.
Figure adapted from the original [Informed RRT* paper](#).

Once RRT* finds a feasible path, it can either stop or continue improving it via rewiring. However, standard RRT* continues to sample the entire C-space uniformly, wasting effort in regions that cannot yield a shorter path. *Informed RRT** instead restricts sampling to an ellipsoidal region containing all paths with lower cost than the current best (see [Figure 7](#) and the original paper). For an n -dimensional C-space, the ellipsoid is n -dimensional, with one major axis (along the first C-space dimension) and $n - 1$ minor axes corresponding to the other C-space dimensions. The semi-major axis length (a) is given by:

$$a = \frac{c_{\text{best}}}{2},$$

where c_{best} is the current best path cost. Each of the $n - 1$ semi-minor axes lengths are given by

$$b = \frac{\sqrt{c_{\text{best}}^2 - c_{\text{min}}^2}}{2}, \quad (3)$$

where c_{min} is the minimum possible path length (straight-line distance between the start and the goal configuration) given by:

$$c_{\text{min}} = \|q_{\text{start}} - q_{\text{goal}}\|_2. \quad (4)$$

Once the ellipsoid has been generated, a new configuration q_{ellipse} can be sampled from it using:

$$q_{\text{ellipse}} = (R_{\text{align}} \cdot L_{\text{scale}} \cdot q_{\text{unit}}) + q_{\text{center}}, \quad (5)$$

where R_{align} is a rotation matrix aligning the start-goal line to the major axis, L_{scale} is a diagonal scaling matrix, q_{unit} is an n -dimensional unit vector in C-space, and q_{center} is the ellipsoid's center.

In this exercise, you will complete a partial implementation of the Informed RRT* algorithm in ROS 2 that we have provided in the file, `informed_rrts_planner.py` under the `exercise` folder within the `robot_3r_planners` ament_python package.

Deliverables: Complete the `informed_rrts_planner.py` planner script by modifying one existing method and adding one new method as described below:

Ellipsoid Parameter Computation Method:

- a. Create a new class method named `compute_ellipse_params` within the inner class `InformedRRTStar`, placing it directly after the `sample_free` method under the `TODO` block. This method must compute the planner variables `c_min` and `b` using (4) and (3). Use the template below and modify only the lines marked with `# <---MODIFY`.

```
def compute_ellipse_params(self):
    try:
        q_start = np.array(self.start.q, dtype=float)
        q_goal = np.array(self.end.q, dtype=float)

        # c_min is the Euclidean distance between q_start and
        # q_goal in C-space
        self.c_min = 0.0 # <--- MODIFY (Equation 4)
    except Exception:
        self.c_min = float('inf') # DO NOT EDIT THIS LINE

    # c_best is a class variable, so use self.c_best to access it
    if hasattr(self, 'c_best') and self.c_best < float('inf'):
        self.b = 0.0 # <--- MODIFY (Equation 3)

    # DO NOT EDIT THIS BLOCK
    if not np.isfinite(self.b) or self.b <= 0.0:
        self.b = 1e-6
    else:
        self.b = None # DO NOT EDIT THIS LINE
```

Informed Sampling Method:

- b. Modify the `InformedRRTStar` class method `informed_sample` to update the class variable for the ellipsoidal sample (`q_ellipse`) using the formula in (5). The beginning of the `informed_sample` method is labeled with a `NOTE` comment. All necessary class variables are already defined in the code template. Your task is simply to compute `q_ellipse` by substituting the variable names into (5). The line to modify is as follows, so you can search for it within VSCode using `Ctrl + F`:

```
self.q_ellipse = 0.0 # MODIFY (Equation 5)
```

Building the Package:

After completing your modifications, in one tmux pane, build the `robot_3r_planners` package by navigating to the workspace root and running `colcon`:

```
cd ~/ros2_ws/
colcon build --symlink-install --packages-select robot_3r_planners
source install/setup.bash
```

Then start the RViz bring-up packages:

```
ros2 launch robot_3r_bringup robot_3r_bringup.launch.py \
launch_planning_nodes:=true run_headless:=true is_dense:=false
```

Then in another sourced tmux pane, test the informed RRT* planner node using the command:

```
ros2 run robot_3r_planners informed_rrts_planner --ros-args -p \
goal_config:="[1.51, 0.61, 1.41]" -p check_collision:=True -p \
show_ee_path:=True -p use_goal_sampling:=True
```

Expected Behavior and Results:

If your implementation is correct, the planner will perform up to 10 refinement attempts after the first RRT* solution is found, printing the cost after each attempt (improvements should begin around the 5th attempt). At the end of the 10th attempt, you should see the following output in the terminal:

```
First RRT* solution cost 3.143
First RRT* waypoint count (17)
-----
Final Informed RRT* solution cost 2.380 (improvement 0.763)
Final Informed RRT* waypoint count (12)
  [0] [0. 0. 0.]
  ...
 [10] [1.318 0.636 1.272]
 [11] [1.51 0.61 1.41]
=====
Publishing final path with 12 waypoints
```

The final attempt will also be published automatically, and both the joint and Cartesian paths will appear in RViz with markers. Before terminating the nodes, make sure you have collected all required files and results for this exercise (see the **Files to Submit** section).

Files to Submit:

- The modified `informed_rrts_planner.py` Python file with complete implementations of the `compute_ellipse_params` and `informed_sample` methods
- A screenshot of your tmux pane showing the path cost of the naive RRT* algorithm and the improved path cost realized by the informed RRT* algorithm (see yellow box above)
- A screenshot of your RViz window showing the generated joint-space path and Cartesian path (in `magenta` color).

4 Polynomial Trajectory Generation in C-Space

Once a C-space path is planned, it must be time-parameterized into a smooth trajectory $q(t) : [t_0, t_f] \rightarrow \mathcal{Q}$ to avoid abrupt or high-velocity motion. The trajectory satisfies $q(t_0) = q_{\text{start}}$ and $q(t_f) = q_{\text{goal}}$, with optional velocity or acceleration constraints. Two trajectory types exist: *point-to-point (P2P)*, moving directly from start to goal, and *via-point*, which passes through intermediate configurations. Joint trajectories are often represented by an m -th order polynomial:

$$q(t) = a_0 + a_1 t + a_2 t^2 + \dots + a_m t^m, \quad t \in [t_0, t_f]. \quad (6)$$

The polynomial order (m) depends on the number of end-point or boundary constraints. Specifying position (q) and velocity ($v = \dot{q}$) at both ends (q_0, v_0, q_f, v_f) requires a *cubic* polynomial ($m = 3$); adding acceleration (α) constraints (α_0, α_f) requires a *quintic* polynomial ($m = 5$).

Question 7. What is the minimum polynomial order needed for a trajectory with *jerk* (third derivative) constraints at both end-points? (**Hint:** see the section on higher-order polynomial splines at [Robotics Unveiled](#).)

4.1 Point-to-Point Polynomial Trajectory Generation

4.1.1 Computing the Polynomial Coefficients

Assuming a cubic polynomial trajectory ($n = 3$ in (6)), the coefficients of the polynomial can be obtained by solving for x in the following linear system using the motion boundary conditions:

$$\overbrace{\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix}}^A \overbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}}^x = \overbrace{\begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix}}^b, \quad v_0 = v_f = [0, 0, \dots, 0]^T = 0_n \in \mathbb{R}^n. \quad (7)$$

NumPy provides a function, `numpy.linalg.solve(A, b)`, for solving systems of linear equations:

```
coeffs = np.linalg.solve(A, b)
```

Once the coefficients are known, the trajectory can be evaluated at discrete time steps between t_0 and t_f . To visualize the resulting motion, we typically sample N points uniformly in this interval, where N controls the trajectory's smoothness. To see this in action, inspect the standalone Python script (`cubic_trajectory.py`) located in the `~/ros2_ws/src/concepts` folder. Then, in one `tmux` pane, change directory to the `concepts` folder and run:

```
python3 cubic_trajectory.py --numpoints 50 --tf 6.0
```

Here, `tf` sets the final time (t_f), and `numpoints` determines how many interpolation points are evaluated along the cubic trajectory (i.e., N). You should see a plot showing the joint positions and velocities, where each interpolated point lies exactly on the computed cubic trajectory.

Question 8. At what time (in s) does Joint 1 reach maximum velocity for $t_f = 8$ s? (**Hint:** $\frac{t_f}{2}$.)

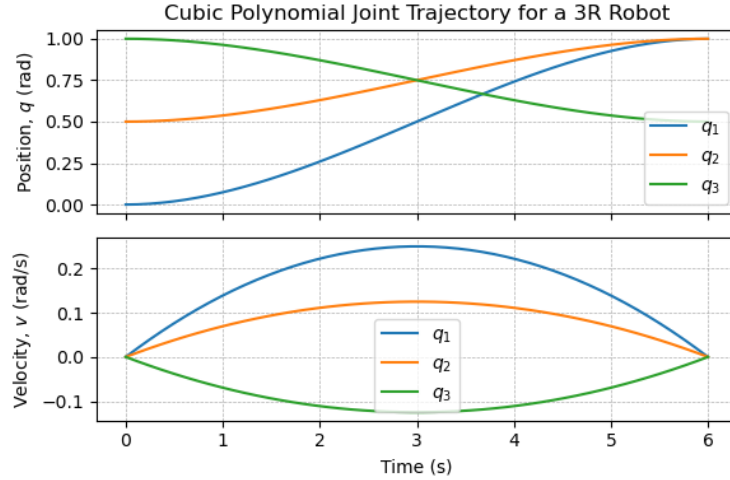


Figure 8: Sample cubic polynomial trajectory for a 3R robot.

4.2 (Piecewise-Cubic) Trajectory Generation for Via-Point Paths

In many tasks, the end-effector must pass through one or more intermediate configurations, or *via-points*. Let $M = \{q_0, q_1, \dots, q_{|M|-1}\}$ be the ordered set of via-points. For each segment $i = 0, \dots, |M| - 1$, define the cubic trajectory generation function:

$$f_{\text{cubic}}(q_i, q_{i+1}, t_0, t_f, N) \mapsto (\text{times}, q_i(t), v_i(t), \alpha_i(t)), \quad (8)$$

where:

- $q_i, q_{i+1} \in M$ are the start and end configurations of segment i ,
- $t_0 = 0.0$ s, t_f = duration of segment i ,
- N = number of interpolated points per segment,
- $\text{times} \in \mathbb{R}^N$ is an array of time samples,
- $q_i(t), v_i(t), \alpha_i(t) \in \mathbb{R}^{N \times |M|}$ constitute the positions, velocities, and accelerations of the trajectory corresponding to segment i and returned by f_{cubic} .

Each segment is solved independently using the same cubic coefficient computation as in the point-to-point case (see (6), (7)), and the final trajectory is obtained by concatenating all piecewise segments into a single trajectory (e.g., a single **JointTrajectory** message), as you will see from working through the next exercise.

Exercise 2. Via-Point-Path Trajectory Generator (25 pts.)

Inside the `ros2_ws/src` folder is an `ament_python` package named `robot_3r_trajectory` containing code templates that we will fill in with trajectory generation methods. Change directory to the `ros2_ws` workspace root, build the package, and source your workspace:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select robot_3r_trajectory
source install/setup.bash
```

Then check that the package is installed by listing its executables using:

```
ros2 pkg executables robot_3r_trajectory
```

You should see the following printed to the terminal

```
robot_3r_trajectory    via_point_trajectory
```

Deliverables: Complete the `ViaPointTrajectoryGenerator` node class as follows. Relevant blocks are highlighted with a `TODO` comment within the code:

Via-Point Trajectory Generation Method:

- Modify the highlighted lines within the `TODO` highlighted block of the class method `generate_via_point_trajectory` using the starter template provided in the code. The helper class method `_generate_cubic_trajectory` (corresponding to f_{cubic} in (8)) has been provided and handles the cubic polynomial coefficient computation and interpolation between two configurations. You only need to fill in the end-point configurations, $q_0(q_i)$, $q_f(q_{i+1})$, times, t_0 , t_f , and the number of interpolation points (N) for each segment (set this to the class parameter, `points_per_segment`). The segment duration, $t_f - t_0$, is stored in the class parameter, `duration`, and $t_0 = 0.0$ for all segments.

Send Trajectory Goal via Timer Callback:

- Once the via-point trajectory is generated, it must be sent to the robot's controller. A class method `send_trajectory_goal` is provided for this purpose. In your node's `__init__` method, create a class timer named `send_traj_goal_timer` with a 1 s period, and set its callback to `send_trajectory_goal`.
- Immediately after sending the trajectory to the robot's controller, the timer from part (b) should be stopped to prevent additional action requests. To cancel the node's `Timer` object, call its `cancel()` method within the callback `get_result_callback`.
- What would happen if the timer were left running and never canceled after the trajectory execution finished? (**Hint:** see the comment on the `oneshot` parameter [here](#).)

Test Your Via-Point-Path Trajectory Generator

After making the modifications to your `via_point_trajectory.py` file, in a tmux pane, run:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select robot_3r_trajectory
source install/setup.bash
```

Next, in another tmux pane, start the bring-up nodes (note the new `exercise` argument):

```
ros2 launch robot_3r_bringup robot_3r_bringup.launch.py \
launch_planning_nodes:=true run_headless:=true is_dense:=true exercise:=true
```

Then run the `via_point_trajectory` node:

```
ros2 run robot_3r_trajectory via_point_trajectory
```

If implemented correctly, the 3R robot will move from its start to final configuration while

passing through **19** via points, and the active terminal should output the string: “Goal successfully reached!”. **Note:** The node uses static via points from an RRT* path (Section 3); in practice, the path planner provides via-points online, which the trajectory generator consumes.

Files to Submit:

- i. Your modified `via_point_trajectory.py` script containing a complete and fully working implementation of the above methods
- ii. A clear (.webm, .mov, or .mp4) video of the robot in RViz showing the 3R end-effector hitting all marked via points (see Figure 9). You can use Ubuntu’s default [Screenshot tool](#) to record it.

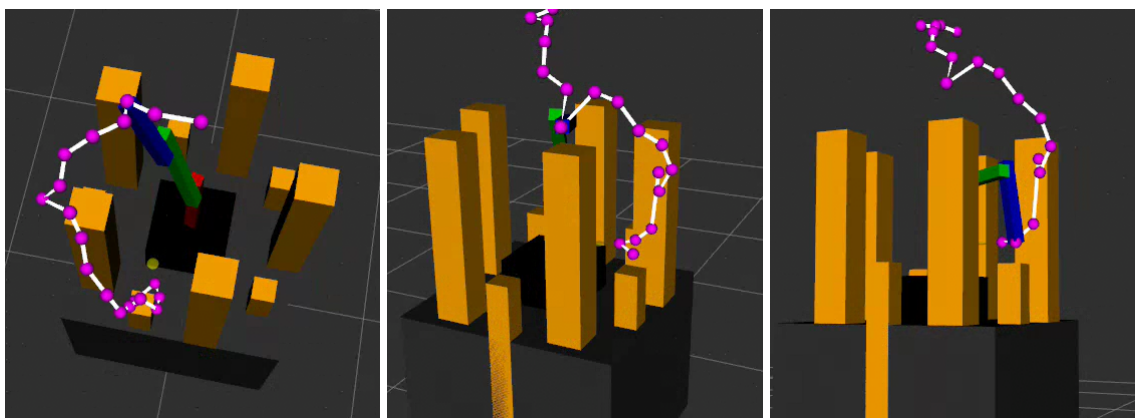


Figure 9: 3R trajectory execution. Via points (magenta); EE path (white line).

5 Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Include your solutions to all of the question boxes in the lab report. Refer to the [List of Questions](#) to ensure you don’t miss any. Include your full solution to the exercises in your lab report. If you wrote any Python programs during the exercises, include these files as a separate attachment, appending your team’s number and last names (in alphabetical order) to the file, e.g., **Jane Doe, Alice Smith, and Richard Roe in Team 90** submitting their modified `main.py` would submit the file `main_Team90_Doe_Roe_Smith.py` and the report `Lab_<LabNumber>_Report_Team90_Doe_Roe_Smith.pdf`, where `<LabNumber>` is the number for that week’s lab, e.g., **0, 1, 2**, etc. Submit your lab report along with any code to ELMS under the assignment for that week’s lab. You must complete and submit your lab report by **Friday, 11:59 PM** of the week following the lab session. See the **Files** section on ELMS for a lab report template named `Lab_Report_Template.pdf`. The grading rubric is as follows:

| Component | Points | Component | Points | Component | Points | Component | Points |
|----------------------------|--------|----------------------------|--------|----------------------------|--------|----------------------------|--------|
| Lab Report | 10 | Question 1 | 5 | Question 2 | 5 | Question 3 | 5 |
| Question 4 | 5 | Question 5 | 5 | Question 6 | 5 | Question 7 | 5 |
| Question 8 | 5 | Exercise 1 | 25 | Exercise 2 | 25 | | |
| | | | | | | Total | 100 |