Motion-Planning Lab 2

Introduction:

In this lab, we will get some hands-on experience in planning for a robotic manipulator. We will develop general-purpose sampling-based motion-planning algorithms such as the RRT but tailor the implementation to one specific platform – the UR5e Robotic manipulator. This manipulator features six DOF and an additional end-effector with a gripper.

The lab will consist of three parts: In the first, we will develop the basic algorithmic building blocks required to implement a sampling-based motion-planning algorithm. In the second, we will use these building blocks to implement a motion-planning algorithm and run it on the robot. Finally, in the third part we will use the planner to implement a task that relies on the planner developed in the first and second parts.

Background material:

The course "Algorithmic motion-planning" (236901) provides all the necessary material to complete this lab. However, understanding robot manipulators will be helpful (this is the platform that we will use…). Thus, the course "Introduction to robotics" (236927) provides additional relevant background. The most-relevant material that is not taught in "Algorithmic motion-planning" relates to forward and inverse kinematics.

Forward kinematics - Forward kinematics (FK) is a concept used to describe the process of determining the position and orientation of the end effector (typically the tip or tool) of a robotic arm, given the joint angles or displacements of its various segments (i.e., the robot's configuration). In simpler terms, FK answers the question: "If I have a robot with specific joint angles or lengths, where will the end of the robot's arm be located and how will it be oriented?"

Mathematically, forward kinematics involves using the geometric and kinematic relationships between the different segments of the robot to compute the transformation matrix that represents the position and orientation of the end effector with respect to a specified reference point. This reference point is often taken as the robot's base or another fixed location.

Inverse kinematics - While FK deals with determining the position and orientation of a robot's end effector based on its joint angles or displacements, inverse kinematics (IK) involves solving the opposite problem: finding the joint angles or displacements that will position the end effector at a specific desired location and orientation. In other words, IK answers the question: "If I want my robot's end effector to be at this particular position and orientation, what joint angles or lengths should the robot's segments have?"

IK problems can be more complex to solve than forward kinematics, as they often involve solving non-linear equations and may have multiple possible solutions or no solution at all in some cases. Depending on the robot's structure and the complexity of the problem, solving inverse kinematics might require iterative numerical methods, optimization techniques, or even symbolic manipulation.

For additional reading material on FK and IK, see Robot Kinematics: Forward and Inverse Kinematics.

Code provided

You are given the following Python files:

- **run.py** – includes the main function `main()` which loads the UR's parameters, the environment, the transform and the planner. It then takes a hardcoded start and goal configurations and calls a planner and visualizes the result.
- **planners.py** – includes the `RRT_STAR` planner.
- **kinematics.py** –includes (1) `UR5e_PARAMS` class which defines the manipulator's geometry and (2) `Transform` which implements the transformations from each link of the manipulator to the base_link.
- **building_blocks.py** includes the `Building_Blocks` class which implements basic functions used by the planner (e.g. sampling, collision detector, local planner).
- **environment.py** – includes the `Environment` class which defines the positions of the spheres that encapsulate the obstacles in the environment.

- **inverse_kinematics.py** -provided to find a configuration given the position and orientation of the end-effector.

- **RRTTree.py** - include the `RRTTree` class which implements tree data structure with functions to add vertices and edges, and find nearest neighbors.

Simulation vs. Real

A common approach in robotics is testing the algorithms in a simulated environment before experiments on real hardware. Simulations enable rapid prototyping, cost effectiveness, safety and easy debugging. In our setting, we will first visualize paths via a simple python interface, then, an advanced real-world simulation based on ROS (Robot Operating System) can be used to enhance simulation reliability. Since there is always a so-called sim-to-real gap, after achieving satisfactory results in simulation, the algorithms should be tested on real hardware.

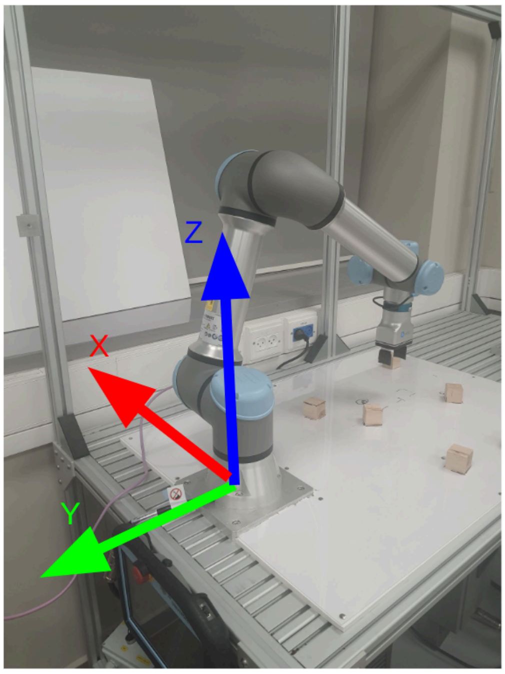
We will the UR_ROS_DRIVER [UR_GIT, UR_DOC] which provides:

- Framework for simulation in ROS.
- Interface for various motion planning algorithms (OMPL) via the Moveit2 package.
- Driver to communicate with real hardware.

Part 2 – RRT* Planner

In this section you will implement RRT* and study the effects of the parameters on its performance. Then, we will demonstrate the paths in an advanced simulation and at the lab.

1. In the lab, there is the added constraint of hitting the window. one solution is adding a wall obstacle constructed by spheres but it result in heavy computations in the collision checking phase. Instead, **add** a condition to the function `is_in_collision()` to return `True` if the manipulator exceeds the plain 0.4 [m] in X direction. provide the function the configuration [130, -70, 90, -90, -90, 0][deg] (convert to radians using the numpy.rad2deg() function) to verify it returns **True.**



2. Implement the `RRT_STAR` class in `planners.py`. The class contains 5 functions:
   - `find_path()` - main RRT-Star algorithms
   - `extend()` - updates the random sample to be in the same direction but at range of maximum step size.
   - `rewire()` - rewires the tree if shorter path exists.
   - `get_shortest_path()` - reconstructs the path from start to goal- return the the path and its cost.
   - `get_k_num()` - this function determines the number of k-nighbors to explore. You may use this function or change it as you wish.

3. Find paths for each test as described in the table below. **Report** the performance in **two figures**, one for each `p_bias` value, plotting the cost as a function of computation time for different `max_step_size` values. **Discuss** how the performance (cost and time) of the planner is affected by the hyper-parameters `max_step_size`, `p_bias`.

Since `RRT*` is non-deterministic you should provide statistical results (average over 5 runs for each test).

* convert [deg] to radians using the numpy.rad2deg() function.
- Set `env_idx=2` in the `Environment` constructor.
- Limit the maximum iteration to 2000 for each run.

ENV	start [deg]	goal [deg]
2	[110,-70, 90, -90, -90, 0]	[50, -80, 90, -90, -90, 0]

run the tests for `Max_step_size` = [0.1, 0.3, 0.5, 0.8, 1.2, 1.7, 2.0] and `p_bias` = [0.05, 0.2]

At Lab:

4. from the paths you have found, Choose the path with the lowest cost and execute it on the UR5e manipulator. **include** a video that visualizes it.
5. **Compare** the path with the shortest time you found to the path found by OMPL.