Motion-Planning Lab- HW3

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.

Software infrastructure

ToDo: complete

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 3 – Cubes task

In this section, you will find a plan that includes multiple paths and operating the gripper.

You are given 6 cubes in pre-defined positions. The positions described in `task3_run.py`.

Manipulate the cubes to form the first letter of your name. The environment is described as `env_idx=3` (See `environment.py`). You must use all 6 cubes. the cubes are 4*4*4 cm.

The cubes constructing the letter should be positioned in the square defined by x = 0.05, x = -0.34, y = -0.19, y = -0.55.

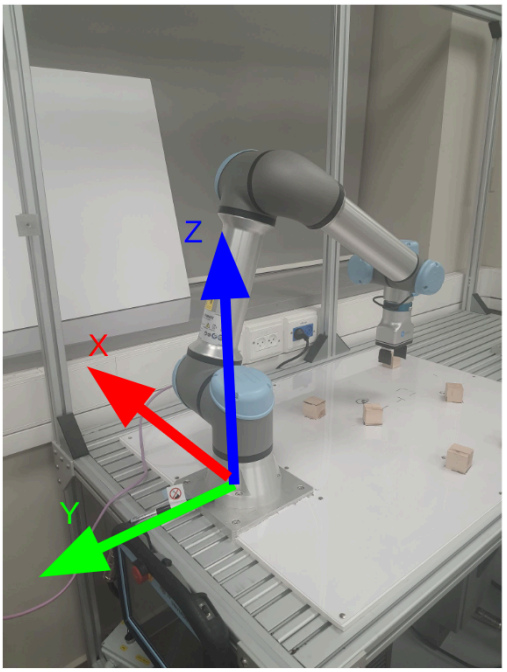
To avoid hitting the cube with the gripper we set the goal as cubex_approach, thus the cube represented as an obstacle while planning. To compensate for this gap we manually add the final (or initial) movement to the path to reach the cube. See the examples(#go to cube 1, #go to cube1_goal for cube1 in the `task3_run.py`).

Note that the environment changes as you move the cubes, thus, you have to consider it during the path planning process and update the obstacles list. You don't have to consider the cube carried by the manipulator as its size is negligible relative to the gripper. The initial coordinates of the cubes are given as the `inital_cubes_coords` list, update the list as you move the cubes, an example of how to transform configuration into coordinates is `given in task3_run.py`.

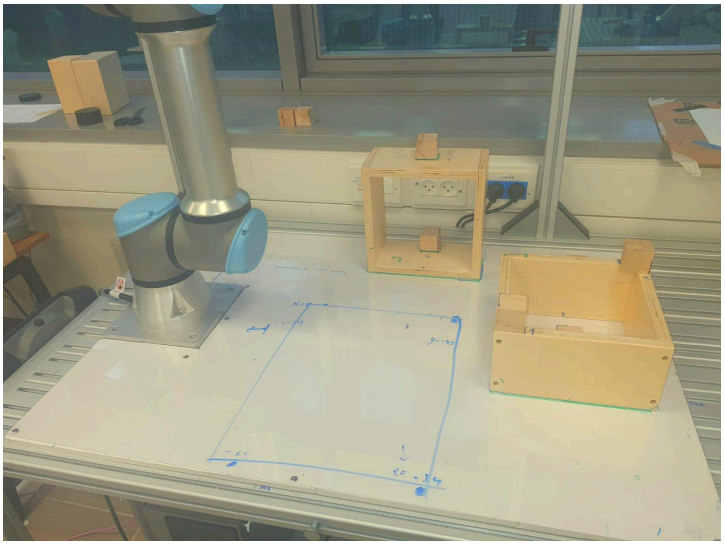
The goal configuration can be found by using `inverse_kinematics.py`. change the `tx and ty` parameters. these set the position of the end effector. `tz` is the height to drop the cube. keep it 0.1. unit in [meter]. `alpha, beta, gamma` (radians) set the orientation, you don't need to change those. The inverse kinematics may find several configurations for the inverse kinematics solution (given in radians) and filter out those that are in collision. You may choose any configuration but it would be better to choose those with the lowest cost from start to goal.

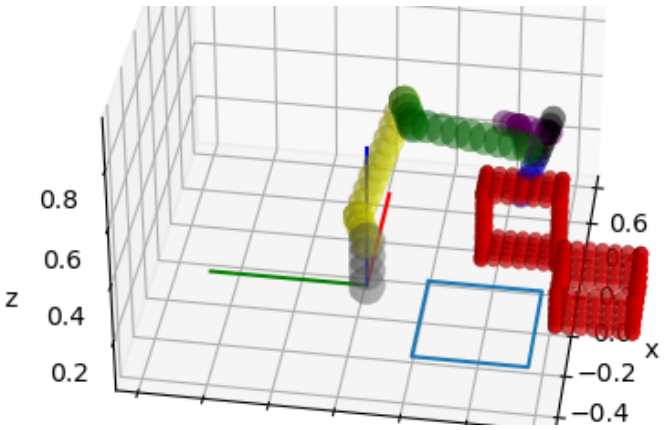
The initial configuration and final configuration of the manipulator is **home**(defined in code)

<u>reference coordinate frame:</u>

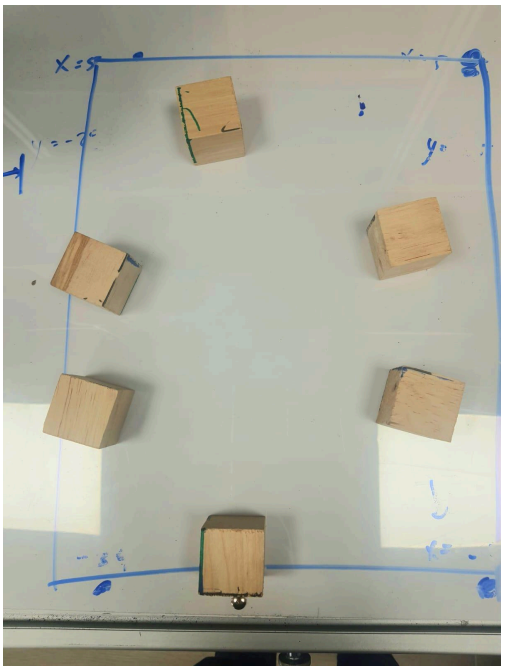


environment setting:





an example of the letter ‘o’:



1. Find paths to form your first name letter constructed by 6 cubes. Repeat the process for all team members. (one letter at a time) . save each path as numpy array.

2. Edit the variable `plan_list` to contain the paths names and command for the gripper('open', 'close'). for example: ['open', 'path_1.npy', 'close', 'path_2.npy', …].
3. At lab, bring the paths as numpy files and plan_list as .py file. Execute the plan to the UR5e and **provide** a video.