Motion-Planning Lab

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

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 1 – Algorithmic building blocks

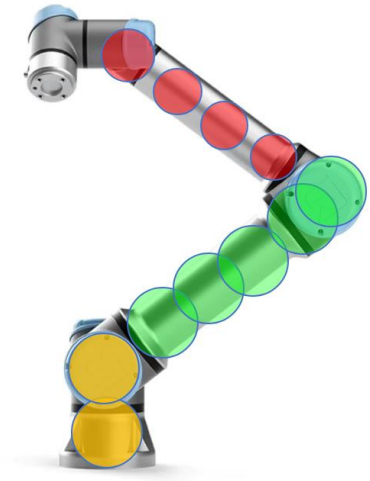
In this part we will fill in the missing part in building_blocks.py.

1. **Sampling** – fill in the missing part of `sample()` to implement configuration sampling for the robot, each joint should be in the interval defined in `UR5e_PARAMS.mechanical_limits`. Recall that we often employ goal biasing (this is a planner's parameter named `p_bias`). Thus, with probability `p_bias` the function should return the goal configuration and with probability 1 - `p_bias`, the function should return the random configuration.

2. **Collision detection** – Recall that a collision detector (CD) is used to determine whether a given configuration is valid or not. The collision detector `is_in_collision()` checks for two types of collisions: (i) internal collisions between different robot links and (ii) external collisions between the robot links and the obstacles.

Here, we will use a simple approach where obstacles and robot links are each modeled as a collection of spheres. We say that a collision occurs when spheres of different bodies (i.e., two different links or a link and an obstacle) intersect. To this end, we need to (i) decide how to model the robot and the obstacles as a collection of spheres (e.g., what radius should we use) and (ii) how to compute intersection between spheres (think about the sphere's radii and their respective centers).

Note that in general, we are willing to have false positives but not false negatives. Namely, it is acceptable that the CD returns that a collision occurs even when it does not but it is unacceptable to return that no collision occurs when one does.



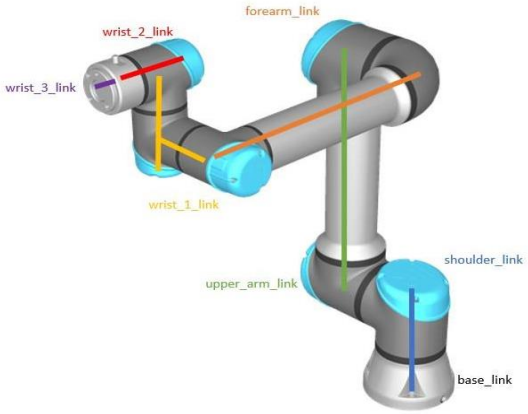
2.1. Assume that a link is modeled as a cylinder with a radius r, and length 10r and that all spheres have equal radii. Furthermore, assume for simplicity that the center of the link is located along the x-axis with one endpoint at (0,0,0) and the other at (10r,0,0). For the following, **give** the location of each sphere together with their radius in order to ensure that there are no false negatives while minimizing the number of false positives.

- 2.1.1. One sphere.
- 2.1.2. Two spheres.
- 2.1.3. Five spheres.
- 2.1.4. Ten spheres

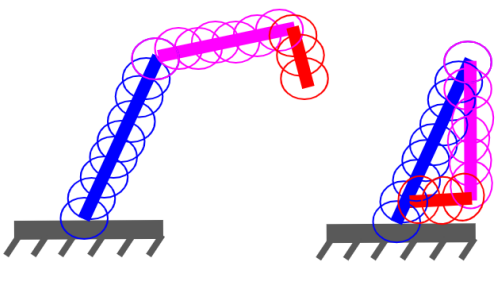
2.2. **Discuss** in general terms the tradeoff and effect of the number of spheres and their radius.

2.3. The `inflation_factor` provides a trade-off between the number of spheres and accuracy. Set values {1, 3} for the `inflation_factor` in the `ur5e_params` class constructor and run the run.py script to visualize the configuration [-0.694, -1.376, -2.212, -1.122, 1.570, -2.26][radians]. **Add snapshots depicting the results.** from now on, work with `inflation_factor=1`.

3. To determine the global coordinate system of each sphere we use transformations. These allow mapping the location of a sphere, given in the robot's local coordinate frame to a global coordinate frame given a specific configuration. We define the coordinate system of the base_link (see figure below) as a global coordinate system. With all sphere coordinates referenced to the common coordinate system we can check if a given configuration is valid.



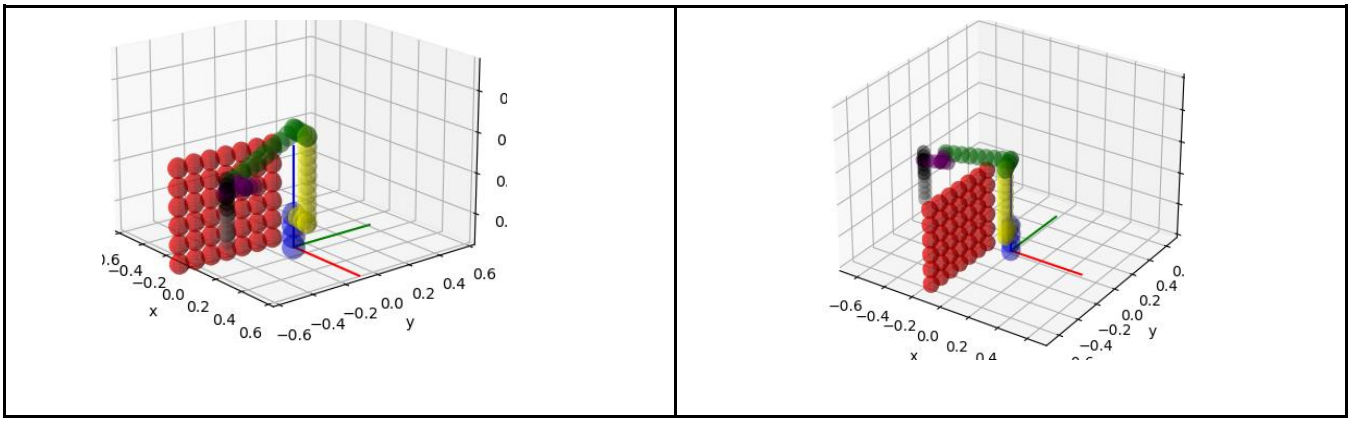
Consider the following simple 2D example where we assume that each link can rotate freely relative to the previous link.



3.1. Given a manipulator geometry, implement the function `is_in_collision()` in building_blocks.py. Here, the class `global_sphere_coords` is a dictionary with "keys": *link name* and "items": *list of coordination of spheres along the link*. **Provide** examples (i.e., a configuration and a snapshot) of (i) one configuration which is collision free and (ii) one configuration that is in collision.

3.2. Extend the collision detector to consider obstacles and floor. Here we treat the floor (i.e., z_cood = 0) as an obstacle. set the `env_idx=1` in `Environment` constructor. **Provide** examples (i.e., a configuration and a snapshot) of (i) one configuration which is collision free and (ii) one configuration that is in collision with the obstacle.

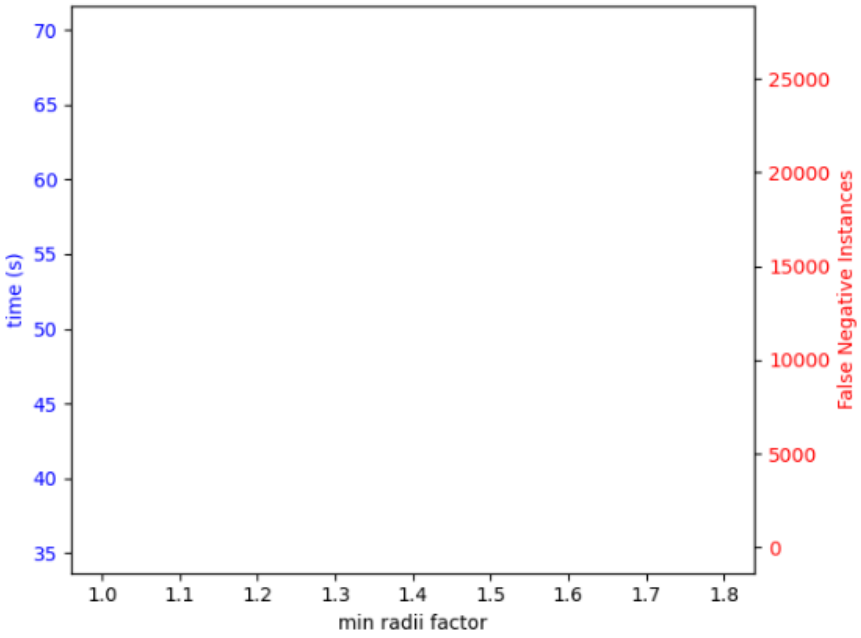
4. To determine if a transition between two configurations is valid we have to check for collisions in intermediate configurations. Implement the `local_planner()` function which takes two configurations *q* and *q'* and checks for collisions in intermediate configurations. The intermediate configurations are determined by the parameter `self.resolution`. Set the minimum value of configurations to check to 2, in this case the local planner checks for collisions only for the provided configurations and not in intermediate ones.
If a transition is valid the `local_planner()` returns True.
Set the `env_idx` to 1 in `Environment` class constructor and test the local planner with the following configurations:
conf1: [80, -72, 101, -120, -90, -10] [deg],
conf2: [20, -90, 90, -90, -90, -10][deg].
convert to radians using the numpy.rad2deg() function.



In this case, It is obvious that the manipulator cannot move from conf1 to conf2 directly without colliding the obstacle. Change the parameter `self.resolution` such that the local planner returns (1) True (2) False. **Report** the values you chose and how many configurations the local planner tested for each value. Finally, Set the minimum value of configurations to check to 3.

5. In the file random_samples_100k.npy, you are given a set of 100,000 random configurations. For each one, compute if it is in collision using the different inflation factors (e.g., an inflation factor of 1.5 inflates the minimal radius by 1.5). The inflation factors and template for the solution are provided at `task1_times_FN_as_function_radii.py` file.

Plot the overall computation time as a function of the radius used and the number of false-negatives as a function of the radius used in range of inflation factor [1.0, 1.8] in intervals of 0.1. Here, we treat the inflation factor = 1.0 as ground truth (the false negative for inflation factor = 1.0 is 0). For the other inflation factors values, we compare with respect to that ground truth.


70
65
60
55
50
45
40
35
time (s)
25000
20000
15000
10000
5000
0
False Negative Instances
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
min radii factor