HW 4—Coordinating two robots

1 General guideline

This homework consists of an open-ended research-like task in contrast to the previous assignments. Consequently, in the instructions, we will provide initial steps and guidance to solve the problem. However, solving the general problem is mostly left as an open question. You are more than welcome to change any part of the code to achieve the challenge. We strongly recommend on reading the whole document before coming to the lab to run your code. The homework must be submitted in pairs.

Submission date is 25.1.2026 end of day. One Zip titled HW_4_ID1_ID2 which includes the pdf + source code. ID1 and ID2 are your student IDs.

Code

The code for the homework is here. Reminder: use Python 3.12 (Macbooks with Apple silicon cannot control the robots)

2 Setting

In this exercise you will compute a plan for two robots tasked with moving a set of cubes from one side of one table to the other side of a second table. This requires computing a plan that involves multiple single-robot paths and multiple gripper operations. The environment is visualized in following images:

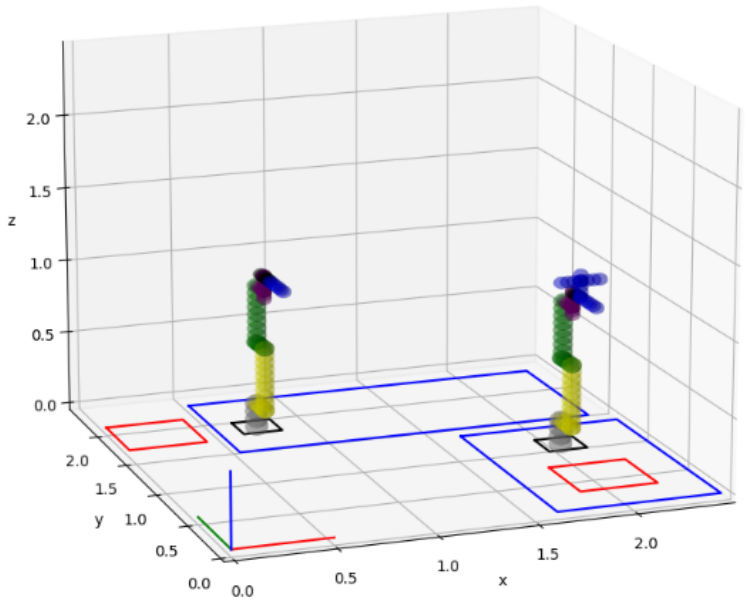

Figure 1

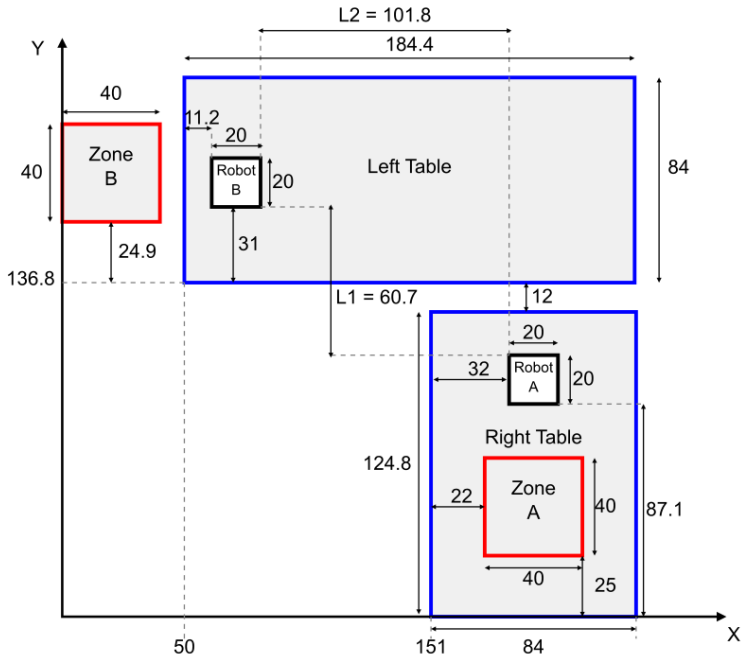

Figure 2: Dimensions are in cm.

As we can see, we have two robots (denoted as $\mathcal{R}_A$ and $\mathcal{R}_B$), one on each table (marked in blue) and next to each robot we have a zone for cubes (marked in red). The task is to move cubes from the red zone on the right table (Zone A) to the one next to the left table (Zone B). No robot can perform this task independently, which is why they need to collaborate: robot $\mathcal{R}_A$ needs to pick up a cube from zone A, hand it over to $\mathcal{R}_B$ who can place it in zone B.

There are no obstacles in the environment (other than the tables), the distances between the robots and the zones as well as other hard-coded measurements can be found in the template provided in the environment class (though they may later be changed as we explain shortly). In addition, the exact location of cubes in Zone A may change. Before executing your precomputed plans on the robots at the lab, we suggest to make sure all measurements are aligned with the environment file. There is another "wall" in the environment defined in code, it is not a real obstacle but it is more interesting to test your solution in a "harder" environment.

3 Structured task—quasi-static approach (70 points)

We start by using a simplified approach to solve the problem: Specifically, this solution will move one arm at any given moment with the second arm being static and treated as an obstacle for the first arm.

This approach requires planning multiple *steps* which include (i) planning paths for robot $\mathcal{R}_A$ from its initial configuration to zone A, (ii) computing a meeting point where $\mathcal{R}_A$ will transfer a cube to

robot B, (iii) planning paths for robot $\mathcal{R}_A$ to the meeting point, (iv) planning paths for robot $\mathcal{R}_B$ from its initial configuration to the meeting point, (v) planning paths for robot $\mathcal{R}_B$ from the meeting point to zone B. Note that this process will be repeated for every cube and that in between gripper operations need to be added to grab and release each cube.

3.1 Software infrastructure

We provide some infrastructure[1] to complete this (and the next) task. Feel free to use any previously-developed code and to adapt the given code as you see fit.

3.1.1 Files

- **main.py**: The main function for the experiment class.
- **experiment.py**: The main structure for your experiment. The function `plan_experiment` creates the variables (`env`, `planner`, etc.), computes the meeting point, plans the paths for each robot and finally create a JSON file with all the necessary information to run the robots.
- **planners.py**: Use a planner implemented from the previous homework (i.e., RRT or RRT*). You are more than welcome to change anything you would like (specifically we recommend to get the start and goal configuration as arguments for the function `find_path` and not in the initialization function.
- **control_robot.py**: A helper file that assists with the control of both robots using the JSON file created by the `plan_experiment` function. We elaborate on how this file is created in Sec. 3.1.2.
- **building_blocks.py**: serves as a core utility component for robotic motion planning
- **environment.py**: defines the physical workspace and obstacle management system for a dual-arm robotic setup (using UR5e manipulators)
- **inverse_kinematics.py**: It provides the mathematical mapping between the robot's joint angles (joint space) and its end-effector position and orientation (Cartesian space)
- **kinematics.py**: It defines how the robot's physical structure is represented mathematically and provides the Forward Kinematics (FK) logic required to translate joint angles into 3D positions
- **live_demo.py**: serves as the primary integration and demonstration script for the dual-arm robotic system
- **robot_interface.py**: acts as a high-level abstraction layer for controlling Universal Robots (UR) manipulators

[1] Note that MacBooks with Apple silicon may face trouble when running the code and alternative machines are recommended.

- **visualizer.py**: provides a 3D visualization suite for the dual-arm UR5e robotic system
- **plan_fix.json**: serves as the "recipe" that the execution scripts follow to perform a pick-and-place operation involving a hand-over between the robots. **It is an example** of an output your code should create

3.1.2 Planning information format

Recall that when a cube is transferred from zone A to zone B, it undergoes multiple steps (e.g., moving $\mathcal{R}_A$ to zone A or moving $\mathcal{R}_B$ to the meeting point). Each step contains different data to be used both by the simulation and the robot controller. Specifically, this is implemented using the dictionary `single_cube_passing_info` and saved in a JSON file.

The following data members are used and need to be filled by your implementation for each step of the plan. You can find two examples of how to fill this structure correctly in **experiment.py line 123**

- **description**: Text to be displayed during the animation.
- **active_id**: Current step's active arm which could be either `LocationType.RIGHT` or `LocationType.LEFT` corresponding to $\mathcal{R}_A$ or $\mathcal{R}_B$, respectively.
- **command**: There are two types of movements to be used (see example for specific details) which correspond to movement along a path described as a sequence of configurations or movement which is relative to the last position. Thus, the "command" can be either `move` or `movel` corresponding to the first and second types of movements, respectively.
- **static**: The configuration of the static arm.
- **path**: The path for the moving arm.
- **cubes**: Coordinates of cubes on the board at the given timestep.
- **gripper_pre**: An action to be executed by the gripper prior to executing the path corresponding to the given state. This could be either `OPEN`, `CLOSE` or `STAY`.
- **gripper_post**: An action to be executed by the gripper after executing the path corresponding to the given state. This could be either `OPEN`, `CLOSE` or `STAY`.

3.2 Task

The code contains three parts marked as "ToDo" Fill in each one according to the following guidelines.

Note 1. We provide an environment file containing the locations of each robot. In addition, we provide hardcoded locations of the cubes. When coming to the lab, the environment will be identical

to the one provided and the hardcoded cube locations will be marked. **However**, we will test your submission on slight variations of these (i.e., we will slightly move the robot and / or the initial and final cube locations).

Note 2. You will be evaluated on how fast you compute a solution.

ToDo 1 This task requires computing the meeting point $\mathcal{M}$ for the two robots as well as a configurations for $\mathcal{R}_A$ and $\mathcal{R}_B$ such that their respective end effectors will be at $\mathcal{M}$. You can use IK (inverse kinematics) to compute these configurations. Note that if both end effectors lie in exactly the same position, a collision will occur, thus a slight distance between the two is required.
experiment.py line: 190

ToDo 2 Complete the first step of planning moving $\mathcal{R}_A$ to a position above the first cube in zone A. The only thing you have to fill in is $\mathcal{R}_A$'s configuration in this step.
experiment.py line: 121

ToDo 3 Complete the function `plan_single_cube_passing` to complete the whole set of steps transferring the cube from zone A to B. You can use relative move for short moves, such as grasping the cube or releasing it. Having said that, you are expected to run a planner to compute the main part of the path and not hardcode this part using multiple relative moves.
experiment.py line: 147

A video (sped up by 4×) demonstrating such a plan is here.

4 Research Task (30 Points):

Now that the first task was completed, you probably observed that moving one robot at a time is simple yet highly inefficient. In addition, you might have seen that moving multiple cubes will require long computation time that can be reduced as the problems are similar.

Now it's your time to improve this algorithm!

Can you think of a way to *dramatically* reduce computation time and / or execution time? You don't have to improve both planning and execution time but implement one non-trivial improvement.

5 Deliverables

For each task (Sec. 3 and 4), **describe** your approach and **add** a (i) gif visualizing one successful run in the simulation and (ii) a video of the robot in the lab successfully executing the task.

In addition, **add** (i) average planning times and (iii) average execution times. Provide a breakdown of these times for each step as well as the total time. **Discuss** the computational bottlenecks, what can be done to further reduce planning and execution time?

Appendix - Setting up the environment in the lab

1. **Before even getting to the lab-** Make sure you have all needed python packages to run the `Control_Robot.py` file. You can make sure that it doesn't warn you on missing packages when trying to run the file.
2. **Connect to lab's router-** Connect to TP-Link_8BC0 router with password- 65729922.
3. **Set static address-** The router has no DHCP so the IP address needs to be configured manually. The IP address should be in the range 192.168.0.20-192.168.0.30 and the mask should be 255.255.0.0.
4. **Adjust the code-** Use the `Control_Robot.py` file, fill in the IP addresses of the robots. You can make sure you are correctly connected to the robots by using the "ping" command to the IP addresses. You can find the IP address of each robot in his tablet's settings.
5. **Validate measurements (!)-** Make sure the environment is as defined in the code (this should be the same as the measurements depicted in Fig. 2)
6. **Execute the plan!-** Run the `Control_Robot.py` file.

Alternative Way to connect the robots

In order to have Internet access while working with the robots, a wired connection is preferred. Connect to TP-Link_8BC0 router with an Ethernet cable and set the static IP address and mask address as shown above to the Ethernet port (Leave gateway empty). Now, you can access the Internet via WIFI while communicating with the robots through the Ethernet connection.

Locally Control The Robot

For instructional video visit this link