ECE 383/MEMS 442/ECE 590 Lab 4: Pick and Place… for Real This Time

Due date: 11/28/2018 at 5pm

Instructions: This assignment is to be completed *in groups of three*. You are allowed to discuss the problems with your classmates, but all work must be your own. You are not permitted to copy code or take written notes between groups, or look up online solutions.

All coding will be done first on the Klamp't Jupyter system. To run your group's program on the physical robot, you will first need to demonstrate your program running in simulation to a TA or the instructor for approval. Once approved, you will be given a scheduled duration of time during an in-class lab, during which you will copy your files to the robot's control computer, make a couple of small necessary changes, and attempt one or more runs on the physical robot.

To submit, upload your PickAndPlace.ipynb, ControllerCalibration.ipynb, RobotControllerEmulator.py, and (optionally) ur5gripper.rob files to Sakai.

A. Emulating the robot's controller in simulation

The first task is to build an accurate simulation model of the robot and its controller. The physical robot's controller API is given in robot_api/RobotController.py, and a basic emulated version that uses a Klamp't simulation is given in RobotControllerEmulator.py.

The Jupyter Notebook file ControllerCalibration.ipynb lets you test your simulation model compared to recordings from the real robot. This problem asks you to tune the PID controller and controller emulator so that the simulated trajectories match the real robot's trajectories as closely as possible. By doing so, you can be confident that your application logic, developed in problems B and C, will work reasonably well when transferred to the physical robot.

At the lowest level, the Klamp't simulation uses a PID controller, and simulates joint friction. More details can be found in https://github.com/krishauser/Klampt/blob/master/Documentation/Manual-Simulation.md. The PID gains and other simulation parameters can be set either in the data/robots/ur5gripper.rob file, or in code using the `SimRobotController.setPIDGains()` function.

At the middle level, the `UR5WithGripperController` class in RobotControllerEmulator.py provides the same basic API as the one in robot_api/RobotController.py. At the moment this is essentially a "pass-through" controller that sends commanded configurations to the PID controller.

ControllerCalibration.ipynb contains a set of test command sequences, each of which corresponds to recordings of the real robot in control_calibration_data/.

- Test1: A step function in the positions of joint 3 and the gripper.
- Test2: A step function in the positions of joint 1 and 3.
- Test3: A discontinuous "impulse" function in the positions of joint 3 and the gripper.

- Test4: A discontinuous "impulse" function in the positions of joint 1 and 3.
- Test5: A continuous sinusoidal position command in joint 1, 3, and gripper.
- Test6: Same as 6 but with a discontinuous commanded position at the start and steeper acceleration throughout the trajectory.

Each time you select a test, you will get plots comparing the commands, actual robot trajectory, and simulated trajectory. Inspect each task and explore what components of each trajectory are not simulated accurately.

1. Tune the emulator / PID gains to achieve more accurate simulation. You can modify the UR5WithGripperController class as you wish, but do not change the overall external API (`setConfig`, `getConfig`, `getVelocity`, `getCurrentTime`).
2. Describe your process and strategy for tuning the controller at the space at the bottom of the file.

B. Motion planning with collision constraints

The next task returns to PickAndPlace.ipynb. Again you are implementing routines for a pick and place sequence, except now you are required to plan collision free motions. Also, your pick and place routines will now be executed in a physics simulator.

The objects will be placed in arbitrary positions and orientations (with the markers visible to the camera), so make sure your planner is able to handle multiple situations.

1. Re-implement the pick and place routines from Lab 2, but making sure that the paths you return are collision free.

   During development, you are allowed to return None if your planner fails to find a path, but ultimately your planner should be able to solve any problem presented to it with high reliability. You can use heuristic methods or sampling-based planners, and you may fill out the indicated helper routines if you wish.

   There are several methods you may use for collision checking. The RobotModel.selfCollision() function, the routines in the klampt.model.collide module, the Geometry3D.collides() function, and the RobotCSpace.feasible() method. Consult the documentation for more information.

2. Describe your approach at the bottom of the file, including answers to the following questions. a) How do you ensure that the IK solution that you have found is feasible? b) During the pick portion of the task, you only have to make sure the robot is collision free. But once you have picked up an object, how can you ensure the object doesn't collide with anything?

3. The robot has very large joint limits, with each joint able to rotate +/- 360 degrees. The IK solver gives you some solution, but there may be equivalent solutions modulo 360 degree rotations of

a limb. Implement your planner so that it always takes the shortest way between subsequent configurations on the path.

Note that here there are two world models, `world` and `controllerWorld`. The planner is allowed to change `world` at will, so you should be doing all of your work there. It is only updated when sensor input is explicitly requested, so if you move a block, you will be able to see it in the simulator, but the robot must re-sense the environment for the planner to access this information. (Note that the robot is updated periodically as you execute a motion, which is done explicitly during `execute_trajectory`.)

C. Enhanced motion generation

Right now, the path execution procedures in PickAndPlace.ipynb . Each path is processed by the `trajectory_from_path()` function, which assigns times, and `execute_trajectory()`, which feeds the trajectory to the controller using a sequence of `setConfig()` calls.

1. Implement a strategy for smooth motion generation that speeds up and slows down between subsequent milestones in the path. Your technique should assign timing between milestones so that it does not exceed the robot's joint velocity limits and acceleration limits. You may use cosine smoothing, Hermite smoothing, trapezoidal velocity profiles, etc.
2. Describe the approach you use, including the mathematical details, at the bottom of the file.

D. Testing on the physical robot (10 points)

Once your group has completed problems A-C, you may ask the TAs/instructor to check your work for safety. The simulated robot should complete the pick and place task, and should not exhibit collisions or large jerks. Moreover, the controller emulator should behave fairly close to reality.

Once your system is approved, you may get a time slot for testing on the physical robot. The procedure for testing your behaviors will be described in more detail later.

* *Extra credit*: After you have tested your basic scheme successfully on the physical robot, your group should tune your planning and motion generation strategy for optimal performance. The team that successfully picks all three objects in the minimum time will receive 10 bonus points. Ideas for doing so may include 1) generating even smoother trajectories by modifying the planned path, 2) seeding the planner with good initial solutions to minimize planning time.