Homework 3

Due: April 07, 2022

Introduction

Manipulation and path planning is an important aspect of robotics. One important application is autonomous bin picking and order fulfillment solutions where a robot picks an item from one bin and places into another. More often than not, the robot has to move in a cluttered environment while avoiding obstacles. In this homework, you will manipulate simulated UR5 robot to perform grasping in simulation in PyBullet physics simulation engine.

In **Problem 1, 2, and 3**, you will implement RRT algorithm for path planning to move the robot from one location to another while avoiding collision with the obstacles. **Problem 4 is an extra credits problem**: you have the chance to integrate your homework 2 to build a complete grasping pipeline.

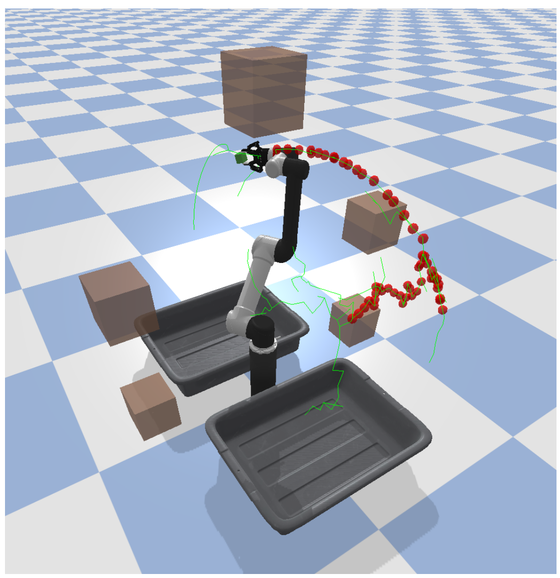

Figure 1: UR5 robot moving an object to another bin

You will be directly editing the provided python files. For extra credits, there are requested written explanations or images in Problem 4 (a). For this problem, compile your answers in hw3_report.pdf.

Apart from the code and report, you are also required to submit videos for all problems. These videos are screen recordings of your completed tasks. You can use any screen recording software as you desire. One choice is OBS or QuickTime Player for

Mac users. Please see reference video for Problem 1, 2 and 3 and reference video for Problem 4 to get idea about video submission and solution in general.

For more details on submission, please see Section Submission instructions and check-list.

Getting started

For this homework, we will install python interpreter and dependencies using mini-forge to avoid any version issues. Please follow the installation instructions for your system (Unix-like or Windows). You can also download the corresponding Mambaforge-OS-arch.sh/exe file at miniforge and execute the downloaded script.

After installation and initialization (i.e. mamba init), launch a **new** terminal and run

```
mamba env create -f environment.yaml
```

in side the unzipped homework zip. This will create a new environment named "comsw4733_hw3", which can be activated by running

```
mamba activate comsw4733_hw3
```

In addition, you need to install the latest version of pytorch if you choose to do Problem 4 (extra credit). Run the following on a machine with GPU available:

```
mamba install pytorch torchvision cudatoolkit=11.3 -c pytorch
```

Or the following on a machine with CPU only:

```
mamba install pytorch torchvision
```

Please do not introduce any other dependencies/packages without taking prior permission from TAs.

This assignment uses PyBullet simulation engine extensively and hence, we recommend you to read the *Introduction* section in PyBullet API documentation to get some working knowledge of the engine. To interact with the simulation GUI, you can change the camera viewpoint by zooming in/out or pressing *ctrl* key and dragging the cursor at the same time. Pressing *g* key will toggle the image windows on left side.

Problem 1: Basic robot movement (10 points)

Implement the function for robot movement, given where you want to move it.

In order to grasp objects from arbitrary locations and move them to another bin, you will need to first implement the basic robot movements. To move a UR5 robot, one can control six joint angles as shown in Figure 2. In this problem, given a target position and orientation of the robot's end effector, your task is to compute the values for all UR5 joint angles to reach them.

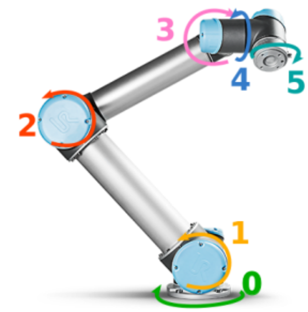

Figure 2: Various joints in UR5 robot

Complete *PyBulletSim.move_tool* method inside *sim.py* file which takes in the target position and orientation of the UR5 end effector link and computes a list of joint angles for all UR5 joints. For this, you should use *calculateInverseKinematics* method from PyBullet. Note that *calculateInverseKinematics* method takes in the link index (and not the joint index). We provide *PybulletSim.robot_end_effector_link_index* as the link index for the UR5's end effector. Once you complete the above method, run main.py file. If implemented correctly, all test cases for this part should pass.

Hint: You might find tuning parameters of PyBullet's *calculateInverseKinematics* method helpful for passing tests. Read the method description of *calculateInverseKinematics* in PyBullet API documentation to get a better idea of the parameters.

After implement all the parts, run

```
python3 main.py -part=1 -n=3 -disp
```

to test your code. The *-n* flag controls the number of trials to run. You can remove *-disp* flag to turn off visualization (note that you need to turn off visualization when running on a server). If implemented correctly, all test cases for this part should pass.

Problem 2: Grasping (30 points)

Get the object pose and move the grasper to grasp it.

We are using Robotiq 2F-85 for grasping objects. Robotiq 2F-85 is a parallel jaw gripper, i.e., one only needs to control opening and closing of the parallel jaws. For simplicity, we will use top-down grasping, i.e., roll and pitch of the gripper will be fixed to zero and we only need to calculate the grasp angle or yaw of the gripper. Moreover, we will perform grasping at the object's center and in perpendicular direction to the object yaw.

How can you be grasping top-down, and be perpendicular to the object yaw???

(a) Get the object pose (10 points)

Inside function *get_grasp_position_angle* in file *main.py*, compute object position and orientation using PyBullet's *getBasePositionAndOrientation* method. Use these to return *grasp_position* and *grasp_angle* for grasping.

(b) Execute grasp (20 points)

Once you have the *grasp_position* and *grasp_angle*, complete the method *PyBulletSim.execute_grasp* inside *sim.py*. You will need to implement the following grasp sequence:

- Open gripper
- Move gripper to *pre_grasp_position_over_bin*
- Move gripper to *pre_grasp_position_over_object*
- Move gripper to *grasp_position*
- Close gripper
- Move gripper to *post_grasp_position*
- Move robot to *PyBulletSim.robot_home_joint_config*
- Detect whether or not the object was grasped and return *grasp_success*

You will find Problem 1 helpful. After implement all the parts, run

```
python3 main.py -part=2 -n=3 -disp
```

to test your code. The *-n* flag controls the number of trials to run. You can remove *-disp* flag to turn off visualization (note that you need to turn off visualization when running on a server). If implemented correctly, all test cases for this part should pass.

Problem 3: Path planning using RRT (60 points)

Once the object is grasped, we will use RRT (Rapidly-exploring random tree) path planning algorithm to find a collision-free path.

How do you know where are collisions? Where do we get the entire environment?

(a) Implement RRT (40 points)

Implement the function *rrt* in *main.py* based on the following pseudocode.

Q: when trying to avoid collisions, how to take into account the different shapes of objects and how they would collide with things differently?

```
Algorithm rrt
    Input:
    - q_init: initial configuration
    - q_goal: goal configuration
    - MAX_ITERS: max number of iterations
    - delta_q: steer distance
    - steer_goal_p: probability of steering towards the goal
    Output:
    - path

    V <- {q_init}; E <- {}
    for i = 1 to MAX_ITERS
        q_rand <- SemiRandomSample(steer_goal_p) # with steer_goal_p
           ↪ probability, return q_goal. With (1 - steer_goal_p),
           ↪ return a uniform sample
        q_nearest <- Nearest(V, E, q_rand)
        q_new <- Steer(q_nearest, q_rand, delta_q)
        if ObstacleFree(q_nearest, q_new):
            V <- Union(V, {q_new})
            E <- Union(E, {(q_nearest, q_new)})
            if Distance(q_new, q_goal) < delta_q:
                V <- Union(V, {q_goal})
                E <- Union(E, {(q_new, q_goal)})
                path <- calculate the path from q_init to q_goal
                return path
    return None
```

Notice that for implementing *rrt*, you will need to implement other functions such as *SemiRandomSample*, *Nearest* etc. We have already provided *MAX_ITERS* in the code. Please don't change it. Feel free to play around with *delta_q* and *steer_goal_p*. While implementing *Nearest* function, you can use any appropriate distance metric, however we recommend simple euclidean distance. Also the chosen distance metric should be same across the *rrt* implementation. For implementing *ObstacleFree* function, ensuring if the new state *q_new* is not in collision with obstacles is sufficient. You should use *PyBulletSim.check_collision* method inside *sim.py* for this.

(b) Visualize exploration tree (10 points)

Modify *rrt* function to add visualization for the exploration tree (green lines in Fig. 1). For this, you should create a line between each newly added state and its nearest state in the RRT tree. You should use *visualize_path* function in *main.py* to add such a line.

(c) Transfer grasped object (10 points)

Once you have the path, implement the code for executing the path. While executing the path, you should also visualize the position of joint 5 (small red spheres in Figure 1). You can get the position of joint 5 with *p.getLinkState(env.robot_body_id, 9)[0]*. You should also use *SphereMarker* class in *sim.py* for this.

After executing the path, the robot should be on top of second bin. At this point open the gripper to drop the object. Close the gripper and retrace the path you just executed to the original bin. Delete all the visualizations once robot is back to its original position.

After implement all the parts, run

```
python3 main.py -part=3 -n=3 -disp
```

to test your code. The *-n* flag controls the number of trials to run. You can remove *-disp* flag to turn off visualization (note that you need to turn off visualization when running on a server). If implemented correctly, all test cases for this part should pass (note that this test is only for transferring the object).

Once you are done with Problem 1, 2 and 3, run all parts with display using the following command and screen record **a video for *3* trials** with your favorite software (one choice is OBS or QuickTime Player for Mac users):

```
python3 main.py -part=all -n=3 -disp
```

Please see a reference video for Problem 1, 2 and 3 here.

Problem 4 (Extra Credits): Pose estimation and grasping (20 points)

As extra credits, we will solve a problem **which can be transferred as it is to grasping in real world**. In this grasping approach, we make following assumptions:

- Access to precise 3D model of objects to be grasped
- RGBD camera (e.g. Intel realsense camera) which can give precise depth value for every pixel. We will use Pybullet functionality to mimic this camera
- Labelled training data for training a segmentation model. In real world, people use labeling platforms like Amazon Mechanical Turk to label segmentation masks given RGB images. For this assignment, we will use PyBullet's functionality to generate such data
- Lastly, we assume the knowledge of optimal grasp pose in object frame, i.e., if we attempt graping the object at this pose, it is very likely to be successful

Main pipeline for this grasping algorithm is as follows:

- Train a segmentation network to predict segmentation mask given an RGB image
- Generate point cloud of the object as follows:
  - Capture an RGBD image
  - Using RGB part of it, generate segmentation mask using the trained segmentation model
  - Mask out this object in depth image using the generated segmentation mask
  - From this depth mask, generate point clouds in world coordinates for this object

    WYM by point clouds, other than a bunch of points with 3-d coordinates in the world frame?
- Sample a point cloud from the original object model as well (remember we assumed the availability of 3D models of the object to be grasped)
- Using ICP, align the original object point cloud to the segmented object point cloud and hence get access to the object position and orientation in world coordinates

  Why… again? If we already have the original object point cloud (from the simulation), why we are seeing from cameras again…?

- Grasp the object by transforming the optimal grasp pose from object frame to the world frame (remember, we assumed knowledge of optimal grasp pose in object frame) Completely lost.. what?

(a) Collecting data and training segmentation model (5 points)

This part is exactly the same as what you did in homework 2. We have provided the file *gen_data_seg_model.py* to generate data required for training segmentation model. Before running this file, please setup dataset size as you find appropriate in the file *gen_data_seg_model.py*. When you run this file, a dataset directory will be created. Please visit this directory and analyze the data generated. In Figure 3, we show an example from generated data.

Figure 3: A sample from data collected for training segmentation model. First image is colored image, second is depth image and the third is segmentation mask for respective objects. Note that the depth image is visually enhanced version of what will be generated.

For training segmentation model, please complete the file *train_seg_model.py*. Note that since all test objects are known beforehand, we randomly split train test set in 9:1 ratio from the generated dataset during training. Using this split, train a segmentation model to reach an approximate test IoU of ≥ 0.9. See Figure 4 for sample output from a trained model. **Report dataset size, test loss and mean IoU of your final model. Report ground truth and predicted mask images from your final model on any two test set images.** For generating these images, you should use *save_prediction* function in *train_seg_model.py* file. **You also need to submit the trained model pth.tar file**. For saving a model, please use *train_seg_model.save_chkpt* function.

(b) Pose estimation using ICP (10 points)

Once you have trained a satisfactory segmentation model, figure out the pose and orientation of the object by aligning the known object point cloud to the segmented

Figure 4: Sample segmentation output from a trained model. Left most image is ground truth segmentation mask, middle is the input RGB image and right most image is prediction from a trained segmentation model

point cloud. For this, first capture an rgbd image of the bin. Then get the segmentation mask for the object using trained segmentation model. Mask out the object depth in depth image using this segmentation mask. Using the object depth, generate a point cloud for the object to be grasped in world coordinates (small red spheres in Figure 5). Let's refer this point cloud as segmented point cloud. Then, sample a point cloud from the original object. Let's refer to this point cloud as ground truth point cloud. Next, align these two point clouds using ICP and in doing so, figure out the pose and orientation of aligned ground truth point cloud. Small black sphere in Figure 5 represents the aligned ground truth point cloud. Complete *clear_bin.py* file which has more detailed steps on this part.

Is this point cloud NOT in the current position in the current world frame??

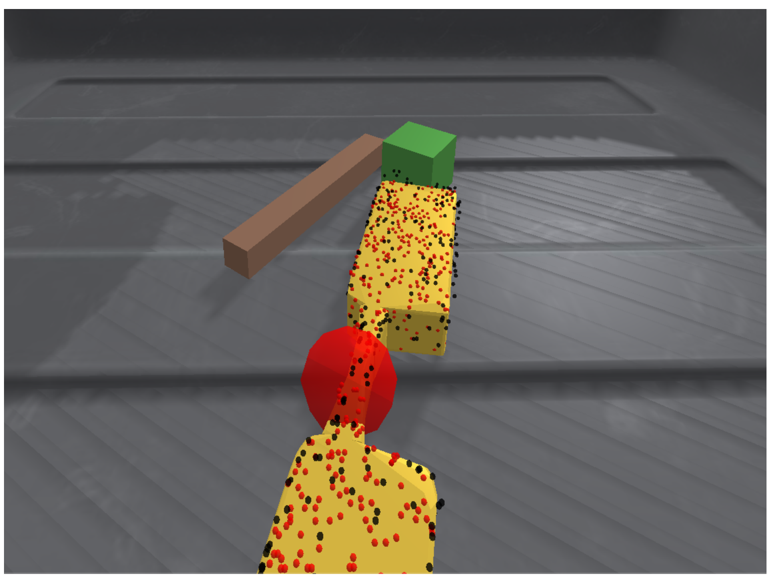

Figure 5: Aligned point clouds after performing ICP. Red points represent the points obtained from real world object using depth camera and segmentation model. The black points are aligned points sampled from the mesh of original object. The big red sphere denotes the transformed grasping location

What's the bins?

(c) Grasping and moving the objects to another bin (5 points)

Q: when trying to avoid collisions, how to take into account the different shapes of objects and how they would collide with things differently?

Implement code to grasp the object (pose obtained from the last part) and use the RRT algorithm from problem 3 to plan the path from one bin to another and move the object to second bin. Note that you might need to adjust the collision distance in order to avoid collision of object with obstacles during moving (see *check_collision* method in *sim.py* file).

Record a video showing the robot clearing a bin using above steps. Please find a reference video for this task here.

```
python3 clear_bin.py -disp
```

Note: It's possible that objects fall off during path execution, or the objects do not fall off even when the gripper opens. It's fine as long as your robot picks the object up, finds the correct path, and executes all the required actions. It's also fine if the robot needs a few tries to pick an object. It's also possible that objects are initialized with hard poses so that robot cannot pick them up. You can simply try to re-run the script, or reset objects in the script.

Submission instructions and checklist

Following is the submission checklist for Problems 1, 2 and 3:

- Generate URL for a single video containing solution to problem 1, 2 and 3 (see instructions for generating video URL at the bottom). Place the URL in *url_main.txt* file.
- Code: Running `python3 main.py -part=all` in the project root should run all three parts. Make sure to complete all *TODO*s in *main.py* and *sim.py*.

Following is the submission checklist for Problem 4 (Extra Credits) if you opt to complete it:

- Generate URL for a single video containing solution to Problem 4 (see instructions for generating the video URL at the bottom). Place the URL in *url_extra.txt* file.
- Code: Running `python3 clear_bin.py` should perform pick and place sequence. Make sure to complete all TODOs in *gen_data_seg_model.py*, *train_seg_model.py*, and *clear_bin.py*

- Place answers to the requested question in Problem 4 (a) in *hw3_report.pdf*. Place *hw3_report.pdf* in the hw3 root directory.

Finally Remove virtual environment folder *env*, *dataset* folder and any other folder that you might have introduced (for example: .git, .vscode, test python files etc). **Rename the directory as [Your-UNI]_hw3 and compress the directory and upload it to courseworks as [Your-UNI]_hw3.zip**. For example, yl4095_hw3.zip.

Please make sure that you adhere to these submission instructions. **TAs can deduct up to 10 points for not following these instructions properly.**

Please note that for all students, we will look at both the code and the video for grading. If significant discrepancies are found between the submitted videos and the results from manually running the code, we will report the student to Academic Committee. **We will also randomly sample a fair proportion of students and ask them for a homework interview**.

Generating video URL

Upload video to your personal google drive account (**not Lionmail**). Generate shareable link such that **anyone with the link can view the file**. For testing, open the link in Incognito Mode / Private window. If done properly, you should be able to view the video without logging in.

You should also make sure that the *Modified* date is visible. For this, click on burger button (three vertical dots) on top-right. Click on *Details*. You should be able to see the *Modified* date (see Figure 6).

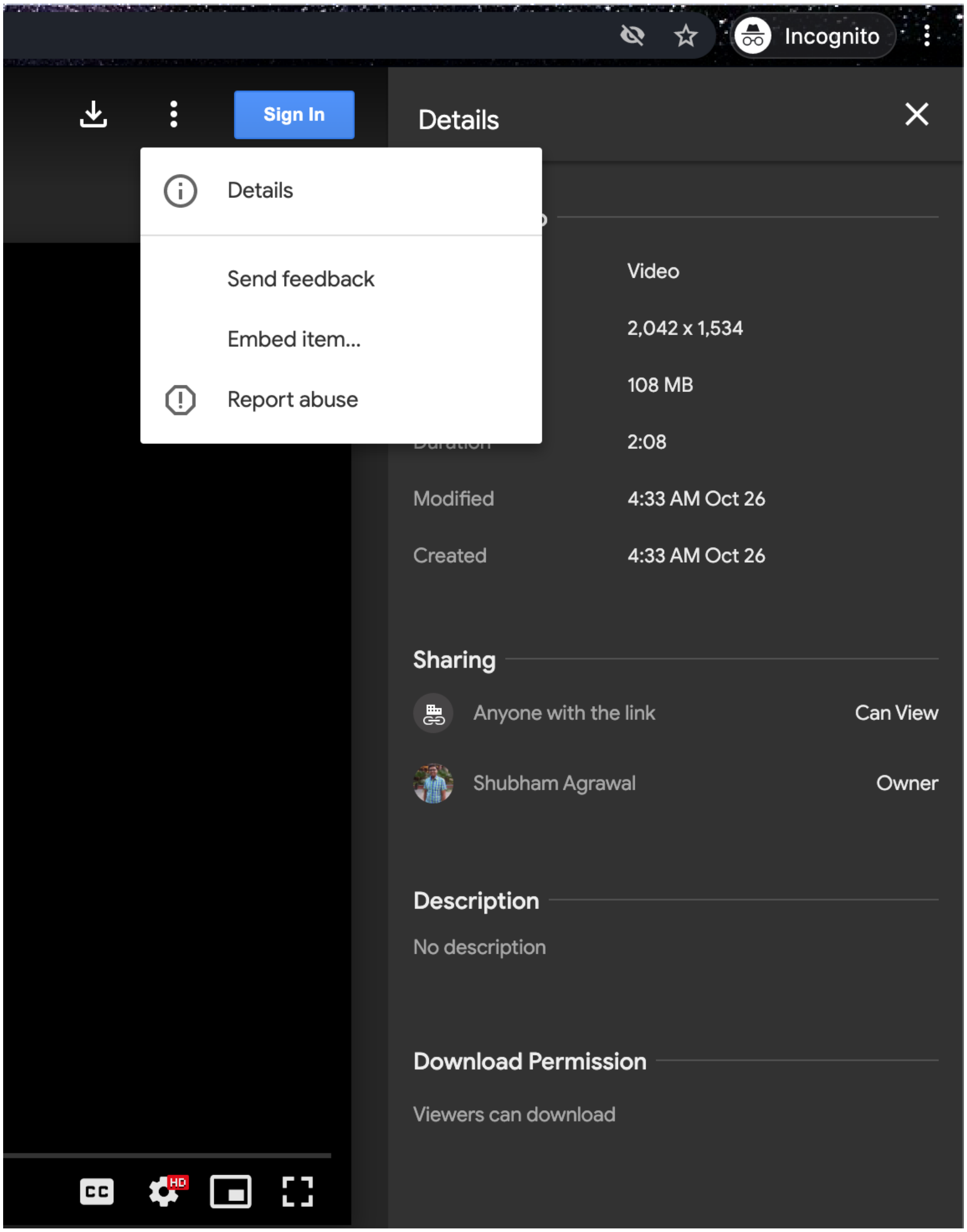

Figure 6: Video URL: should be accessible in Incognito / Private window. Modified date should be visible.