Homework 4

Instructions

In this homework assignment, we will continue to develop the remaining building blocks for a vision-based pick-and-place system. By the end of this assignment, we will complete the first version of our pick-and-place system – time to get excited!

This homework has three problems. Problem 1, basic concept review. Problem 2, coding assignment. Problem 3, discussion and analysis. For Problems 1 and 3, please compile your answers in a PDF titled `hw4.pdf`. For Problem 2, you will directly edit Python files provided by us.

Submission: Compress

- `hw4_report.pdf`
- code files `env.py` and `perception.py`
- video of the robot execution for 2.3 (video or shareable link in pdf)

into `SUID_hw4.zip`, and upload it to Canvas before the homework deadline.

Due Date: By 5:00 PM PST, Friday, Nov 14, 2025.

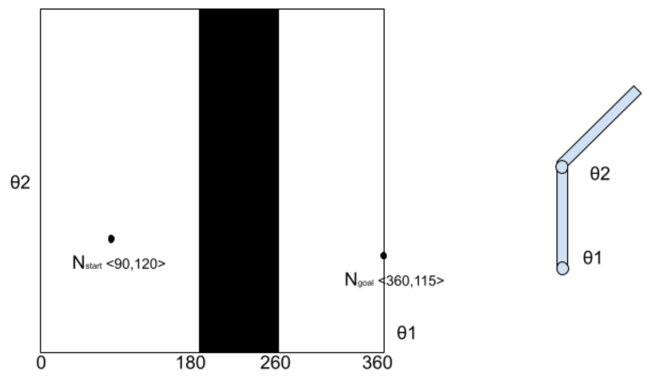
Problem 1 Concepts (45 pts)

1.1 Rapidly-exploring Random Trees (RRT) – 15 points

Given a two link robot arm with two revolute joints, let's define its configuration space (without obstacle) as $\theta_1 \in [0, 360), \theta_2 \in [0, 360)$, which wraps around on the boundary. There exists a C-space obstacle is shown below:

Now, we want to find a path from the starting point $N_{start} < 90, 120 >$ to the goal point $N_{goal} < 360, 115 >$ while avoiding the obstacles using RRT. Assume the step size is 10, and we use L1 distance as the distance metric. Let's complete the following steps. Note that the graph only contains N_{start} before the 1st iteration (N_{goal} is not part of the starting search graph as we are not doing bidirectional RRT in this problem).

1) (5 points) 1st iteration: random sampled point is $N_{rand} =< 70, 100 >$ What is the nearest Node N_{near}? What is the new Node N_{new} added to the tree? Is N_{new} valid?
Answer: We start with $N_{\text{start}} = \langle 90, 120 \rangle$ and generate $N_{\text{rand}} = \langle 70, 100 \rangle$.



Since we only have one node so far, our nearest node is $N_{\text{near}} = N_{\text{start}}$.

We use the L_1 (Manhattan) distance metric:

$$d_{L1} = |-20| + |-20| = 40$$

With a step size of 10, the scaled step vector toward N_{rand} is:

$$10 \times \left(\frac{-20}{40}, \frac{-20}{40} \right) = (-5, -5)$$

$$\Rightarrow N_{\text{new}} = N_{\text{start}} + (-5, -5) = \langle 85, 115 \rangle$$

Therefore:

$$N_{\text{near}} = \langle 90, 120 \rangle, \quad N_{\text{new}} = \langle 85, 115 \rangle$$

Because N_{new} lies within free space, it is **valid**.

2) (5 points) 2nd iteration: random sampled point is $N_{rand} =< 200, 120 >$ (N_{new} from the 1st iteration is on the graph already) What is the nearest Node N_{near}? What is the new Node N_{new} added to the tree? Is N_{new} valid?

Answer: On the second iteration, the random sample N_{rand} lies within the obstacle space. I'll show the calculations that proceed the RRT iteration as if Nrand is within the free space, but it's important to note upfront that it lies within the obstacle space and thus the Node new that will be generated is not considered to be a valid node. First, we compare $\langle 200, 120 \rangle$ to both N_{new} and N_{start} to determine which node to grow from.

By observation, $\langle 90, 120 \rangle$ is closer to $\langle 200, 120 \rangle$ than $\langle 85, 115 \rangle$, so we grow the tree from $N_{\text{near}} = \langle 90, 120 \rangle$.

L_1 distance:

$$d_{L1} = |200 - 90| + |120 - 120| = 110$$

Scaled step (steer):

$$10 \times \left(\frac{110}{110}, 0\right) = (10, 0)$$

New node:

$$N_{\text{new}} = N_{\text{near}} + (10, 0) = \boxed{\langle 100, 120 \rangle}$$

So although the N_{new} lies within free space, the Noderand is in the obstacle space, so it considered **not valid**.

3) (5 points) 3rd iteration: random sampled point is N_{goal}, (the nodes from the 1st and 2nd iterations are still on the graph) What is the nearest Node N_{near}? What is the new Node N_{new} added to the tree? Is N_{new} valid?

Answer:

On the third iteration, we have three nodes from which we can grow toward $N_{\text{rand}} = \langle 0, 115 \rangle$. The current nodes are:

$$\langle 85, 115 \rangle, \quad \langle 90, 120 \rangle, \quad \langle 100, 120 \rangle.$$

Because the configuration space wraps around at 360°, we compute the wrapped L_1 distance using the shorter angular difference.

For each node:

$$\begin{aligned} d(\langle 85, 115 \rangle, \langle 0, 115 \rangle) &= |0 - 85| + |115 - 115| = 85, \\ d(\langle 90, 120 \rangle, \langle 0, 115 \rangle) &= |0 - 90| + |115 - 120| = 95, \\ d(\langle 100, 120 \rangle, \langle 0, 115 \rangle) &= |0 - 100| + |115 - 120| = 105. \end{aligned}$$

Thus, the nearest node is:

$$N_{\text{near}} = \boxed{\langle 85, 115 \rangle}.$$

Steer with step size 10:

$$\Delta = (-85, 0), \quad d_{L1} = 85, \quad 10 \times \left(\frac{-85}{85}, 0\right) = (-10, 0).$$

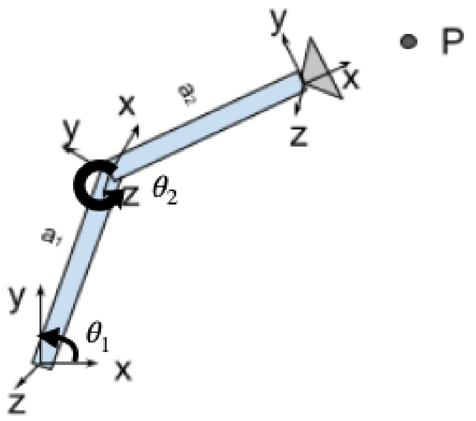
New node:

$$N_{\text{new}} = N_{\text{near}} + (-10, 0) = \boxed{\langle 75, 115 \rangle}.$$

Because N_{new} lies within free space, it is **valid**.

1.2 Kinematics & Pick and Place – 30 points

We have a two-link arm, shown below. Define the world coordinate in the robot base as shown below. All joints are revolute joints that rotate along the axis (pointing outward of the paper) with joint angle $<\theta_1, \theta_2>$ and link length $a_1 = 1,\ a_2 = 1$. Assume we mount a camera on the robot's end-effector, with the camera coordinate defined as shown. The camera is rigidly attached to the second link.



1) (20 points) Write down the camera pose in world frame ${}^wT_c$ in the form of $\theta_1, \theta_2, a_1, a_2$. Please express your answer in matrix form and you may express your answer as a product of matrices (you do not need to compute this product). Make sure to include a_1 and a_2 in your answer and to use 3D homogenous coordinates (meaning your transformation matrices are 4x4). **Answer:**

$$
{}^w\mathbf{T}_1 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad {}^1\mathbf{T}_c = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
$$

$$
\Rightarrow {}^w\mathbf{T}_c = {}^w\mathbf{T}_1\, {}^1\mathbf{T}_c = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
$$

2) (10 points) When $\theta_1 = \pi/2, \theta_2 = -\pi/2$, the robot is able to observe the target object and its position in camera frame is ${}^cP =< 1, 0, 0 >$, what is the object's position in world coordinate frame ${}^wP$? Is this object within the robot's reach

range? Why? **Answer:**
We can find the object's position in the world frame using:

$$^{w}P = {}^{w}T_c\, {}^{c}P$$

Substitute $\theta_1 = \frac{\pi}{2}$ and $\theta_2 = -\frac{\pi}{2}$ and $a_1 = a_2 = 1$ into

$$^{w}T_c = {}^{w}T_1\, {}^{1}T_c = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which gets you:

$$^{w}T_c = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Therefore:

$$^{w}P = {}^{w}T_c\, {}^{c}P = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \boxed{^{w}P = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 1 \end{bmatrix}}.$$

To see if this position is within the robot's reach, we use

$$r = \sqrt{(a_2 + 1)^2 + a_1^2}.$$

and see if:

$$|a_1 - a_2| \le r \le a_1 + a_2.$$

$$r = \sqrt{(1+1)^2 + 1^2} = \sqrt{5} \approx 2.236 > a_1 + a_2 = 2,$$

so the object is **not within reach**.

Problem 2 Implementation (40 pts)

Please implement the following functions in `env.py` and `perception.py`.

```
env.py
   move_tool() -- 10 points
   execute_grasp() -- 10 points
preception.py
   pose_est_segicp() -- 15 points
Record final video -- 5 points
```

Setup simulation environment

Same as the previous homework, install the relevant packages and start the environment. You do not need a GPU for this assignment.

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

After installation, you can check whether they are properly installed by running the following command:

```
python pose_based_pnp.py --check_sim
```

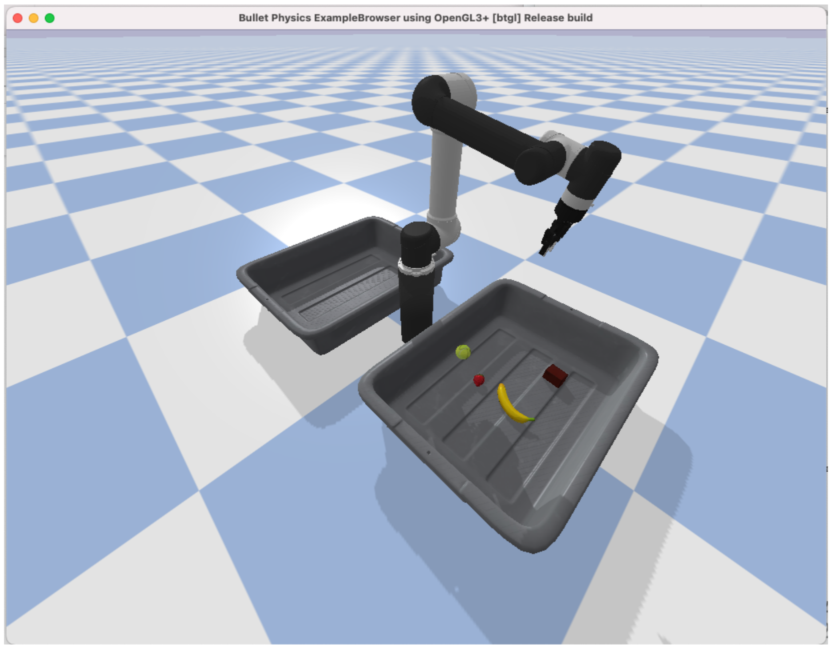
If everything is installed successfully, you should be able to use the pybullet simulation environment with UR5 robot, two bins and 4 objects (see figure below). The task for this assignment is to make the robot pick up all the objects in one bin and place them in the other.

In the terminal, you can see the following text:

```
Environment successfully loaded.  Press Ctrl-C in terminal to exit ...
```

Notes on PyBullet: This assignment uses PyBullet simulation engine extensively. We highly recommend 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 windows on the sides.

Read `class UR5PickEnviornment` initialization function to see how we load robots and objects in the environment, and how we add a top-down camera to this environment. While we provide this part of the code in this assignment, it is important to understand them so that you can setup your own environment in the future for different projects, robots and tasks.


Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build

System Overview.

The overall system has four high-level steps. In this assignment, we will first implement other parts of the system assuming ground truth state using `pose_est_state()`, then we will return to implement the perception module `pose_est_segicp()` and put everything together.

```
Step 1: Get top-down camera observation
    env.observe() -- return RGB-D image, we provide
Step 2: Estimate object poses using either
    pose_est_state() -- using simulation state, we provide
    pose_est_segicp() -- using pose estimation, you implement
Step 3: Compute grasp pose from object pose
    env.ob_pos_to_ee_pos()
Step 4: Execute pick and place action
    env.execute_grasp() -- you implement
    env.execute_place() -- we provide
```

2.1: Compute robot target configuration using IK – `move_tool()`

Complete this function that moves the robot tool (end-effector) to a specified pose. To do so, please use the PyBullet inverse kinematics function
(called `p.calculateInverseKinematics`) to find out the target joint configuration of the robot to reach the desired `end_effector` position and orientation.

`p.calculateInverseKinematics` takes in the end effector link index and not the joint index. You can use `self.robot_end_effector_link_index` for this. HINT: You might want to tune optional parameters of `p.calculateInverseKinematics` for better performance.

2.2: Implement the grasping primitive – `execute_grasp()`

Implement the following sequence of action for grasping. Read and understand other functions we implemented for you in the `env.py` file, and use them to complete this primitive.

1. open gripper
2. Move gripper to pre_grasp_position_over_bin
3. Move gripper to grasp_position
4. Close gripper
5. Move gripper to post_grasp_position
6. Move robot to robot_home_joint_config
7. Detect whether or not the object was grasped and return grasp_success

Once you implement this part you can run the code.
Note: setting `--use_state` will have the robot use the ground truth object poses to plan grasps.

```
python pose_based_pnp.py --use_state
```

The robot should be able to pick up all objects and transport them to the other bin. Note that the grasp may not always be successful; observe the typical failures. Question to think about: How do we check the grasp success? (no written response needed)

2.3: Implement pose estimation algorithm

Finally, use the functions you implemented in the previous homework to complete this pose estimation algorithm by implementing `pose_est_segicp` in perception.py. To simplify the implementation, here we assume the ground truth object segmentation is provided (skipping the UNet training and inference part). However, you still need to 1) get the object point cloud from mask and depth. 2) get object model point cloud. 3) use ICP to align this two-point cloud and get the object pose.

Once you implement this part you can run the code (here the robot uses ICP rather than ground truth state):

```
python pose_based_pnp.py
```

The robot should will attempt to pick up all objects and transport them to the other bin. Note that the grasp success is much lower now with the estimated pose.

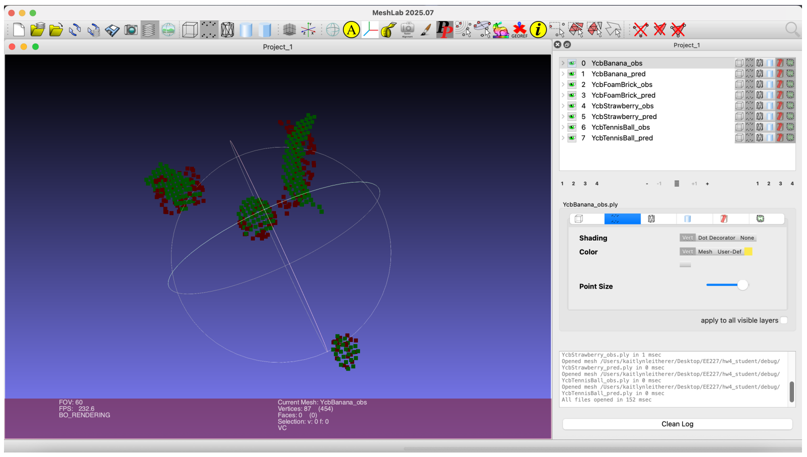
Record a video of the final system executing all four grasps with the ICP method for object pose detection. The system does not need to be perfect (the video should show at least two successful grasps); it's ok if you need to rerun a couple times before you get two successful grasps in one video. You can use any app (e.g., zoom) to screen record the video.

In Problem 3, we will look closer at the failure cases.

Problem 3 Discussion (15 pts)

3.1 Visualization & Failure mode – 6 pts

1) (2 points) Check the visualization of the pose estimation result (saved in '`./debug`'). Screenshot them and include them in your report. Below is an example of the visualization for the tennis ball.



0 YcbBanana_obs
1 YcbBanana_pred
2 YcbFoamBrick_obs
3 YcbFoamBrick_pred
4 YcbStrawberry_obs
5 YcbStrawberry_pred
6 YcbTennisBall_obs
7 YcbTennisBall_pred

Shading: Vert Dot Decorator None
Color: Vert Mesh User-Def

```
YcbStrawberry_obs.ply in 1 msec
Opened mesh /Users/kaitlynleitherer/Desktop/EE227/hw4_student/debug/
YcbStrawberry_pred.ply in 0 msec
Opened mesh /Users/kaitlynleitherer/Desktop/EE227/hw4_student/debug/
YcbTennisBall_obs.ply in 0 msec
Opened mesh /Users/kaitlynleitherer/Desktop/EE227/hw4_student/debug/
YcbTennisBall_pred.ply in 0 msec
All files opened in 152 msec
```

```
FOV: 60
FPS:   232.6
BO_RENDERING
```

```
Current Mesh: YcbBanana_obs
Vertices: 87    (454)
Faces: 0    (0)
Selection: v: 0 f: 0
VC
```

2) (2 points) Look at the visualization and robot behavior. Describe two common failure modes with this pick-and-place system. For example, in the earlier visualization, red is the prediction, and green is the observed surface, they are close but not exactly aligned, why? **Answer:**
The predicted pose estimation vs observed surface are not exactly aligned. This could be due to poor initialization. ICP relies on accurate correspondences and initial good alignment. If ICP is matching incorrect points, it could converge to a local minima and produce poor alignment results, like what we see in the screenshot.

3) (2 points) Based on the common failure modes, list two ways to improve the result.
Answer:
Bad results come from ICP's poor initialization, so there are a few ways to improve initial correspondence.
First, in order to mitigate any potential of ICP forming incorrect point matches, we could add more cameras and improve the point cloud so that ICP has the full object pose to match to. Second, we could run an algorithm like RANSAC to use before ICP so that it produces a better initial pose and reduces the potential of ICP converging to a local minima.

3.2 Assumptions – 9 points

1) (5 points) The current system does not consider obstacles in the environment. Describe how you could incorporate collision avoidance in the system using algorithms we learned in class. Which aspects need to be changed and what algorithms/methods could we use to enable collision avoidance? **Answer:**
We didn't consider obstacles in our environment, but we have collision-detection algorithms like Rapidly-Exploring Random Tree which can help us explore an unknown space by incrementally searching the configuration space for a solution. It would make sure the robot never selects an action that would land us in the obstacle space. To deploy it, we would need to make sure we created a C-space and defined our obstacle vs free space.

2) (4 points) List out at least 5 major assumptions we made in this pick-and-place system that may not be true in a real-world pick-and-place system. (Max: 100 words)
Answer: First, it never considered that the objects could be partially occluded, i.e. lighting was ideal. Second, it never considered that the object could not be rigid under the robot's grasp (strawberry could get mushy under grip). Third, it never considered that there may be a grasp pose that's not possible for the robot (i.e. outside of the robot's reach range). Fourth, it assumed that the robot's hardware execution was perfect. Fifth, the scene was clutter free and there was nothing in the way of the robot grasping one object (like another object).

Submission checklist

When you are done, create a new folder called `SUID_hw4` (replacing with your SUID) and copy in the required items listed below into the root directory of that folder. Then zip the folder into `SUID_hw4.zip` and upload to Canvas. Double check your submission contains:

1. `hw4_report.pdf`: that includes answers to P1 and P3.

2. Completed python code: `env.py, preception.py`
3. Video of the robot execution. If the video is too big, create a shareable link (e.g., upload to Google Drive) and include the link in the report. Double-check check the link is viewable to anyone.