ECE 470 (Spring 2020)

Final Report

Intelligent Beer-pong Robot

Xinglong Sun (xs15)
Haozhe Si(haozhes3)
Team: Beer-pong Robot
Gitlab: https://gitlab.engr.illinois.edu/haozhes3/ece470_project_update.git
YouTube: https://www.youtube.com/watch?v=G5rgLZmNWX4&t=17s

1 Abstract

Imagine a robot that can pick up a ball on the table and throw it into the cup. Wouldn't it be cool to place it in an amusement park like Disneyland or a bar for crowd-pleasing? For the ECE470 final project, we designed and built an intelligent beer-pong robot that could play beer-pong even better than a human does. With the application of industrial utilitarian robots gets more and more mature, people start to pay more attention to the potential social impacts of entertainment robots. Thus, the goal of us designing a robot like this is to bring fun and joy to people. There's no doubt that this is a challenging task, particularly for a team of two people. However, our smooth corporation and the ability to abstract and modularize the entire project convinced us of the feasibility of it. As expected, we ended up with a decent simulation of the robot equipped with almost all of our desired functionalities. The robot is able to locate and grab a ball smoothly and precisely throw it into a cup positioned at the other end of the table. Besides the great sense of accomplishment obtained during the process, our knowledge of several fields of robotics is also deepened. This includes perception and sensing, dynamics, and motion planning.

2 Introduction

We spent the first week getting ourselves familiarized with the Vrep simulator. It's always not an easy task to work in a new development environment, and lots of features of the Vrep are quite unique. It did take us some time to get comfortable with this simulator. After we managed to establish the connection between Vrep and our Python code, we started to plan our design. We modularized the entire project into several blocks and set a weekly goal to cope with each block. In this way, we gradually moved towards our goal. In the following sections, modules are introduced in the order as we completed them. We will first introduce the environment and simulation setup. The props in the setting will be listed and explained. A detailed block diagram will be provided in the Design Process section for a clear visualization of the workflow. We will then provide an analysis for each module, including Sensing, Dynamics, and Motion Planning. The experimental setup and testing procedures are discussed in the next section. A physical engine, **Bullet V2.78** in our case, is enabled in testing to approximate real work practices as much as possible instead of faking a seemingly workable simulation. Several data and results will then be provided for verification.

3 Environment

Our environment includes a UR3 robot, a Jacohand as the end effector, one global vision sensor, an office table, six cups, and two balls initially randomly located on the table. The goal for the robot is to locate the balls, pick them, and toss them into the cups on the other side of the table. We specifically choose the cups and the balls so that the diameter of the cup is only slightly greater than that of the ball. In that case, we can showcase the precise motion control and trajectory of our robot. Here is a diagram illustrating the aforementioned props:

Here are some specifications of the objects:

Desk:	Width: 0.89m Length: 1.83m
Ball:	Bounding Box Length: 0.05m Mass: 0.06545kg
Cup:	Bounding Box Length: 0.08m

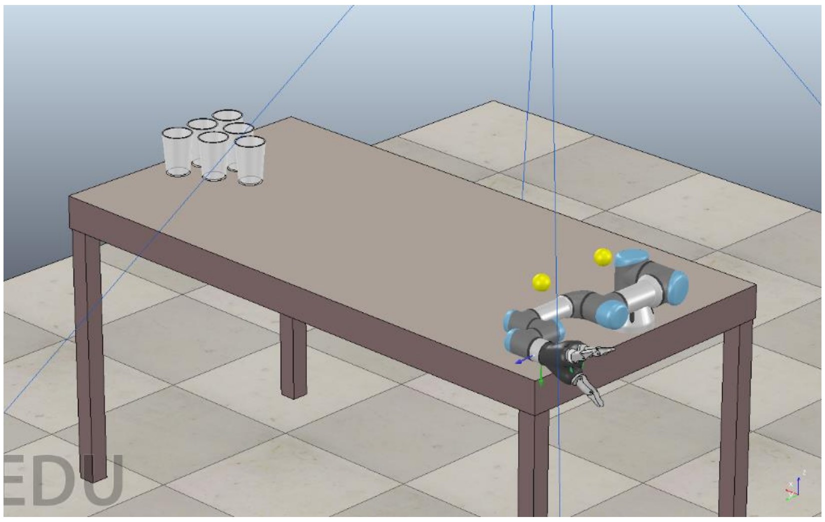
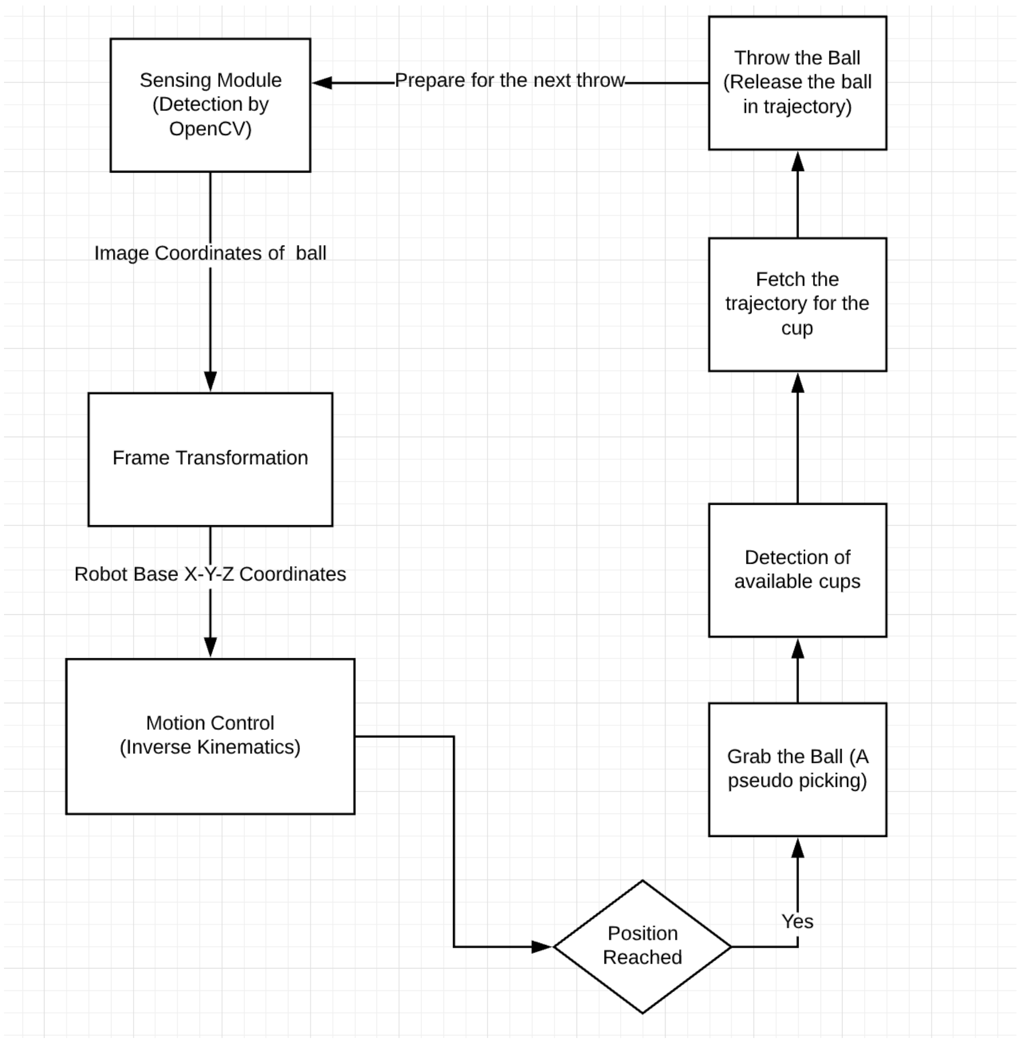


Figure 1the Environment Setup

4 Design Process

The following is a rough flowchart showing the workflow of the robot:



Sensing Module (Detection by OpenCV)
Prepare for the next throw
Throw the Ball (Release the ball in trajectory)
Image Coordinates of ball
Fetch the trajectory for the cup
Frame Transformation
Robot Base X-Y-Z Coordinates
Detection of available cups
Motion Control (Inverse Kinematics)
Grab the Ball (A pseudo picking)
Yes
Position Reached

Figure 2 Robot Workflow

5 Modules Explanation

5.1 Sensing

We used the OpenCV simpleBlobdector module to implement the ball detecting function. To be specific, we set a ball detection camera sensor in our environment to view the table from the top. In order to correctly locate the position of the balls, we chose to use the “filterByColor” function and setting the HSV range to filter yellow out according the color of the ping pong ball. Having the sensor and parameters set, we then used the simpleBlobdector to detect circles (ping pongs) in the image. We draw a small red circle around each Ping-Pong ball in the image captured by the vision sensor as an identification. To get the actual location of the ball in world frame, we then need to calibrate the camera and do the frame transformation. We arbitrarily set two balls in the environment and get their real distance, *exp_dist*, from reading their properties. With the help of simpleBlobdector, we can easily get the coordinates of the two circles and thus the distance of the two circle in units of pixels in the image, *img_dist*. By dividing image distance with the real distance, we can get the value of β, the constant value that scales distances in space to distances in the image:

$$\beta = img\_dist / exp\_dist$$

The equation for intrinsic transformation gives as follows:

$$r = \beta x^c + O_r$$

$$c = \beta y^c + O_c$$

Where (O_r, O_c) is the coordinates of *principle point* of image, in unit of pixels, and determined by the environment setting; (r, c) is the coordinates of image pixel returned by the simpleBlobdector. With β that we just calculated, we can easily get (x^c, y^c), the coordinates of point in camera frame. To transform the position in camera frame to world frame, we can simply add a translation constant, [T^x, T^y], to the camera frame coordinates since there is no camera distortion in simulation. The world frame coordinates is given by:

$$\begin{bmatrix} x^w \\ y^w \end{bmatrix} = \begin{bmatrix} x^c \\ y^c \end{bmatrix} + \begin{bmatrix} T^x \\ T^y \end{bmatrix}$$

The output of the ball detector is as follows:

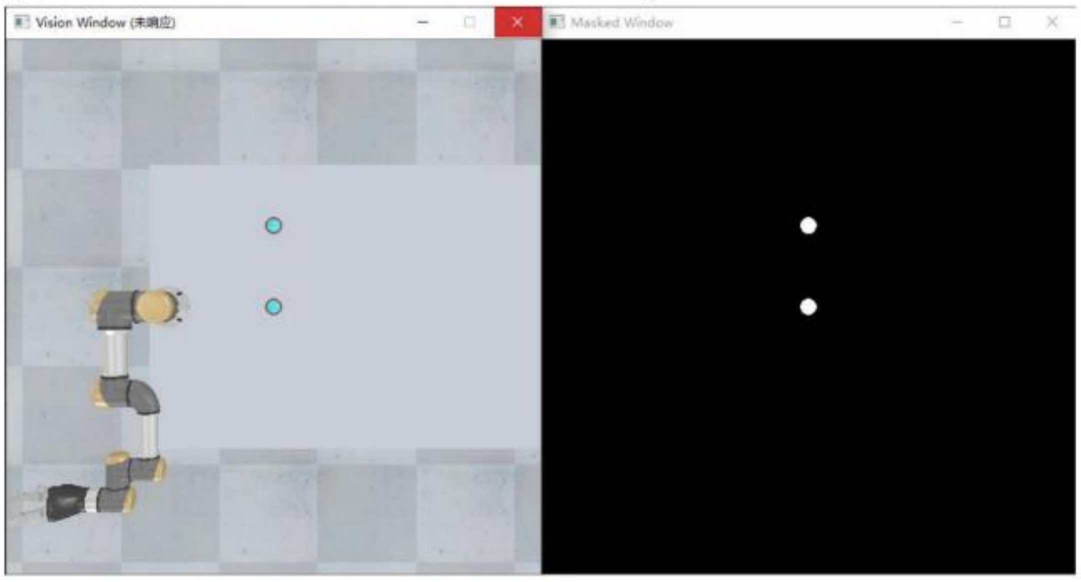


Figure 3 Ball Detection Output

5.2 Motion

To have the Jacohand pick up the ball, we implemented inverse kinematics to reach the target and used forward kinematics to verify the implementation

We implemented the forward kinematics by calculating the zero position and screw axis of the UR3 robot. To simplify our calculation, we assigned the body frame of the UR3 robot as the world base frame robot. The world frame and the zero position are defined as shown in fig.4.

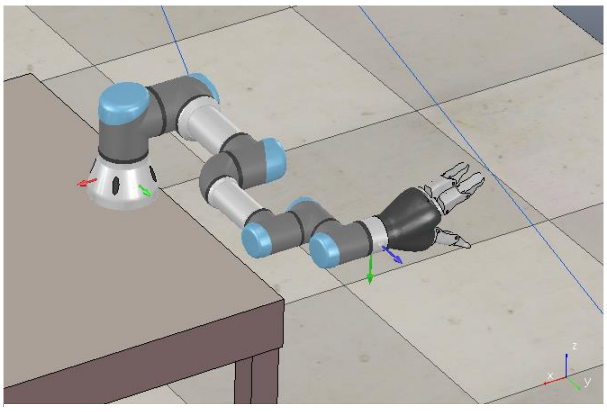


Figure 4 Robot Zero Position

To calculate the zero position, we used the dimension graph provided by UR3 robot as shown in fig.5. Since we also need the positions of every joints in order to calculate the screw axis in following steps, we noted them as well. The position of the six joints as well as the ender-factor in the robot base frame is as shown in table 1.

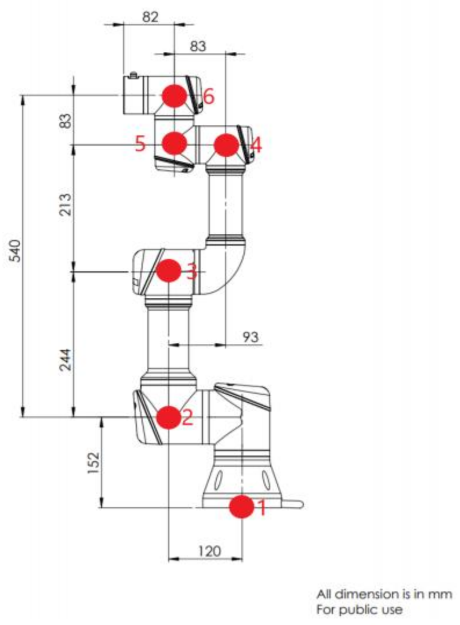


Figure 5 Robot Dimension

Table 1 Joint Position (Unit: m)

Joint #	x	y	z	Joint #	x	y	z
1	0	0	0	5	0.457	0.11	0.152
2	0	0.12	0.152	6	0.54	0.11	0.152
3	0.244	0.12	0.152	Ender Effector	0.54	0.251	0.2055
4	0.457	0.27	0.152				

As shown in fig.4, the rotation matrix from robot base frame to ender factor frame can be easily, which can express as follows:

$$R = \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

Thus, matrix M gives as follows:

$$M = \begin{bmatrix} 0 & 0 & -1 & 0.54 \\ 1 & 0 & 0 & 0.251 \\ 0 & -1 & 0 & 0.2055 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To calculate the screw axis, we need to determine the rotation axis, which specifies the angular velocity ω, in addition to the joint position q. The direction of the screw axis are define as in fig.6.

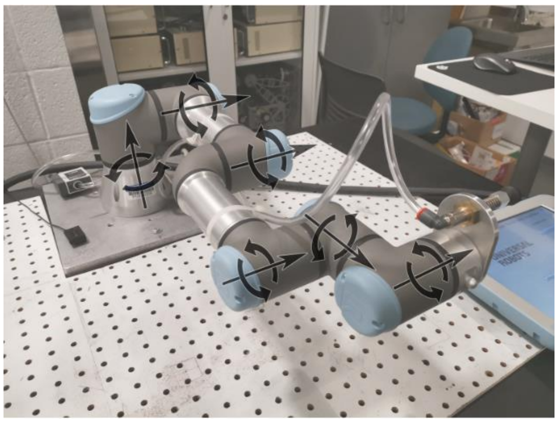


Figure 6 Rotation Axis

Since all of the joints are rotation joints, with ω and q for every joint, we can calculate the liner velocity by the following equation:

$$v = -\omega \times q$$

The resulting liner velocity goes follows:

Table 2 Angular and Linear Velocity

Joint#	ω	q	v
1	(0, 0, 1)	(0, 0, 0)	(0, 0, 0)
2	(0, 1, 0)	(0, 0.12, 0.152)	(-0.152, 0, 0)
3	(0, 1, 0)	(0.244, 0.12, 0.152)	(-0.152, 0, 0.244)
4	(0, 1, 0)	(0.457, 0.27, 0.152)	(-0.152, 0, 0.457)
5	(1, 0, 0)	(0.457, 0.11, 0.152)	(0, 0.152, -0.11)
6	(0, 1, 0)	(0.54, 0.11, 0.152)	(-0.152, 0, 0.54)

The screw axis can be calculated by concatenating ω and v for each joint in the following way:

$$S_i = \begin{bmatrix} \omega_i \\ v_i \end{bmatrix}$$

Since the ping pong balls are laid on the table, the same level as the robot base, they are relatively easier and more stable to using analytical method to calculate the inverse kinematics. The process of analyzing is generally the same as we have done in Lab 5.

In this environment, we are provided with the coordinates of the grip in world frame ($x_{wgrip}, y_{wgrip}, z_{wgrip}$). We can then calculate the θ_1 by adding an auxiliary line. By connecting the base frame origin and the center coordinate, we can calculate α using arccosine and calculate β using arctangent:

$$\alpha = arccos(\frac{b_{xy}}{l_{xy}})$$

$$\beta = arctan(\frac{y_{wgrip}}{x_{wgrip}})$$

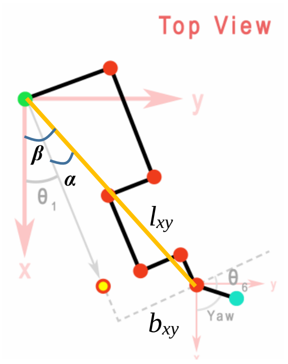


Figure 7 Top View of the Robot

By minus α from β, we can get θ_1:

$$\theta_1 = \beta - \alpha$$

θ_6 does not matter since the end effector is a Jacohand, thus we simply assigned it with an arbitrary value:

$$\theta_6 = \theta_1 + 90°$$

To calculate the other angles, we first need to set an auxiliary point—the project end point, (x_{3end}, y_{3end}, z_{3end}), and express it in base frame. We can achieve that by first expressed in the end effector frame. The center-point frame and the project end point are shown as in fig.8.

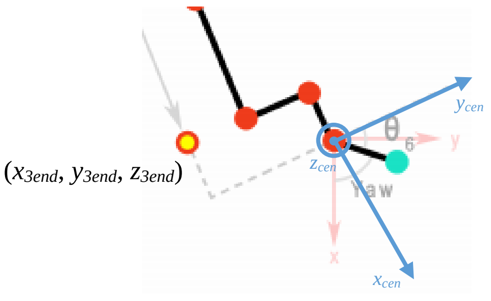


Figure 8 Robot End Effector Frame

Then, we can express the project end point in base frame by multiplying the expression in end effector frame with a translation matrix from base frame to end effector frame. The translation matrix goes as follows:

$$T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & x_{wgrip} \\ \sin(\theta_1) & \cos(\theta_1) & 0 & y_{wgrip} \\ 0 & 0 & 1 & z_{wgrip} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Similarly, for the sake of simplicity, we set the value of θ_5 to $\frac{\pi}{2}$ as a constant.

To calculate the rest of the angles, we need to add one auxiliary line and to calculate them. The way to add auxiliary line is shown in fig.9.

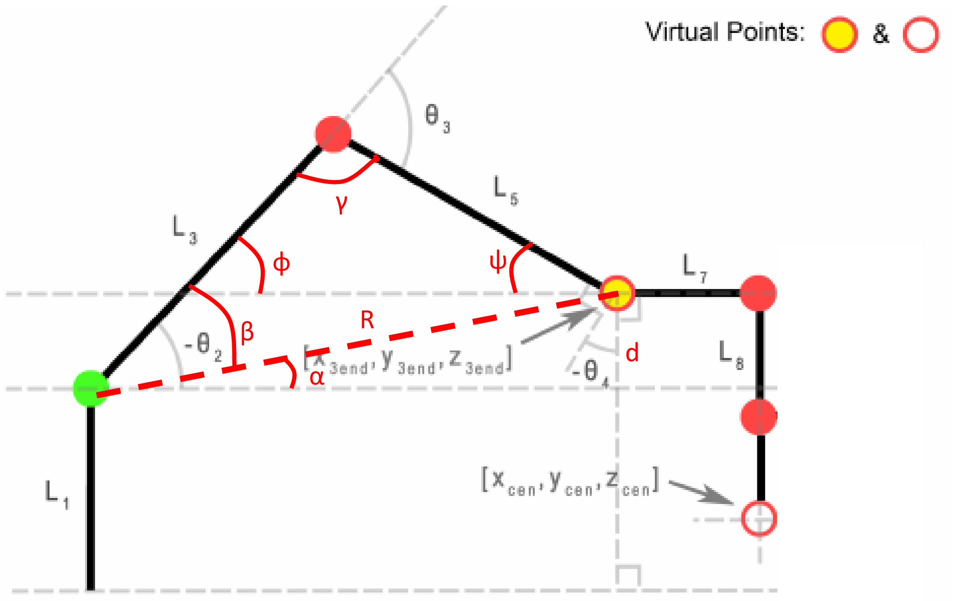


Figure 9 Calculation of Angles

With (x_{3end}, y_{3end}, z_{3end}) known, we can easily calculate d through equation:

$$d = z_{3end} - L_{01}$$

Using Pythagorean Theorem, we can calculate R:

$$R = \sqrt{x_{end}^2 + y_{end}^2 + d^2}$$

We can then use trigonometry to calculate α, which is a part of θ_2:

$$\alpha = arcsin(d/R)$$

Knowing L_{03}, L_{05} and R, we can calculate β and γ using Cosine Law:

$$\beta = \frac{R^2 + L_{03}^2 - L_{05}^2}{2R \times L_{03}}$$

$$\gamma = \frac{L_{03}^2 + L_{05}^2 - R^2}{2L_{05} \times L_{03}}$$

Then we can easily calculate that:

$$\theta_2 = -(\alpha + \beta)$$

$$\theta_3 = \pi - \gamma$$

With the property of parallel lines, we know that:

$$\varphi = -\theta_2$$

Also, by inspection of the relationship between angles, we can calculate that:

$$\psi = -\theta_4$$

Therefore, we can calculate θ_4 through equation:

$$\theta_4 = -\psi = -(\pi - \gamma - \varphi) = -(\theta_2 + \theta_3).$$

Hence, with the expected end effector position in world frame provided, we can now calculate angles we need to reach our goal.

5.3 Planning

Throwing the ball in simulation is more difficult than we thought. We initially encountered huge problems regarding picking the ball and releasing the ball at the right time. Through researching online, we ended up using a pseudo picking and throwing mechanism. The idea is pretty straightforward. When the robot is about to pick the ball, we set Jacohand as the parent of the ball. When the ball should be released, we simply detached this parent-child pair. In this way, we can overcome the difficulty of using a simulator and also control precisely the release time because the ball will just inherit the moving speed of the robot arm. A short snippet of code will be provided below:

```
close_gripper()
#Perform a pseudo grip
vrep.simxSetObjectParent(clientID, sphere_handle, force_handle, True ,vrep.simx_opmode_oneshot)
vrep.simxSetObjectPosition(clientID, sphere_handle, force_handle, (0,0,0), vrep.simx_opmode_blocking)
```

Since the trajectory of the ball will be a parabola, and the robot is going to perform an oblique projectile motion, we need to derive an equation relating the horizontal distance traveled to the projectile angle and the initial velocity. The detailed derivation is provided as follows:

We use two annotated auxiliary graphs to aid the derivation:

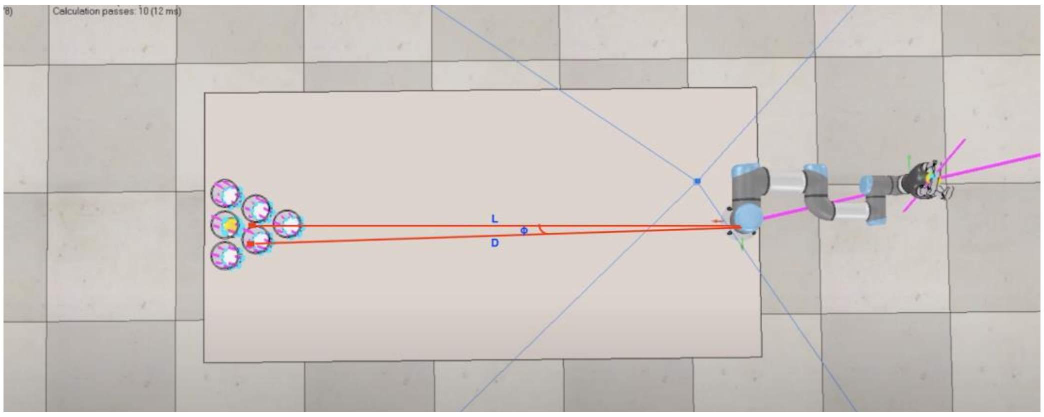


Figure 10 Robot Configuration When Releasing the Ball (Top View)

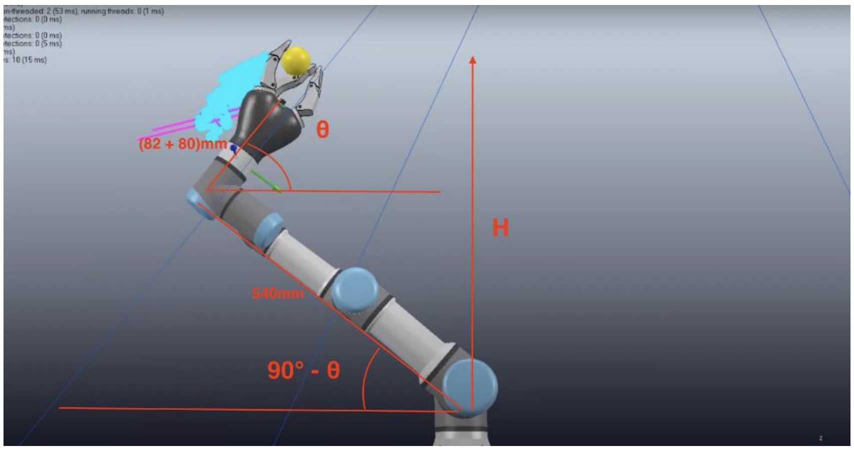


Figure 11 Robot Configuration When Releasing the Ball (Side View)

As illustrated in the above graphs, is the horizontal projectile angle, H is the vertical distance, D is the horizontal distance, is the base angle of the robot, and v will be used in the later derivation as the initial velocity of the ball.

Referring to the fundamental formulas of kinematics, we obtained the following equations:

$$-H = V \sin\theta t - \frac{1}{2}gt^2 \tag{1}$$

$$t = \frac{-2V\sin\theta - \sqrt{\delta}}{-2g} \tag{2}$$

$$\delta = 4V^2(\sin\theta)^2 + 8gH \tag{3}$$

$$D = V\cos\theta \times \frac{V\sin\theta + \frac{1}{2}\sqrt{\delta}}{g} \tag{4}$$

After the simplification, the final equation is:

$$v^2 = \frac{D^2 g}{3gH^2 + \frac{\sqrt{3}}{2}D} \tag{5}$$

Referring to the two attached auxiliary graphs above, we can calculate D and H as follows:

$$H = \sin(\frac{1}{2}\pi - \theta) \times 540mm + \sin\theta \times (82mm + 80mm) \tag{6}$$

$$D = \frac{L}{\sin\phi} \tag{7}$$

We obtained the two values 540mm and 82mm based upon the following UR3 specifications:

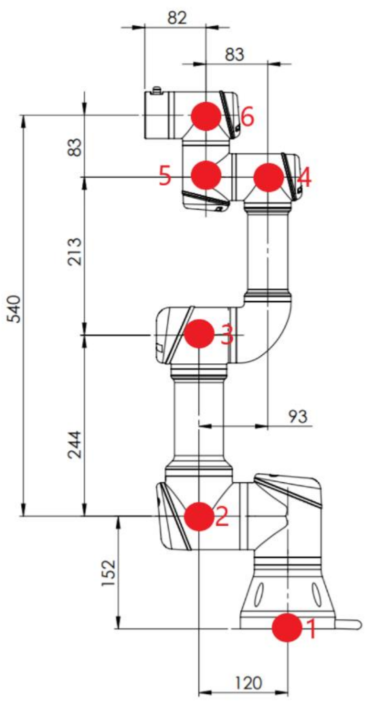


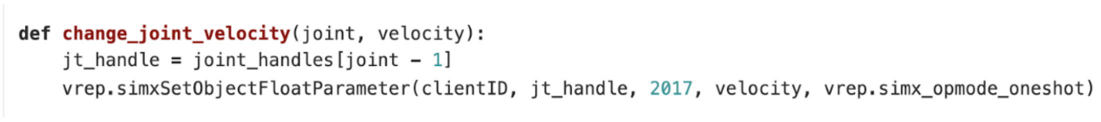
Figure 12 Robot Dimension

The length 80mm is brought from the Jacohand.

In order to reduce the number of variables and facilitate the computation, we could choose to fix either the initial projectile angle at 30 degrees or fix the initial velocity. We used the latter approach for simplicity because it’s easier to set the joint velocity.

We didn’t think of a proper way to accurately measure. Therefore, we determined this value for different cups through trials and errors. In the example provided in fig.10, the base angle will be 0.555π. L, by our measurement, will roughly be 1.7m. Therefore, for this example, the release angle will be roughly 0.724*π.

How we set the joint velocity is illustrated in the following code snippet:



```
def change_joint_velocity(joint, velocity):
    jt_handle = joint_handles[joint - 1]
    vrep.simxSetObjectFloatParameter(clientID, jt_handle, 2017, velocity, vrep.simx_opmode_oneshot)
```

6 Experiment Setup

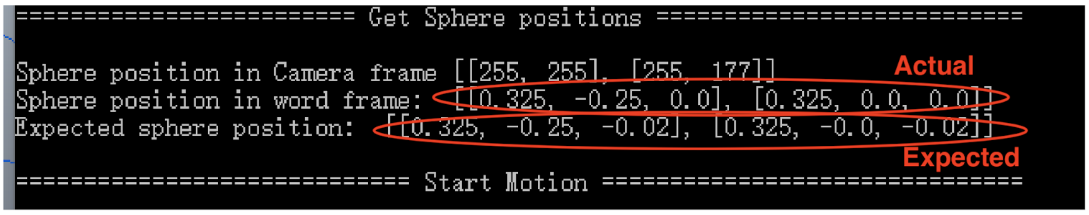
As describe in the name, our robot needs to play beer-pong as human does—pick up the ping pong and throw it into the cup. To be more specific, the robot need to find the ball placing randomly by the camera sensor, calculate the inverse kinematics to reach the ball; and calculate the trajectory to throw the ball. The environment setup is discussed in the Environment section. With all the motion and planning functions implemented, we start the experiment by testing the tasks one by one.

We first tested if the vision sensor can find the ball placing in the random place, and the sensor worked perfectly—the sensor can always find multiple balls and return their positions in world frame which can be verified from comparing their true values get from the simulator. The error was controlled in 2%. The experiment data will be provided in the next section. However, we found that randomly placed balls may fall outside of the work space and making the inverse kinematics fail. Therefore, we set the balls in fixed location in the following experiments for the sake of stability and controlling variables.

When we testing throwing the ball, we noticed that the timing for the Jacohand to release is the most critical variable that cannot be calculated and needs to be tune. Jacohand will release as soon as it receive the command “open_gripper()”, however, it takes time for the UR3 robot to move to the desired position and reached the ideal velocity. To control the robot using Python, we need to have the Jacohand wait before releasing the ball. In order to find the perfect timing, we sweep through a range of possible time and observe a trial throw. With multiple trails of experiments, we found that 0.5 seconds is a reasonable to wait.

7 Data and Results

It’s easy to qualitatively measure the functionality of our forward kinematics, ball detection, and inverse kinematics. We wrote a testing code in our development process to verify our implementation by printing the expected values and the actual values. It’s always easy to get the correct values in a simulator. Therefore, the expected values are provided by the data “getter” of the Vrep simulator (by calling the simxGetObjectPosition function on the target object), and the actual values are the current position of the robot after our implemented functions (by calling the simxGetObjectPosition function on the robot end-effector). Examples and data for the verification are provided in the following screenshots:

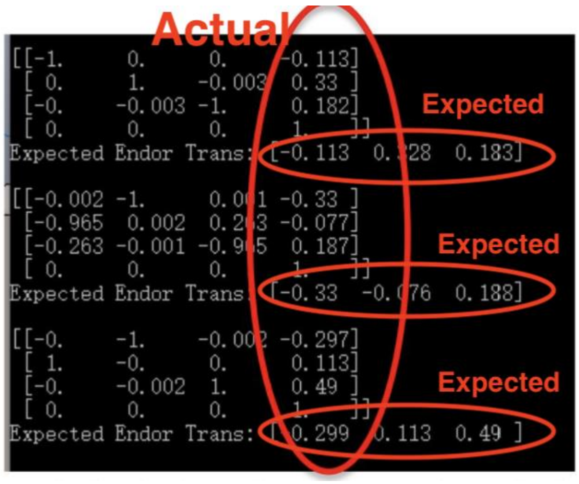


```
========================= Get Sphere positions ===========================

Sphere position in Camera frame [[255, 255], [255, 177]]
Sphere position in word frame: [[0.325, -0.25, 0.0], [0.325, 0.0, 0.0]]
Expected sphere position: [[0.325, -0.25, -0.02], [0.325, -0.0, -0.02]]

============================== Start Motion ===============================
```

Figure 14 Ball Detection Verification

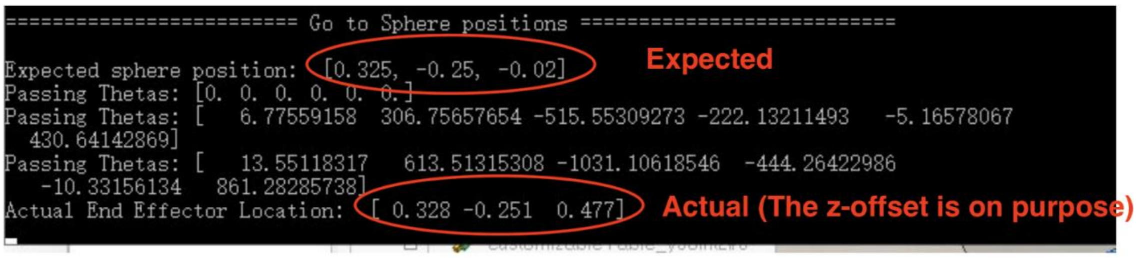


```
[[-1.      0.      0.     -0.113]
 [ 0.      1.     -0.003   0.33 ]
 [-0.     -0.003 -1.       0.182]
 [ 0.      0.      0.      1.   ]]
Expected Endor Trans: [-0.113  0.328  0.183]

[[-0.002 -1.      0.001 -0.33 ]
 [-0.965  0.002  0.263 -0.077]
 [-0.263 -0.001 -0.965  0.187]
 [ 0.      0.      0.      1.   ]]
Expected Endor Trans: [-0.33  -0.076  0.188]

[[-0.     -1.     -0.002 -0.297]
 [ 1.     -0.      0.      0.113]
 [-0.     -0.002  1.      0.49 ]
 [ 0.      0.      0.      1.   ]]
Expected Endor Trans: [-0.299  0.113  0.49 ]
```

Figure 13 Forward Kinematics Verification



```
========================= Go to Sphere positions ==========================

Expected sphere position: [0.325, -0.25, -0.02]
Passing Thetas: [0.  0.  0.  0.  0.  0.]
Passing Thetas: [   6.77559158  306.75657654 -515.55309273 -222.13211493   -5.16578067
  430.64142869]
Passing Thetas: [   13.55118317   613.51315308 -1031.10618546  -444.26422986
   -10.33156134   861.28285738]
Actual End Effector Location: [ 0.328 -0.251  0.477]
```

Figure 15 Inverse Kinematics Verification

For the final result, it’s harder to measure it quantitatively but qualitatively. After all, we can call a throw “good” if the ball actually goes into the cup. We tried nearly 100 throws into different cups. Among these trials, the ball goes into the cup roughly **75%** of the time. The thing that confused us is that when the ball misses, the error is very big. After researching online, we found many people using the Bullet physical engine encountered similar problems. The reason seems to be that the amount of the computational resources allocated for a physical engine varied and fluctuated greatly, and the performance of the engine is directly related to it. **Theoretically, we can solve this problem by disabling the dynamics and the physical engine. In this way, we can still produce a seemingly good simulation. However, we want to approximate real-world practices as much as possible.**

8 Conclusion

We’re pretty satisfied with the result we achieved in doing this final project. Accomplishing this project gets us more familiarized with Robotics and its application. Concretely, our understanding of perception and sensing, kinematics, spatial frame transformation, and trajectory generation are all deepened in the process. However, there’re still many potential improvements. First, we could make the trajectory generation more “intelligent.” In our current setting, the positions for the six cups are fixed instead of randomly located on the table. A potential future modification would be to also randomize the positions of the cups and make our trajectory calculation algorithm more intelligent. Second, since our robot is for entertainment purposes, we intend to add a voice control feature to the robot. In fact, we have already finished the coding and the setup of the voice recognition part and its interface with the robot. We used AWS IoT and Alexa for this feature. However, we have not tested it yet, and we’re very excited to get it fully implemented in the future.