Monika Spytek (mspyte3) and Seung Hyun Shin (sshin14)
Frat Daddy
https://github-dev.cs.illinois.edu/mspyte3/ece470
https://youtu.be/_PW6flCHniM

Abstract

Our goal for this project was to create a simulation of a robotic arm that could play beer pong. Being in our last year of college, a beer pong robot makes for an entertaining concept and handling projectiles in our decision-making process is an interesting alternative to the more common sorting or path planning robotics problems. Many of the difficulties we encountered were centered around interfacing with the simulator. Throughout this project, we practiced a lot of dynamic problem solving as we needed to reevaluate our methods to gather data and train the robot.

Ultimately, we judged the success of our robot based on both how well it could play a full game of beer pong and how reliably each shot can make it into the desired cup outside of a game. We tested the reliability of each input and were able to find inputs that consistently landed in about 7 of the 10 cups. The few cups that we don't have reliable inputs for often require that other cups are still in play so that the ping pong can bounce off of them. With our current dataset, the robot can usually eliminate about 7 or 8 cups within 50 shots, including the shots where the simulation has an error when lifting the ball. We consider this fairly successful, especially as we continue to train the robot more and understand that most of the missed shots are due to errors in our simulation and not errors in our data.

Introduction

During the semester, our main goals included learning to use the simulator to move the robotic arm and throw a ball through a python script, find input combinations that could land the ball in one of 10 cups, and then formulate a way to gather inputs for every cup and choose inputs during a game of beer pong. Some of our key inspiration for designing a robot that works with projectiles was based off of knowing that most of robotics tends to be focused on classifying and path planning. Projectiles tend to be very difficult to implement because there are so many outside factors, key ones being air resistance or crosswinds [1]. Still, the precision required and the opportunity to train the robot to aim for different cups was tempting. Google's TossingBot was an interesting, more advanced robotic arm that continued to spark our interest in the challenge of handling projectile robotics [7].

The beginning of designing our robot was to learn to work with CoppeliaSim. We used portions of several tutorials to learn how to handle sensors, attach end effectors, and use a remote API. When first learning to read sensors, we used youtube tutorials and when trying to interface with python we used tutorials found through CoppeliaSim, along with extensively referencing Coppelia's remote API documentation and online forums [2][3][5][6].

After being able to control the robot fairly well, we dove into training our robot to aim and shoot for certain cups. This ended up being a much more difficult task than we anticipated because, even in simulation, we dealt with a lot of factors that made each shot inconsistent. We

spent a long time researching neural networks but weren't able to create one that would return reliable inputs for the robot to aim and shoot with [4]. Eventually, we settled on a more brute-force method for gathering potential inputs and created a simpler method for weighting each input and choosing the one most likely to work.

Method

Our robotic arm uses forward kinematics to move to lift a ping pong ball on the table. The gripper of the arm doesn't actually physically lift the ball. We chose to simulate lifting the ball by attaching the ball to a point between the grips so that we could circumvent meticulously tuning the gripper's variables so that it would apply the right amount of force to grip the ball. This choice was made after researching forums online that promised that tuning the gripper's variables is tedious and produces inconsistent results. We also already introduce some error into our program because sometimes the gripper will bump into the ping pong when it goes to lift it and make it roll out of place.

Next, we decided to minimize the task of throwing a ball to two input parameters: an amount to rotate the base of the robot and a velocity to launch the ball with. We also simulated the process of releasing the ball at a velocity. Since our method of lifting the ball meant that the ball stopped being a dynamic object while the robot is holding it, we couldn't naturally launch the ball. Instead we disconnected the ball from its parent object (the arm) and gave it an initial horizontal velocity. In a real application, we would need to convert the velocity input into a speed and distance to move the end of the arm at before releasing the ball. Figure 1 shows a block diagram of the actions and decision making process of our robotic arm and Figure 2 shows the portion of code used to throw the ball.

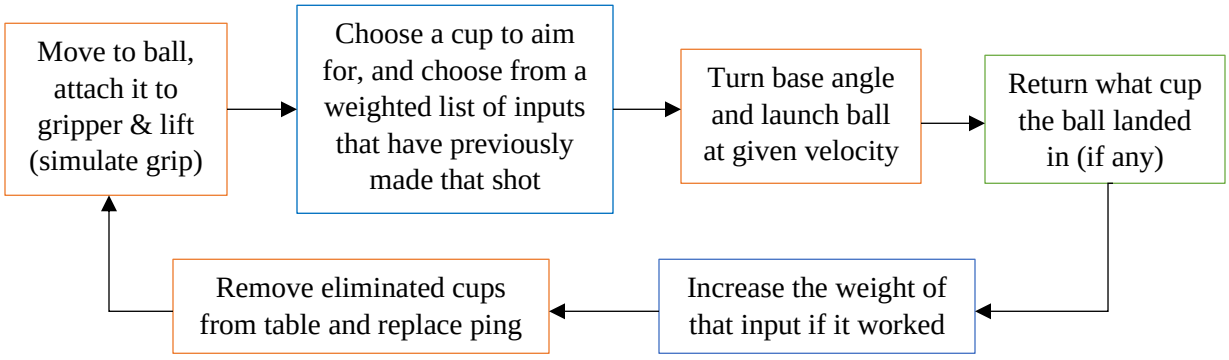

Figure 1: Block diagram of robotic system.

```
# THROW BALL ---------------------------------
    sim.simxSetJointTargetPosition(clientID, jointHandle[0],
        math.radians(baseAngle), sim.simx_opmode_oneshot)  # turn base
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[3],
        math.radians(80), sim.simx_opmode_oneshot)  # wrist back
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[3],
        math.radians(-5), sim.simx_opmode_oneshot)  # wrist forward
    time.sleep(.5)
    error_code, pos = sim.simxGetObjectPosition(clientID, ballHandle, -1,
        sim.simx_opmode_blocking)  # gets position of ball in world frame
                                       just before assigning velocity

    sim.simxPauseCommunication(clientID, True)  # pauses communication so we
                               can send multiple pieces of data at once
    sim.simxSetObjectParent(clientID, ballHandle, -1, True,
        sim.simx_opmode_oneshot)  # disconnect ball from parent
    sim.simxSetObjectPosition(clientID, ballHandle, -1, pos,
        sim.simx_opmode_oneshot)  # "resets" ping pong to given location in
                                      the world frame (-1)
    sim.simxSetObjectFloatParameter(clientID, ballHandle, 3000,
        ballVelocity, sim.simx_opmode_oneshot)  # throws ball with given
                                                velocity in the x direction
    sim.simxPauseCommunication(clientID, False)  # sends data again
```

Figure 2: The code used to throw the ping pong in fratDaddy_control.py.

Experimental Setup

To train our robot to make each shot, we set up 10 cups with proximity sensors inside each one so that the program returns which cup the ball landed in, if any. We initially wanted to create a neural network that would guess an angle to turn the base and a velocity to throw the ball at, given the location of the cup in the world frame. This ended up not working for multiple reasons, including that we had a very small training set and that our program only outputted if the ball made it in a cup but where the ball landed when it missed. Since the neural network had so little information to train off of, we didn't see much success and decided to use a brute-force method to gather datapoints.

We intuitively set a minimum and maximum range for the base to turn and for the velocity the ball should be launched at, and we randomly chose a value in each range and recorded which cup that input pair landed the ball in, if any. We ran our brute-force method for gathering input pairs that worked for about 20 hours total and, because of how long it takes to read all 10 sensors, only tried about 2,000 input combinations. We also replaced our neural network concept with a simpler reinforcement system for reliable inputs. Once we had a decent amount of input combinations that had worked at least once before, we ran several full games of beer pong. In a game, the robot randomly chooses a cup to aim for and selects an input for that cup from a weighted list of inputs that have worked at least once before. If that shot works, the weight on it is increased and the cup is removed from the table. Doing this weeds out inputs that only work when bouncing off other cups on the table and inputs that worked because of errors

with the ball not being gripped correctly. Figure 3 shows some of the functions used when collecting data and when playing the full game. shootRandomly() is used to select an angle to turn the base and a velocity to give the bar from within our intuitive range and shootForRandomCup() is used to select a random cup of the cups still in play and then choose from its weighted list of potential successful inputs. addToShots(…) is used to add an input pair to a cup if it has not been used to make that shot before. This function is most useful when collecting datapoints but sometimes useful when playing the full game because some shots that only landed in a certain cup due to an error with gripping the ball work correctly for landing the ball in a different cup.

```
# cup holds all the input pairs that have worked for that select cup
# cupcount holds weights for each input
# cupindex helps to choose a random index of the inputs for that cup

def addToShots(cup, cupcount, cupindex, pair):
    if pair not in cup:     # if this pair hasn't been recorded as making
                               this shot before then add it
        cup.append(pair)
        cupcount.append(1)
        cupindex.append(len(cupindex)-1)
    else:            # if this shot has been used before, increase its
                        reliability
        shot = cup.index(pair)
        cupcount[shot] += 1

def shootForRandomCup():
    # select a cup that is still in play
    cup = 0
    pair = []
    while cup == 0:
        i = random.randrange(10)  # random index
        cup = cups_left[i]  # choose random cup to aim for among cups left
                               in play
        if not np.any(allcounts[i]):        # if cupcount for this cup is
                                              all zeros
            pair = random.choice(allshots[i])  # random [baseAngle,
                                                  ballVelocity] for given cup
        else:
            norm = sum(allcounts[i])     # find sum of all elements in list
            normalized = []
            for element in allcounts[i]:        # normalize each element to
                                                    make it a probability
                normalized.append(element/norm)
            # select a shot using the weights accumulated
            pairIndex = np.random.choice(allshotindex[i], p=normalized)
            pair = allshots[i][pairIndex]
    return pair

def shootRandomly():
    baseAngle = random.uniform(-100, -55)
    ballVelocity = random.uniform(-6.5, -5)
    return [baseAngle, ballVelocity]
```

Figure 3: The code used to randomly select inputs when collecting data or playing the full game.

Data and Results

When collecting data with random input combinations, we ran our simulation for about 20 hours. As stated earlier, because of the speed of the simulation, this only tried about 2,000 combinations including shots that were ruined because the gripper bumped the ball out of place. Still, during this 20 hour period, we gathered about 80 inputs that landed the ball in one of 10 cups. Some cups tended to have many more inputs than others, which is most likely due to the range we chose to randomly test. With our 80 inputs, our robot can consistently land the ball in 7 of the 10 cups when only that cup was on the table. The last 3 cups still have 2 to 4 input pairs that worked at some point during our testing but, to make it in, the ball needs to bounce off other cups. This makes finishing a full game difficult because our robot selects a cup to aim for randomly, so there is no guarantee that the cups needed are left on the table for our difficult shots.

Figure 4 plots data points for two cups that had collected the most successful inputs and charts them as the magnitude of the velocity vs the degrees to turn the base left or right. The top corner shows the labeling system we used for our cups, and it can be seen that cup21 is closer to the robot than cup42 but they are directly in line with each other. Intuitively, this means that we expect the velocity needed for cup21 to be less than for cup42 but for the difference in angle to be small, and that is backed up by Figure 4. The small difference in angle is because the location of the ball when it is lifted is not centered on the table. From Figure 4, we see that cup42 is much more closely clustered than cup21. Part of this is because cup42 has twice as many datapoints but also because more of cup42's input pairs land directly in the cup. Some of cup21's input pairs bounce off other cups or off the rim of cup21 before settling. The same explanation can be used for the one clear outlier for cup42. In fact, Figure 5 shows a weighted chart of just cup42's input shots and the outlier only ever worked once.

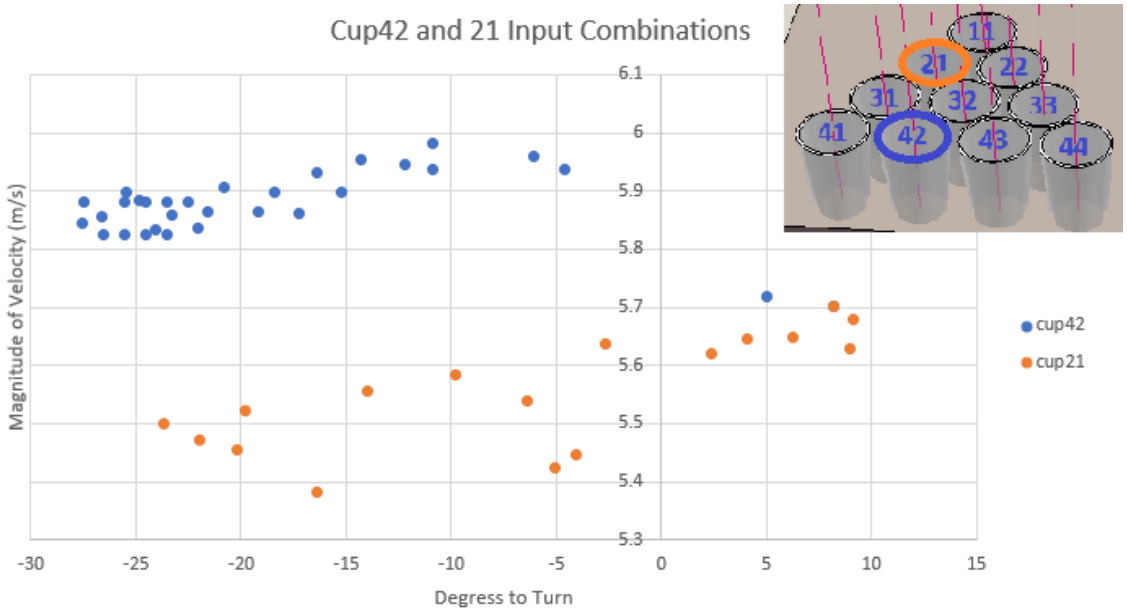

Figure 4: Input combinations that worked for cup42 and cup21.

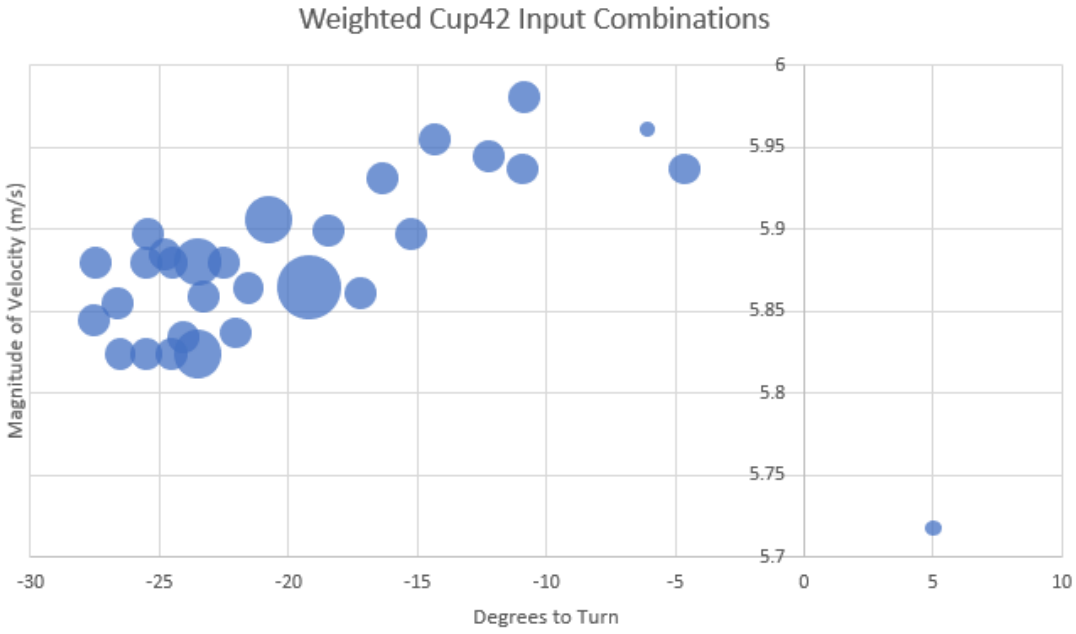

Figure 5: Weighted input combinations for cup42.

Further analyzing Figure 5, it becomes more clear that the inputs that worked best tend to be clustered in one area and less of a streak (as seen in Figure 4). Since cup42 has so many input combinations, the weights for each input is pretty even. For some cups with less inputs to try in a full game, only one or two inputs consistently works without bouncing off of other cups and so there is a larger difference in weights. For instance, cup11 only has 4 inputs so far and one of them is 5 times more likely to work than the other 3.

We ran each full game for about 50 shots and are consistently able to eliminate about 7 or 8 cups in that time. Our robot hasn't been able to complete a full game in 50 shots, primarily because of the few cups that we don't have enough reliable inputs for. Gathering more data would make a full game more possible but we would also need to fix the issue where the gripper bumps the ball and moves it out of place. The bump bug happens fairly often and our simulation doesn't have a way to tell us when that happens so we don't have an automated way to know how often that interferes with our game but from visual inspections it happens once or twice for every 5 shots. That means that at the moment our robot has about 15% accuracy, but if we fix the bump bug we would expect closer to 23% accuracy. This is based on the assumptions that right now it takes exactly 50 shots to eliminate 7 cups but fixing the bug would make it take 20 shots less to eliminate the same 7 cups. We would be able to further increase our accuracy by finding inputs that consistently work for our 3 difficult cups because it is also likely that many of the missed shots are shots aiming for one of the 3 difficult cups and not working. From there, the accuracy would be nearly perfected with enough training games to increase the weights on the most reliable inputs for each cup.

Conclusion

In conclusion, it could be said that we have successfully created a robot that plays beer pong considerably well and has the potential to successfully finish the game. The initial goal when starting this project was to teach the robot how to play and finish a game of beer pong, and we have successfully achieved that goal, especially given more time to find more reliable inputs.

From looking at the results of our data, because some of the cups have more reliable success rate in having the ping pong being landed than the others, it is our belief that all of the cups should have a consistent success rate given the correct adjustments in their variables. Had we more time and knowledge in the neural networks, a possible improvement could have been changing not only the base angle but all of the angles of the robot arm, because such an implementation would have provided it with more degree of freedom to successfully place the ball in to the cup with higher probability and hopefully decrease the bump bugs.

Another feature that we haven't had the chance to implement was the bounce rule. In a typical game of beer pong, the player also has the freedom of bouncing the ball on the table before landing it in the cup. This feature was overlooked because it was difficult enough to simply have the ping pong land in a cup in projectile motion alone.

Working on this project helped us to realize both the necessity and helpfulness of a simulator such as the CoppeliaSim, and also the difficulty of reenacting a real life situation with it at the same time. Everything from setting the dynamic properties of a sphere shape to represent a ping pong ball and programming a gripper that would grab that ball to toss it into a designated cup required a lot of research which tended to lead to more research. Although setting up the environment of the game of beer pong was difficult, had it not been for the simulator, it would have been impossible for us to run the data collection for 20 hours and collect data as we did. Overall, the project really helped us to learn to appreciate simulators and to think intuitively on topics covered in ECE 470.

References

Allain, Rhett. “Projectile Motion Primer for FIRST Robotics.” Wired, Conde Nast, 3 June 2017, www.wired.com/2012/01/projectile-motion-primer-for-first-robotics/.

“Enabling the Remote API - Client Side.” Enabling the Remote API - Client Side, www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm.

Iacobelli, Francisco. “Controlling Your Robot with Python.” Francisco Iacobelli's Academic Website, 9 Oct. 2018, fid.cl/courses/ai-robotics/vrep-tut/pythonBubbleRob.pdf.

Imran, Alishba. “Training a Robotic Arm to Do Human-Like Tasks Using RL.” Medium, Data Driven Investor, 3 Mar. 2019, medium.com/datadriveninvestor/training-a-robotic-arm-to-do-human-like-tasks-using-rl-8d3106c87aaf.

mechatronics Ninja. “Pick and Place Application with KUKA KR16 Robot Using v-Rep (CoppeliaSim) and Matlab.” Youtube, 20 Jan. 2019, www.youtube.com/watch?v=CVoV08T0Aqo.

“Regular API.” CoppeliaSim API, www.coppeliarobotics.com/helpFiles/en/apiOverview.htm.

Reichman, Ran. “TossingBot - Teaching Robots to Throw Objects Accurately.” Lyrn.AI, 3 Apr. 2019, www.lyrn.ai/2019/04/03/tossingbot-teaching-robots-to-throw-objects-accurately/.

Appendix

fratDaddy_control.py

```
import sim     # access CoppeliaSim elements
import numpy as np
import math
import time

def setJointPosition(clientID, thetas, jointHandle):
    # order of joint changes is currently optimized for not slapping the ball
        off the table
    sim.simxSetJointTargetPosition(clientID, jointHandle[0], thetas[0],
        sim.simx_opmode_oneshot)
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[1], thetas[1],
        sim.simx_opmode_oneshot)
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[3], thetas[3],
        sim.simx_opmode_oneshot)
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[5], thetas[5],
        sim.simx_opmode_oneshot)
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[4], thetas[4],
        sim.simx_opmode_oneshot)
    time.sleep(.5)
    sim.simxSetJointTargetPosition(clientID, jointHandle[2], thetas[2],
        sim.simx_opmode_oneshot)
    time.sleep(.5)  # wait for robot to move

def readCupSensor(clientID, handle):
    error_code, detectionState, detectedPoint, detectedObjectHandle,
        detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
        handle, sim.simx_opmode_blocking)
    time.sleep(1)
    if error_code != sim.simx_return_ok:
        raise Exception("could not read cups's sensor")
        stopSim(clientID)
    return detectionState

def stopSim(clientID):
    time.sleep(2)
    sim.simxStopSimulation(clientID, sim.simx_opmode_oneshot)
    sim.simxGetPingTime(clientID)  # guarantees that last command sent out
                                    has time to arrive
    sim.simxFinish(clientID)  # Close the connection to V-REP
    print("Simulation ended")

def shoot(clientID, baseAngle, ballVelocity, cups_left):
    # LABEL HANDLES FOR CONTROL ----------------------
    # Get handle for ping pong ball
    error_code, ballHandle = sim.simxGetObjectHandle(clientID, 'Ping_Pong',
        sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for ping pong')
```

```
        stopSim(clientID)

    jointHandle = np.zeros((6, 1))  # creates array of zeros, 1 row, 6
                                      columns
    error_code, baseHandle = sim.simxGetObjectHandle(clientID,
        'UR3_link1_visible', sim.simx_opmode_blocking)  # Get handle for base
                                                          of robot
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for base frame')
        stopSim(clientID)

    # Get handle for all joints of robot
    error_code, jointHandle[0] = sim.simxGetObjectHandle(clientID,
        'UR3_joint1', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for first joint')
        stopSim(clientID)
    error_code, jointHandle[1] = sim.simxGetObjectHandle(clientID,
        'UR3_joint2', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for second joint')
        stopSim(clientID)
    error_code, jointHandle[2] = sim.simxGetObjectHandle(clientID,
        'UR3_joint3', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for third joint')
        stopSim(clientID)
    error_code, jointHandle[3] = sim.simxGetObjectHandle(clientID,
        'UR3_joint4', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for fourth joint')
        stopSim(clientID)
    error_code, jointHandle[4] = sim.simxGetObjectHandle(clientID,
        'UR3_joint5', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for fifth joint')
        stopSim(clientID)
    error_code, jointHandle[5] = sim.simxGetObjectHandle(clientID,
        'UR3_joint6', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for sixth joint')
        stopSim(clientID)

    # Get handle for end-effector of robot
    error_code, endHandle = sim.simxGetObjectHandle(clientID,
        'UR3_link7_visible', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception('could not get object handle for end effector')
        stopSim(clientID)

    # Get gripper handles
    error_code, gripAttachPoint = sim.simxGetObjectHandle(clientID,
        'RG2_attachPoint', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception("could not get object handle for gripper's attach
                        point")
        stopSim(clientID)
```

```
    # REMOVE CUPS MADE ------------------------------
    for i in range(0, 10):
        if cups_left[i] == 0:
            if i == 0: handle = "Cup11"
            elif i == 1: handle = "Cup21"
            elif i == 2: handle = "Cup22"
            elif i == 3: handle = "Cup31"
            elif i == 4: handle = "Cup32"
            elif i == 5: handle = "Cup33"
            elif i == 6: handle = "Cup41"
            elif i == 7: handle = "Cup42"
            elif i == 8: handle = "Cup43"
            elif i == 9: handle = "Cup44"
            else: handle = ""
            error_code, cup = sim.simxGetObjectHandle(clientID, str(handle),
               sim.simx_opmode_blocking)
            if error_code != sim.simx_return_ok:
                raise Exception('could not get object handle for the cup to
                                remove')
                stopSim(clientID)
            error_code, cupPos = sim.simxGetObjectPosition(clientID, cup, -1,
               sim.simx_opmode_oneshot)
            sim.simxSetObjectPosition(clientID, cup, -1, [cupPos[0],
               cupPos[1], 0.075], sim.simx_opmode_oneshot)

    # MOVE ARM TO GRAB BALL -------------------------
    liftBallPosition = [math.radians(-90), math.radians(-35),
        math.radians(-80), math.radians(35), math.radians(90),
        math.radians(90)]
    error_code = sim.simxSetJointTargetVelocity(clientID, gripAttachPoint, 0,
        sim.simx_opmode_streaming)
    time.sleep(.5)
    error_code = sim.simxSetJointTargetVelocity(clientID, gripAttachPoint, 0,
        sim.simx_opmode_streaming)
    time.sleep(.5)
    setJointPosition(clientID, liftBallPosition, jointHandle)

    # GRAB BALL -------------------------------------
    # sim.simxSetObjectParent(clientID, ballHandle, gripAttachPoint, True,
        sim.simx_opmode_oneshot)  # makes the gripper the parent object to the
                                    ball (fakes picking it up)
    error_code = sim.simxSetJointTargetVelocity(clientID, gripAttachPoint, 0,
        sim.simx_opmode_streaming)
    time.sleep(.5)
    error_code = sim.simxSetJointTargetVelocity(clientID, gripAttachPoint, 0,
        sim.simx_opmode_streaming)
    time.sleep(.5)
    sim.simxSetObjectParent(clientID, ballHandle, gripAttachPoint, False,
        sim.simx_opmode_oneshot)

    # RETURN TO UPRIGHT POSITION --------------------
    throwPosition = [math.radians(-90), math.radians(0), math.radians(45),
        math.radians(45), math.radians(90), math.radians(90)]
    setJointPosition(clientID, throwPosition, jointHandle)
    # THROW BALL ----------------------------------
    sim.simxSetJointTargetPosition(clientID, jointHandle[0],
```

```
    math.radians(baseAngle), sim.simx_opmode_oneshot)  # turn base
time.sleep(.5)
sim.simxSetJointTargetPosition(clientID, jointHandle[3],
    math.radians(80), sim.simx_opmode_oneshot)  # wrist back
time.sleep(.5)
sim.simxSetJointTargetPosition(clientID, jointHandle[3],
    math.radians(-5), sim.simx_opmode_oneshot)  # wrist forward
time.sleep(.5)
error_code, pos = sim.simxGetObjectPosition(clientID, ballHandle, -1,
    sim.simx_opmode_blocking)  # gets position of ball in world frame just
                                   before assigning velocity

sim.simxPauseCommunication(clientID, True)  # pauses communication so we
                            can send multiple pieces of data at once
sim.simxSetObjectParent(clientID, ballHandle, -1, True,
    sim.simx_opmode_oneshot)  # disconnect ball from parent
sim.simxSetObjectPosition(clientID, ballHandle, -1, pos,
    sim.simx_opmode_oneshot)  # "resets" ping pong to given location in
                                   the world frame (-1)
sim.simxSetObjectFloatParameter(clientID, ballHandle, 3000, ballVelocity,
    sim.simx_opmode_oneshot)  # throws ball with given velocity in the x
                                   direction
sim.simxPauseCommunication(clientID, False)  # sends data again
time.sleep(1)

# INITIALIZE AND READ CUP SENSORS -------------
# Initialize cup's sensor, doing this earlier causes a notable decrease
    in accuracy for some reason
error_code, cupSensor11 = sim.simxGetObjectHandle(clientID,
    'Proximity_sensor11', sim.simx_opmode_blocking)
if error_code != sim.simx_return_ok:
    raise Exception("could not get object handle for cup11 sensor")
error_code, detectionState, detectedPoint, detectedObjectHandle,
    detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
    cupSensor11, sim.simx_opmode_blocking)
time.sleep(1)
if error_code != sim.simx_return_ok:
    raise Exception("could not read cup11 sensor")
landedIn11 = readCupSensor(clientID, cupSensor11)  # returns detection
                                                   state of prox sensor11

# Initialize cup21's sensor
error_code, cupSensor21 = sim.simxGetObjectHandle(clientID,
    'Proximity_sensor21', sim.simx_opmode_blocking)
if error_code != sim.simx_return_ok:
    raise Exception("could not get object handle for cup21 sensor")
error_code, detectionState, detectedPoint, detectedObjectHandle,
    detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
    cupSensor21, sim.simx_opmode_blocking)
time.sleep(1)
if error_code != sim.simx_return_ok:
    raise Exception("could not read cup21 sensor")
landedIn21 = readCupSensor(clientID, cupSensor21)  # returns detection
                                                   state of prox sensor

# Initialize cup22's sensor
error_code, cupSensor22 = sim.simxGetObjectHandle(clientID,
```

```
    'Proximity_sensor22', sim.simx_opmode_blocking)
if error_code != sim.simx_return_ok:
    raise Exception("could not get object handle for cup22 sensor")
error_code, detectionState, detectedPoint, detectedObjectHandle,
    detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
    cupSensor22, sim.simx_opmode_blocking)
time.sleep(1)
if error_code != sim.simx_return_ok:
    raise Exception("could not read cup22 sensor")
landedIn22 = readCupSensor(clientID, cupSensor22)  # returns detection
                                                   state of prox sensor

# Initialize cup31's sensor
error_code, cupSensor31 = sim.simxGetObjectHandle(clientID,
    'Proximity_sensor31', sim.simx_opmode_blocking)
if error_code != sim.simx_return_ok:
    raise Exception("could not get object handle for cup31 sensor")
error_code, detectionState, detectedPoint, detectedObjectHandle,
    detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
    cupSensor31, sim.simx_opmode_blocking)
time.sleep(1)
if error_code != sim.simx_return_ok:
    raise Exception("could not read cup31 sensor")
landedIn31 = readCupSensor(clientID, cupSensor31)  # returns detection
                                                   state of prox sensor

# Initialize cup32's sensor
error_code, cupSensor32 = sim.simxGetObjectHandle(clientID,
    'Proximity_sensor32', sim.simx_opmode_blocking)
if error_code != sim.simx_return_ok:
    raise Exception("could not get object handle for cup32 sensor")
error_code, detectionState, detectedPoint, detectedObjectHandle,
    detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
    cupSensor32, sim.simx_opmode_blocking)
time.sleep(1)
if error_code != sim.simx_return_ok:
    raise Exception("could not read cup32 sensor")
landedIn32 = readCupSensor(clientID, cupSensor32)  # returns detection
                                                   state of prox sensor

# Initialize cup33's sensor
error_code, cupSensor33 = sim.simxGetObjectHandle(clientID,
    'Proximity_sensor33', sim.simx_opmode_blocking)
if error_code != sim.simx_return_ok:
    raise Exception("could not get object handle for cup33 sensor")
error_code, detectionState, detectedPoint, detectedObjectHandle,
    detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
    cupSensor33, sim.simx_opmode_blocking)
time.sleep(1)
if error_code != sim.simx_return_ok:
    raise Exception("could not read cup33 sensor")
landedIn33 = readCupSensor(clientID, cupSensor33)  # returns detection
                                                   state of prox sensor

# Initialize cup41's sensor
error_code, cupSensor41 = sim.simxGetObjectHandle(clientID,
    'Proximity_sensor41', sim.simx_opmode_blocking)
```

```
    if error_code != sim.simx_return_ok:
        raise Exception("could not get object handle for cup41 sensor")
    error_code, detectionState, detectedPoint, detectedObjectHandle,
        detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
        cupSensor41, sim.simx_opmode_blocking)
    time.sleep(1)
    if error_code != sim.simx_return_ok:
        raise Exception("could not read cup41 sensor")
    landedIn41 = readCupSensor(clientID, cupSensor41)  # returns detection
                                                       state of prox sensor

    # Initialize cup42's sensor
    error_code, cupSensor42 = sim.simxGetObjectHandle(clientID,
        'Proximity_sensor42', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception("could not get object handle for cup42 sensor")
    error_code, detectionState, detectedPoint, detectedObjectHandle,
        detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
        cupSensor42, sim.simx_opmode_blocking)
    time.sleep(1)
    if error_code != sim.simx_return_ok:
        raise Exception("could not read cup42 sensor")
    landedIn42 = readCupSensor(clientID, cupSensor42)  # returns detection
                                                       state of prox sensor

    # Initialize cup43's sensor
    error_code, cupSensor43 = sim.simxGetObjectHandle(clientID,
        'Proximity_sensor43', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception("could not get object handle for cup43 sensor")
    error_code, detectionState, detectedPoint, detectedObjectHandle,
        detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
        cupSensor43, sim.simx_opmode_blocking)
    time.sleep(1)
    if error_code != sim.simx_return_ok:
        raise Exception("could not read cup43 sensor")
    landedIn43 = readCupSensor(clientID, cupSensor43)  # returns detection
                                                       state of prox sensor

    # Initialize cup44's sensor
    error_code, cupSensor44 = sim.simxGetObjectHandle(clientID,
        'Proximity_sensor44', sim.simx_opmode_blocking)
    if error_code != sim.simx_return_ok:
        raise Exception("could not get object handle for cup44 sensor")
    error_code, detectionState, detectedPoint, detectedObjectHandle,
        detectedSurfaceNormalVector = sim.simxReadProximitySensor(clientID,
        cupSensor44, sim.simx_opmode_blocking)
    time.sleep(1)
    if error_code != sim.simx_return_ok:
        raise Exception("could not read cup44 sensor")
    landedIn44 = readCupSensor(clientID, cupSensor44)  # returns detection
                                                       state of prox sensor

    isItInYet = [landedIn11, landedIn21, landedIn22, landedIn31, landedIn32,
                 landedIn33, landedIn41, landedIn42, landedIn43, landedIn44]

    stopSim(clientID)
```

```
    return isItInYet
```

fratDaddy_fullGame.py

```
import random, csv
import fratDaddy_control, sim
import numpy as np
import pyautogui, time

# CUP DATA ---------------------------------------
cups_left = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]      # 1 if still in play
# baseAngle and ballVelocity combinations that have landed in the
        corresponding cup
cup11 = [[-66.3, -5.35], [-64.067, -5.4778], [-65.401, -5.427],
        [-59.943, -5.792]]
cup21 = [[-66.3, -5.5], [-84.93, -5.424], [-99.145, -5.679],
        [-98.222, -5.701], [-69.809, -5.455], [-96.246, -5.649],
        [-87.379, -5.636], [-76.019, -5.557], [-80.193, -5.584],
        [-73.597, -5.381], [-92.386, -5.621], [-85.907, -5.448],
        [-68.025, -5.471], [-98.97, -5.629], [-83.602, -5.538],
        [-94.111, -5.646], [-70.167, -5.521]]
cup22 = [[-58.563, -5.979], [-59.183, -5.881], [-58.5, -6],
        [-59.943, -5.792], [-59.919, -5.782]]
cup31 = [[-93.23, -5.657], [-93.814, -5.716], [-88.565, -5.423],
        [-85.994, -5.407], [-82.141, -5.671], [-104.467, -5.824],
        [-87.379, -5.636], [-80.193, -5.584]]
cup32 = [[-60.542, -5.873], [-59.919, -5.782], [-59.989, -5.917],
        [-70.196, -5.352], [-60.5, -5.88], [-56.5, -6.15], [-98.222, -5.701]]
cup33 = [[-61.417, -5.628], [-80.856, -5.972], [-75.282, -5.456]]
cup41 = [[-59.943, -5.792], [-67.241, -5.0266], [-88.565, -5.423],
        [-85.994, -5.407]]
cup42 = [[-66.467, -5.824], [-83.938, -5.961], [-95.003, -5.718],
        [-74.752, -5.897], [-69.209, -5.906], [-72.783, -5.861],
        [-67.938, -5.837], [-79.154, -5.981], [-68.446, -5.864],
        [-85.367, -5.937], [-79.102, -5.937], [-66.691, -5.859],
        [-73.639, -5.931], [-77.794, -5.945], [-75.682, -5.955],
        [-71.573, -5.899], [-70.807, -5.865], [-65.181, -5.885],
        [-64.532, -5.897], [-63.349, -5.855], [-65.944, -5.834],
        [-62.454, -5.845], [-67.5, -5.88], [-66.5, -5.88], [-65.5, -5.88],
        [-64.5, -5.88], [-62.5, -5.88], [-65.467, -5.824], [-64.467, -5.824],
        [-63.467, -5.824], [-98.222, -5.701]]
cup43 = [[-55.45, -6.22], [-55.47, -6.22], [-59.183, -5.881],
        [-70.196, -5.352]]
cup44 = [[-75.282, -5.456], [-55.47, -6.22]]
allshots = [cup11, cup21, cup22, cup31, cup32, cup33, cup41, cup42, cup43,
        cup44]

# counts the number of times the corresponding shot has made it into that cup
cup11count = [1, 1, 5, 1]
cup21count = [1, 1, 2, 1, 2, 2, 1, 2, 1, 3, 2, 2, 2, 2, 4, 2, 2]
cup22count = [3, 2, 2, 4, 1]
cup31count = [3, 2, 1, 1, 4, 1, 2, 2]
cup32count = [3, 1, 5, 1, 2, 1, 1]
cup33count = [1, 3, 1]
cup41count = [1, 1, 2, 1]
```

```
cup42count = [3, 1, 1, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2,
              2, 2, 3, 2, 2, 2, 2, 2, 2, 1]
cup43count = [2, 1, 1, 2]
cup44count = [2, 1]
allcounts = [cup11count, cup21count, cup22count, cup31count, cup32count,
             cup33count, cup41count, cup42count, cup43count, cup44count]

# ugly way to get a weighted random index for selecting the next shot
cup11index = [0, 1, 2, 3]
cup21index = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
cup22index = [0, 1, 2, 3, 4]
cup31index = [0, 1, 2, 3, 4, 5, 6, 7]
cup32index = [0, 1, 2, 3, 4, 5, 6]
cup33index = [0, 1, 2]
cup41index = [0, 1, 2, 3]
cup42index = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
              18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
cup43index = [0, 1, 2, 3]
cup44index = [0, 1]
allshotindex = [cup11index, cup21index, cup22index, cup31index, cup32index,
             cup33index, cup41index, cup42index, cup43index, cup44index]

def addToShots(cup, cupcount, cupindex, pair):
    if pair not in cup:     # if this pair hasn't been recorded as making
                              this shot before then add it
        cup.append(pair)
        cupcount.append(1)
        cupindex.append(len(cupindex)-1)
    else:           # if this shot has been used before, increase its
                      reliability
        shot = cup.index(pair)
        cupcount[shot] += 1

def shootForRandomCup():
    # select a cup that is still in play
    cup = 0
    pair = []
    while cup == 0:
        i = random.randrange(10)  # random index
        cup = cups_left[i]  # choose random cup to aim for among cups left in
                              play
        if not np.any(allcounts[i]):        # if cupcount for this cup is all
                                              zeros
            pair = random.choice(allshots[i])  # random [baseAngle,
                                               ballVelocity] for given cup
        else:
            norm = sum(allcounts[i])     # find sum of all elements in list
            normalized = []
            for element in allcounts[i]:        # normalize each element to
                                                   make it a probability
                normalized.append(element/norm)
            # select a shot using the weights accumulated
            pairIndex = np.random.choice(allshotindex[i], p=normalized)
            pair = allshots[i][pairIndex]
    return pair

def shootRandomly():
```

```
    baseAngle = random.uniform(-100, -55)       # -100 to -55
    ballVelocity = random.uniform(-6.5, -5)
    return [baseAngle, ballVelocity]

print("MOVE CURSOR OVER PLAY BUTTON IN COPPELIASIM")
input("Press Enter to continue...")
x, y = pyautogui.position()


pyautogui.click(x, y)
time.sleep(2)

while np.any(cups_left):
    pair = shootForRandomCup()      # used for full game
    #pair = shootRandomly()     # used when gathering data
    print(pair)

    pyautogui.click(882, 66)
    time.sleep(2)

    # SET UP CLIENT ----------------------------------
    sim.simxFinish(-1)  # closes all opened connections
    # starts a connection
    clientID = sim.simxStart('127.0.0.1', 19999, True, True, 5000, 5)
    if clientID != -1:
        print("Connected to remote API server")
    else:
        print("Not connected to remote API server")

    sensors = fratDaddy_control.shoot(clientID, pair[0], pair[1], cups_left)
    time.sleep(2)

    for i in range(0, 10):      # if the sensor says that the ball landed in
                                    that cup, remove that cup from the game
        if sensors[i]:
            cups_left[i] = 0
            if i == 0: addToShots(cup11, cup11count, cup11index, pair)
            elif i == 1: addToShots(cup21, cup21count, cup21index, pair)
            elif i == 2: addToShots(cup22, cup22count, cup22index, pair)
            elif i == 3: addToShots(cup31, cup31count, cup31index, pair)
            elif i == 4: addToShots(cup32, cup32count, cup32index, pair)
            elif i == 5: addToShots(cup33, cup33count, cup33index, pair)
            elif i == 6: addToShots(cup41, cup41count, cup41index, pair)
            elif i == 7: addToShots(cup42, cup42count, cup42index, pair)
            elif i == 8: addToShots(cup43, cup43count, cup43index, pair)
            elif i == 9: addToShots(cup44, cup44count, cup44index, pair)
            else: print("why are we here?")

    if not np.any(cups_left):
        for i in range(0, 10):
            cups_left[i] = 1

print("ALL CUPS HAVE BEEN ELIMINATED")

# saves a file with all the cup inputs in case new inputs were found for cups
with few datapoints
with open('cuprun.csv', 'w') as myfile:
```

```
    wr = csv.writer(myfile, quoting=csv.QUOTE_ALL)
    wr.writerow([cup11])
    wr.writerow([cup11count])
    wr.writerow([cup21])
    wr.writerow([cup21count])
    wr.writerow([cup22])
    wr.writerow([cup22count])
    wr.writerow([cup31])
    wr.writerow([cup31count])
    wr.writerow([cup32])
    wr.writerow([cup32count])
    wr.writerow([cup33])
    wr.writerow([cup33count])
    wr.writerow([cup41])
    wr.writerow([cup41count])
    wr.writerow([cup42])
    wr.writerow([cup42count])
    wr.writerow([cup43])
    wr.writerow([cup43count])
    wr.writerow([cup44])
    wr.writerow([cup44count])
```