ECE 486: Lab 5 Report

Irene Boby (20873663), Young Ha Ahn (20838600)

July 25, 2024

Contents

Part 1: The End-Effector

We ran the provided script to move the Dobot manually and print the end-effector coordinates. We manually moved it to trace a random shape on the table and observed the following coordinates, noting the z-coordinate. As shown below, a z-coordinate of around -27 was the offset required to touch the table. The last three points observed in the output below (where the z-coordinate is around -4) was when the end effector was lifted to not touch the table's surface. We used these observations to offset the z coordinates in our planned path coordinates to draw the shapes.

```
Get new pose? Enter q to quity
272.71807861328125 , -6.426929950714111 , -27.253822326660156 , -1.3499940633773804 , 54.792301177978516 , 45.63317108154297
Get new pose? Enter q to quity
272.71807861328125 , -6.426929950714111 , -27.253822326660156 , -1.3499940633773804 , 54.792301177978516 , 45.63317108154297
Get new pose? Enter q to quity
272.71807861328125 , -6.426929950714111 , -27.253822326660156 , -1.3499940633773804 , 54.792301177978516 , 45.63317108154297
Get new pose? Enter q to quity
272.71807861328125 , -6.426929950714111 , -27.253822326660156 , -1.3499940633773804 , 54.792301177978516 , 45.63317108154297
Get new pose? Enter q to quity
272.71807861328125 , -6.426929950714111 , -27.253822326660156 , -1.3499940633773804 , 54.792301177978516 , 45.63317108154297
Get new pose? Enter q to quity
254.18556213378906 , 91.64514923095703 , -4.417182922363281 , 19.82647705078125 , 47.279850006103516 , 40.774662017822266
Get new pose? Enter q to quity
254.18556213378906 , 91.64514923095703 , -4.417182922363281 , 19.82647705078125 , 47.279850006103516 , 40.774662017822266
Get new pose? Enter q to quity
254.18556213378906 , 91.64514923095703 , -4.417182922363281 , 19.82647705078125 , 47.279850006103516 , 40.774662017822266
```

In the following parts where we draw using the points in the `path_planning.csv` file [4], we show that the offset is set so that the z-coordinate while drawing is -27.5 and to lift the pen, the z-coordinate is around -3.5.

Drawing a 6cm line

Using the observational knowledge gained above, we wrote the python code in [2] to draw a 6cm line on a piece of paper by giving the coordinates of the points and ensuring that the z-coordinates were -27.5. Figure 1 shows the line that the Dobot drew on the paper.

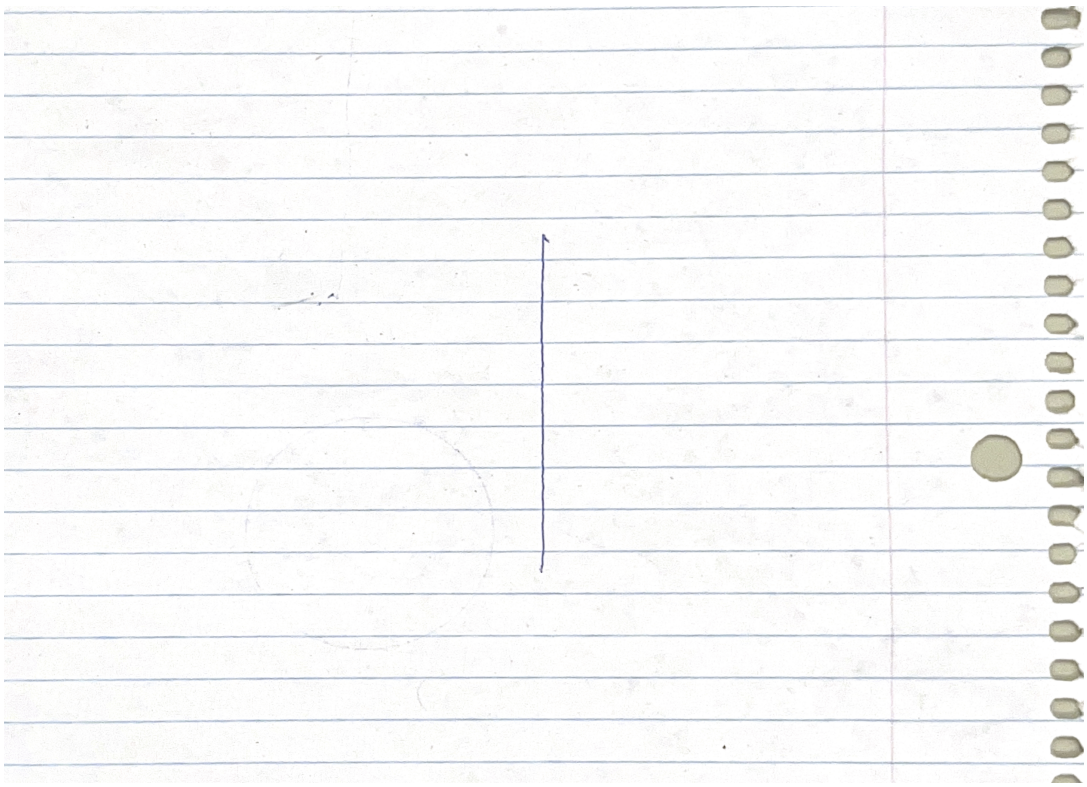

Figure 1: 6cm line drawn using Dobot IK

Part 2: The Shape File

Scaling Points

As described in Lab 4, the code in [1] scales the points to fit within a defined workspace, ensuring the robot could accurately draw them. The workspace limits for the x and y coordinates are x ranging from 150 to 250 and y from -50 to 50. The points are first read from the CSV file, and any connected shapes are identified based on a distance threshold of 0.25 units. Each shape is then processed to fit within the workspace boundaries.

To scale the points, the minimum and maximum x and y values from the shapes were determined. These values were used to calculate the scaling factors for both the x and y dimensions, ensuring the shapes would be proportionally resized to fit within the workspace. The minimum scaling factor was selected to maintain the aspect ratio and avoid distortion. Additionally, offsets were computed to translate the scaled points so they would align correctly within the workspace limits. Finally, the localized shapes were written to a new CSV file, adding a z-coordinate of 0 to indicate they lie on the base plane. This approach ensured that the robot could draw the shapes accurately within its designated working area. This CSV was then modified to add more intermediate points to lift the pen and provide smoother drawings as outlined below.

Figure 2 shows the localized points in the XY frame that the Dobot has to draw.

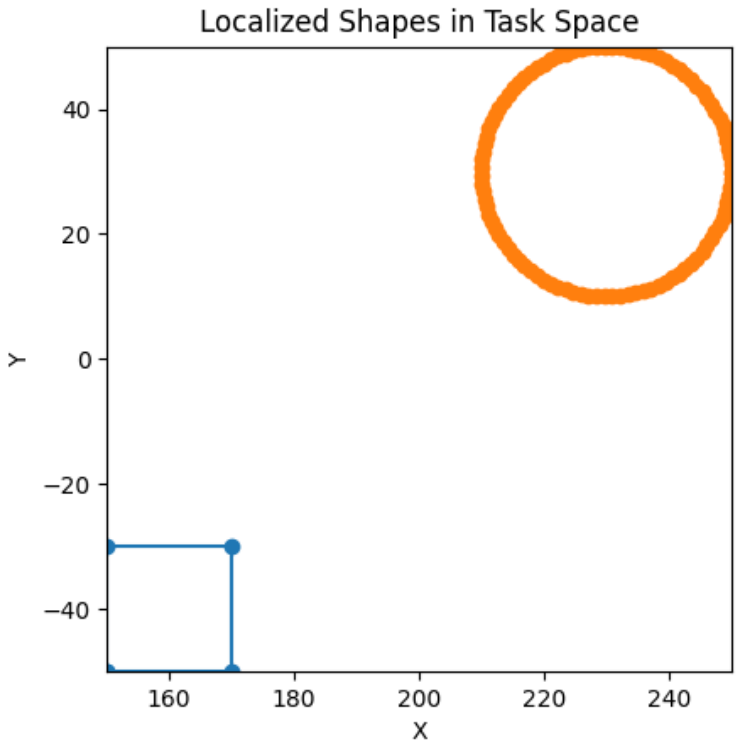

Figure 2: Shapes drawn by the scaled and localized points in the XY frame of the base frame

Moving Points

More transition coordinates were added to scaled and localized points to lift the pen as well as give more smoother transitions within the shapes. For the square, initially the points in the path were just the corners of the square. We added the following midpoints of the sides as well.

```
    150.0,-40.0,0
    160.0,-30.0,0
    170.0,-40.0,0
    160.0,-50.0,0
```

To lift the pen after the square is done and bring the pen back down to draw the circle, we added the points (150.0,-50.0,30) and (250.0,30.0,30) between the last point of the square and the first point of the circle.

```
    150.0,-50.0,0
    150.0,-50.0,30
    250.0,30.0,30
    250.0,30.0,0
```

We added the point (150.0,-50.0,30) to lift the pen before moving to the first point in the circle. The next point in the path is (250.0,30.0,30) which was added so that the Dobot moves in a straight line to 30mm above the start of the circle. It then moves back down to the first point of the circle to draw.

Drawing the Shapes

We wrote Python code in [3]to command the robot to draw the scaled shapes, including lifting the pen between shapes to avoid connections. The coordinates in the path are found in the `path_planning.csv` file [4]. Figure 3 shows the shapes drawn by the robot.

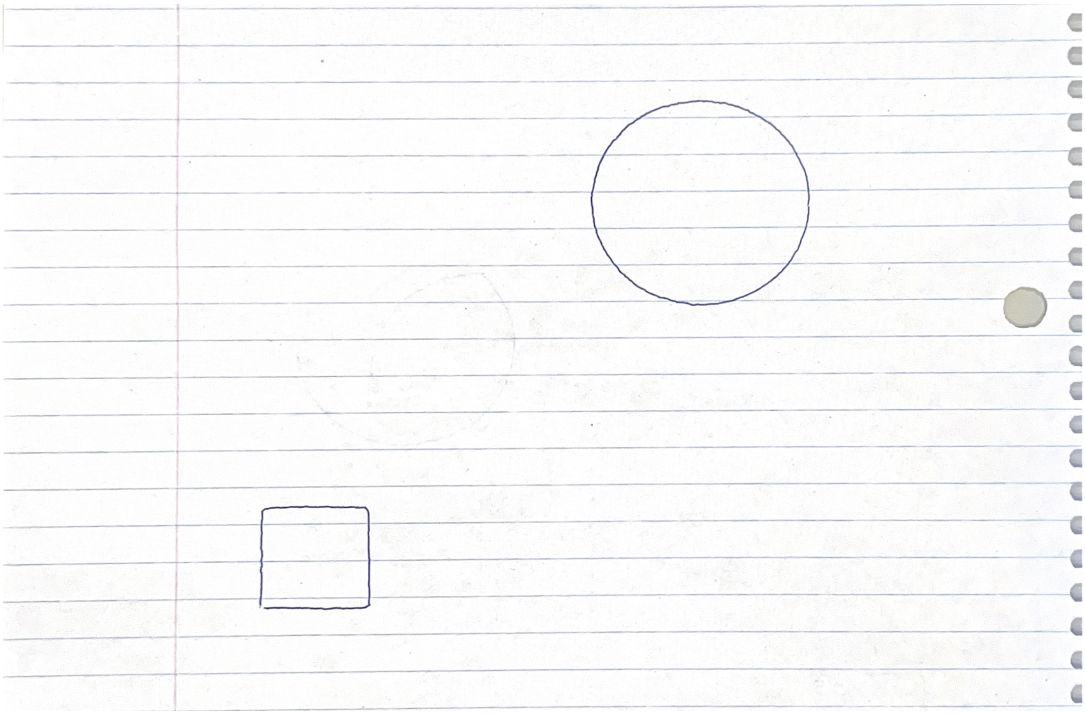

Figure 3: Shapes drawn using Dobot IK

Experimentation

During the experiments, we observed the robot's movements and identified minor misalignments. While we initially determined that having the drawing be on the XY plane on z-coordinate -27, we noticed that it did drew certain parts of the circle very lightly since not enough pressure was applied on the paper at these points. Given the spring mechanism attached to the pen, we felt comfortable changing it to -27.5 so we got much more bolder drawings. Adjustments were made to the pen's height position and translations to correct these issues. The successful completion of the task was verified through visual inspection and measurements of the drawn shapes.

Reflection

In robotics, the distinction between mathematical models and physical systems is crucial. Mathematical robots are abstract representations, often involving simplified assumptions and idealized conditions that allow engineers to develop and test algorithms in a controlled, predictable environment. They provide a foundational framework for understanding the dynamics, kinematics, and sensor integrations of a robot and are thus, crucial. However, while mathematical models are invaluable for planning and conceptualization, they do not account for all the nuances of real-world interactions, such as friction, wear and tear, or

unexpected obstacles. In the case of our robot, while an offset of -27 should have been enough, it did not give the pen enough pressure to give clear drawings.

Physical robots on the other hand do not exist in ideal environments. They must navigate real environments which introduce challenges that mathematical models might overlook or oversimplify. Therefore, engineers must bridge the gap between theoretical models and practical implementation, ensuring that algorithms are robust enough to handle real-world variability and constraints. This can be done through iterative testing and refinement, incorporating feedback from physical trials to adjust and improve the models and controls.

Safety is also an important consideration, particularly with larger, more powerful robots. While mathematical models help in predicting safe operational parameters and designing fail-safes, physical testing must rigorously validate these predictions. Engineers should prioritize safety by implementing redundant systems, emergency stop mechanisms, and thorough testing protocols. Understanding the potential hazards and designing with these in mind is thus, critical to prevent accidents.

Appendix

[1] Script from Lab 4 to localize and scale points

```python
import numpy as np
import matplotlib.pyplot as plt
import csv

# Assuming z axis offset is zero for a 2D plane plotting on the base frame
workspace_xy_limits = {
    'x_min': 150,
    'x_max': 250,
    'y_min': -50,
    'y_max': 50
}

# Read points from CSV file and return points as an array
def read_points_from_csv(file_path):
    points = []
    with open(file_path, 'r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
            points.append((float(row[0]), float(row[1])))
    return np.array(points)

# Find connected shapes based on the distance threshold of 0.25
def find_shapes(points, threshold=0.25):
    shapes = []
    current_shape = []

    for i in range(len(points) - 1):
        current_shape.append(points[i])
        if np.linalg.norm(points[i] - points[i + 1]) > threshold: # new shape if above threshold
            shapes.append(np.array(current_shape))
            current_shape = []

    current_shape.append(points[-1]) # append the last point
    shapes.append(np.array(current_shape))

    return shapes

# Scale and localize the points
def scale_and_localize_shapes(shapes, workspace_xy_limits):
    all_points = np.concatenate(shapes)
    x_min, y_min = all_points.min(axis=0)
    x_max, y_max = all_points.max(axis=0)

    # scaling factors
    scale_x = (workspace_xy_limits['x_max'] - workspace_xy_limits['x_min']) / (x_max - x_min)
    scale_y = (workspace_xy_limits['y_max'] - workspace_xy_limits['y_min']) / (y_max - y_min)
    scale = min(scale_x, scale_y)
```

```
48
49      # offsets
50      offset_x = workspace_xy_limits['x_min'] - x_min * scale
51      offset_y = workspace_xy_limits['y_min'] - y_min * scale
52
53      # apply scaling and offsets
54      localized_shapes = []
55      for shape in shapes:
56          localized_shape = shape * scale
57          localized_shape[:, 0] += offset_x
58          localized_shape[:, 1] += offset_y
59          localized_shapes.append(localized_shape)
60
61      return localized_shapes
62
63  # Output the scaled and localized points to a CSV file with z-coordinate of 0
        indicating base frame
64  def write_points_to_csv(shapes, file_path):
65      with open(file_path, 'w', newline='') as file:
66          csv_writer = csv.writer(file)
67          for shape in shapes:
68              for point in shape:
69                  csv_writer.writerow([point[0], point[1], 0]) # Adding z coordinate as
        0
70
71  def plot_shapes(shapes):
72      plt.figure()
73      for shape in shapes:
74          plt.plot(shape[:, 0], shape[:, 1], marker='o')
75      plt.xlim(workspace_xy_limits['x_min'], workspace_xy_limits['x_max'])
76      plt.ylim(workspace_xy_limits['y_min'], workspace_xy_limits['y_max'])
77      plt.gca().set_aspect('equal', adjustable='box')
78      plt.title('Localized Shapes in Task Space')
79      plt.xlabel('X')
80      plt.ylabel('Y')
81      plt.show()
82
83  def main():
84      csv_file_path = 'Lab 4/lab4_data.csv'
85      points = read_points_from_csv(csv_file_path)
86      shapes = find_shapes(points)
87      localized_shapes = scale_and_localize_shapes(shapes, workspace_xy_limits)
88
89      output_csv_file_path = 'Lab 4/scaled_localized_points.csv'
90      write_points_to_csv(localized_shapes, output_csv_file_path)
91
92      plot_shapes(localized_shapes)
93
94  if __name__ == "__main__":
95      main()
```

Listing 1: Lab 4: Script to localize and scale points

[2] Dobot Drawing a 6cm Line

```
import threading
import DobotDllType as dType
from math import sin
from math import cos
import time
import csv
import math

#load dll. This time, let's name it "magician"
magician = dType.load()

#Create string array useful for connection and IO purposes
CON_STR = {
    dType.DobotConnect.DobotConnect_NoError:  "DobotConnect_NoError",
    dType.DobotConnect.DobotConnect_NotFound: "DobotConnect_NotFound",
    dType.DobotConnect.DobotConnect_Occupied: "DobotConnect_Occupied"}

#connect to the dobot
state = dType.ConnectDobot(magician,"",115200)[0]

#print status so we can see if something goes wrong
print("Connection status: ",CON_STR[state])

#define limits - running at 100% isn't dangerous, but it's best to
#keep it slow at first
speedLimit = 40
accelLimit = 40

#define home position
xHomePos = 200
yHomePos = 0
zHomePos = 0
fourthJointHomePos = 0 #doesn't matter, we don't have a fourth joint in lab 1

xMin = xHomePos - 50
xMax = xHomePos + 50
yMin = yHomePos - 50
yMax = yHomePos + 50
zMin = zHomePos - 50
zMax = zHomePos + 50

l1 = 85
l2 = 135
l3 = 147
l4 = 27.5

# Move using coordinates
def bind_movement_to_workspace (coordinates):
    x, y, z = coordinates
    if ((x >= xMin and x <= xMax) and (y >= yMin and y <= yMax) and (z >= zMin and z
   <= zMax)):
        return dType.SetPTPCmd(magician, dType.PTPMode.PTPMOVLXYZMode, x, y, z, 0,
   isQueued = 0)[0]
```

```
    else:
        return -1

# Return joint angles required to move to given coordinates
def IK(coordinates):
    x,y,z = coordinates
    # Calculate joint angle 1
    J1 = math.atan2(y, x)

    x = x - l4
    # distance from the base to the target point in the XY plane
    r = math.sqrt(x**2 + y**2)
    # distance from the base to the target point in the XZ plane
    d = math.sqrt(r**2 + z**2)

    # Calculate joint angle 3
    cos_J3 = (d**2 - l2**2 - l3**2) / (2 * l2 * l3)
    sin_J3 = math.sqrt(1- math.pow(cos_J3, 2))
    J3 = math.atan2(sin_J3, cos_J3)

    # Calculate joint angle 2
    alpha = math.atan2(z, r)
    beta = math.asin((l3*sin_J3) / d)
    J2 = alpha - beta

    return (math.degrees(J1), math.degrees(J2), math.degrees(J3))

# Function to check if the joint angles are within the workspace limits and execute the command
def move_using_joint_angles(joint_angles):
    J1, J2, J3 = joint_angles
    forearmAngle = J2 + J3
    if (J1 >= -90 and J1 <= 90 and forearmAngle >= -30 and forearmAngle <= 85):
        return dType.SetPTPCmd(magician, dType.PTPMode.PTPMOVJANGLEMode, J1, J2, forearmAngle, 0, isQueued=0)[0]
    else:
        return -1

if(state == dType.DobotConnect.DobotConnect_NoError):

    dType.SetQueuedCmdClear(magician); #clear the queue before sending a command
    dType.SetHOMEParams(magician,xHomePos,yHomePos,zHomePos,fourthJointHomePos, isQueued=1) #Specify the home location
    dType.SetPTPCommonParams(magician, speedLimit, accelLimit, isQueued = 1) #Set the speed and acceleration limits
    dType.SetHOMECmd(magician,0,isQueued=1) #Actually go home. We need to queue this command, or things get weird

    dType.SetQueuedCmdStartExec(magician) #Execute queued commands

    path_straight_line = [(200,-30,0), (200,-30,-27.5), (200, -15,-27.5), (200,0,-27.5), (200,15,-27.5), (200,30,-27.5), (200,30,0)]
    for point in path_straight_line:
        joint_angles = IK(point)
        execCmd = move_using_joint_angles(joint_angles)
```

```
        cmdIndx = -1
        while execCmd != cmdIndx:
            dType.dSleep(100)
            cmdIndx = dType.GetQueuedCmdCurrentIndex(magician)[0]
            # Get and print the current pose
        pose = dType.GetPose(magician)
        print(f"Robot (x, y, z) = ({pose[0]}, {pose[1]}, {pose[2]})")

    #Same as before, stop execution if anything weird was left there
    dType.SetQueuedCmdStopExec(magician)

    dType.DisconnectDobot(magician)
```

Listing 2: Python script to draw a 6cm line

[3] Dobot Drawing Shapes Script

```
import threading
import DobotDllType as dType
from math import sin
from math import cos
import time
import csv
import math

#load dll. This time, let's name it "magician"
magician = dType.load()

#Create string array useful for connection and IO purposes
CON_STR = {
    dType.DobotConnect.DobotConnect_NoError:  "DobotConnect_NoError",
    dType.DobotConnect.DobotConnect_NotFound: "DobotConnect_NotFound",
    dType.DobotConnect.DobotConnect_Occupied: "DobotConnect_Occupied"}

#connect to the dobot
state = dType.ConnectDobot(magician,"",115200)[0]

#print status so we can see if something goes wrong
print("Connection status: ",CON_STR[state])

#define limits - running at 100% isn't dangerous, but it's best to
#keep it slow at first
speedLimit = 40
accelLimit = 40

#define home position
xHomePos = 200
yHomePos = 0
zHomePos = 0
fourthJointHomePos = 0 #doesn't matter, we don't have a fourth joint in lab 1

xMin = xHomePos - 50
xMax = xHomePos + 50
yMin = yHomePos - 50
```

```
yMax = yHomePos + 50
zMin = zHomePos - 50
zMax = zHomePos + 50

l1 = 85
l2 = 135
l3 = 147
l4 = 27.5

# Move using coordinates
def bind_movement_to_workspace (coordinates):
    x, y, z = coordinates
    if ((x >= xMin and x <= xMax) and (y >= yMin and y <= yMax) and (z >= zMin and z <= zMax)):
        return dType.SetPTPCmd(magician, dType.PTPMode.PTPMOVLXYZMode, x, y, z, 0, isQueued = 0)[0]
    else:
        return -1

# Return joint angles required to move to given coordinates
def IK(coordinates):
    x,y,z = coordinates
    # Calculate joint angle 1
    J1 = math.atan2(y, x)

    x = x - l4
    # distance from the base to the target point in the XY plane
    r = math.sqrt(x**2 + y**2)
    # distance from the base to the target point in the XZ plane
    d = math.sqrt(r**2 + z**2)

    # Calculate joint angle 3
    cos_J3 = (d**2 - l2**2 - l3**2) / (2 * l2 * l3)
    sin_J3 = math.sqrt(1- math.pow(cos_J3, 2))
    J3 = math.atan2(sin_J3, cos_J3)

    # Calculate joint angle 2
    alpha = math.atan2(z, r)
    beta = math.asin((l3*sin_J3) / d)
    J2 = alpha - beta

    return (math.degrees(J1), math.degrees(J2), math.degrees(J3))

# Function to check if the joint angles are within the workspace limits and execute the command
def move_using_joint_angles(joint_angles):
    J1, J2, J3 = joint_angles
    forearmAngle = J2 + J3
    if (J1 >= -90 and J1 <= 90 and forearmAngle >= -30 and forearmAngle <= 85):
        return dType.SetPTPCmd(magician, dType.PTPMode.PTPMOVJANGLEMode, J1, J2, forearmAngle, 0, isQueued=0)[0]
    else:
        return -1

if(state == dType.DobotConnect.DobotConnect_NoError):
```

```
    dType.SetQueuedCmdClear(magician); #clear the queue before sending a command
    dType.SetHOMEParams(magician,xHomePos,yHomePos,zHomePos,fourthJointHomePos,
isQueued=1) #Specify the home location
    dType.SetPTPCommonParams(magician, speedLimit, accelLimit, isQueued = 1) #Set the
speed and acceleration limits
    dType.SetHOMECmd(magician,0,isQueued=1) #Actually go home. We need to queue this
command, or things get weird

    dType.SetQueuedCmdStartExec(magician) #Execute queued commands

    Path_CSV = './path_planning.csv'
    with open(Path_CSV, 'r') as infile:
        path = csv.reader(infile)

        for row in path:
            x = float(row[0])
            y = float(row[1])
            z = float(row[2])
            z = z - 27.5

            joint_angles = IK((x, y, z))
            execCmd = move_using_joint_angles(joint_angles)
            cmdIndx = -1

            while execCmd != cmdIndx:
                dType.dSleep(100)
                cmdIndx = dType.GetQueuedCmdCurrentIndex(magician)[0]
                # Get and print the current pose
            pose = dType.GetPose(magician)
            print(f"Robot (x, y, z) = ({pose[0]}, {pose[1]}, {pose[2]})")

    #Same as before, stop execution if anything weird was left there
    dType.SetQueuedCmdStopExec(magician)

    dType.DisconnectDobot(magician)
```

Listing 3: Python script to move the Dobot to draw the two shapes in its workspace

[4] Path Coordinates

```
150.0,-50.0,30
150.0,-50.0,0
150.0,-40.0,0
150.0,-30.0,0
160.0,-30.0,0
170.0,-30.0,0
170.0,-40.0,0
170.0,-50.0,0
160.0,-50.0,0
150.0,-50.0,0
150.0,-50.0,30
250.0,30.0,30
250.0,30.0,0
```

```
249.96,31.268,0
249.839,32.53200000000001,0
249.639,33.785,0
249.35899999999998,35.02300000000001,0
249.00099999999998,36.241,0
248.567,37.43300000000001,0
248.059,38.596000000000004,0
247.477,39.724000000000004,0
246.825,40.813,0
246.105,41.85799999999999,0
245.321,42.85600000000001,0
244.475,43.80199999999999,0
243.57,44.69199999999999,0
242.611,45.522999999999996,0
241.601,46.292,0
240.54500000000002,46.995000000000005,0
239.445,47.629000000000005,0
238.308,48.193,0
237.138,48.68299999999999,0
235.938,49.098,0
234.71499999999997,49.43600000000001,0
233.473,49.696,0
232.21699999999998,49.87700000000001,0
230.952,49.977000000000004,0
229.683,49.997,0
228.41500000000002,49.937,0
227.154,49.79599999999999,0
225.904,49.57599999999999,0
224.671,49.277,0
223.459,48.900000000000006,0
222.273,48.447,0
221.119,47.92,0
220.0,47.321,0
218.92200000000003,46.650999999999996,0
217.888,45.91499999999999,0
216.90300000000002,45.11500000000001,0
215.971,44.254000000000005,0
215.095,43.33500000000001,0
214.279,42.363,0
213.526,41.34100000000001,0
212.84,40.274,0
212.22999999999998,39.165000000000006,0
211.678,38.019000000000005,0
211.20600000000002,36.83999999999999,0
210.81,35.63499999999999,0
210.49099999999999,34.406000000000006,0
210.251,33.16,0
210.091,31.900999999999996,0
210.01,30.635000000000005,0
210.01,29.364999999999995,0
210.091,28.09899999999999,0
210.251,26.840000000000003,0
210.49099999999999,25.593999999999994,0
210.81,24.36500000000001,0
211.20600000000002,23.159999999999997,0
```

```
211.678,21.980999999999995,0
212.22299999999998,20.835000000000008,0
212.84,19.726,0
213.526,18.659000000000006,0
214.279,17.637,0
215.095,16.664999999999992,0
215.971,15.74600000000001,0
216.90300000000002,14.885000000000005,0
217.888,14.085000000000008,0
218.92200000000003,13.348999999999997,0
220.0,12.678999999999995,0
221.119,12.079999999999998,0
222.273,11.553000000000004,0
223.459,11.100000000000001,0
224.671,10.723000000000006,0
225.904,10.424,0
227.154,10.204,0
228.41500000000002,10.063000000000002,0
229.683,10.002999999999993,0
230.952,10.023000000000003,0
232.21699999999998,10.123000000000005,0
233.473,10.304000000000002,0
234.71499999999997,10.563999999999993,0
235.938,10.902000000000001,0
237.138,11.317,0
238.308,11.807000000000002,0
239.445,12.370999999999995,0
240.54500000000002,13.005000000000003,0
241.601,13.707999999999998,0
242.611,14.47699999999999,0
243.57,15.307999999999993,0
244.475,16.198000000000008,0
245.321,17.144000000000005,0
246.105,18.141999999999996,0
246.825,19.186999999999998,0
247.477,20.27600000000001,0
248.059,21.403999999999996,0
248.567,22.567000000000007,0
249.00099999999998,23.759,0
249.35899999999998,24.977000000000004,0
249.639,26.215000000000003,0
249.839,27.468000000000004,0
249.96,28.732,0
250.0,30.0,0
250, 0, 30
```

Listing 4: path_planning.csv