浙江大学

Undergraduate Experiment Report

Trajectory Planning

Wanting Yao 3210100894

Minghe Wang 3210102120

Erzat Abdnur 3210106206

Shener Chen 3210103954

Yuxin Huang 3210103751

Yi Zhang 3210102174

Siyun Wen 3190104714

April. 11, 2024

Contents

List of Figures

浙江大学实验报告

Major: Automation
Date: April. 11, 2024
Place: Online

Course: Robot Modeling and Control Instructor: Chunlin Zhou Tutor: Yinuo Yu
Experiment: Trajectory Planning Experiment Type: Validation Grade:

I Purpose

1) Master the robotic arm trajectory planning methods.
2) Learn how to use the CoppeliaSim simulation tool to simulate the motion of the ZJU-I robotic arm.
3) In the simulation, transport the four blocks from the starting place through the dyeing pool to the destination and place them according to the requirement.

II Content and Principles

1. Find the Poses of the Blocks

Through the simulation environment layout, we can get the starting point and the endpoint of the dyeing pool, as well as the destination of the four blocks. By calling the function `sim.getObjectPose()` and `sim.getObjectOrientation()`, we can get the position and orientation of the starting point of the four blocks respectively.

2. Trajectory Planning Method

We split the process of transporting a single block into seven phases. For example, when transporting the first block, we denote the starting state of each phase as $q_0, q_1, q_2, q_{left}, q_{right}, q_3, q_4$:

- q_0: The initial joint angles.
- q_1: The joint angles at the sucking point of the block.
- q_2: The joint angles when the robotic arm has sucked and lifted the block.
- q_{left}: The joint angles when the robotic arm reaches the left side of the dyeing pool.
- q_{right}: The joint angles when the robotic arm reaches the right side of the dyeing pool.
- q_3: The joint angles when the robotic arm reaches the place above where the object is to be placed.
- q_4: The joint angles when the robotic arm places the block.

To obtain the above joint angles, we first get the poses of each state in the world coordinate system (as shown above), then substitute them into the IK solver and manually pick the legitimate one to get the joint angles. The code for solving the joint angles is as below:

```
q1 = iks.solve(np.array([0.4, 0.12, 0.15, -np.pi, 0, -48/180*np.pi]))[:,2]
```

The seven phases are the movement processes of the robotic arm between states. They can be divided into linear movement phases ($q_{left} \rightarrow q_{right}$) and non-linear movement phases, and the two different types of phases need to use different trajectory planning methods as shown below.

2.1 Non-linear trajectory planning

For this type of trajectory planning, we use the quintic polynomial solution:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

The constraints are:

$$\begin{cases} q(0) = q_0 & q(t) = q_f \\ q'(0) = q'_0 & q'(t) = q'_f \\ q''(0) = q''_0 & q''(t) = q''_f \end{cases}$$

Hence, the following equation can be obtained:

$$\begin{pmatrix} t_0^5 & t_0^4 & \cdots & 1 \\ t_f^5 & t_f^4 & \cdots & 1 \\ 5t_0^4 & 4t_0^3 & \cdots & 0 \\ 5t_f^4 & 4t_f^3 & \cdots & 0 \\ 20t_0^3 & 12t_0^2 & \cdots & 0 \\ 20t_f^3 & 12t_f^2 & \cdots & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} q_0 \\ q_f \\ q'_0 \\ q'_f \\ q''_0 \\ q''_f \end{pmatrix}$$

The coefficients of the quintic polynomial can be obtained by solving the matrix equation. We can get the joint angles of the robotic arm at any time by substituting the time `t` into the quintic polynomial function.

The above procedures are shown in function `trajPlanningDemo()` as follows.

```
def trajPlaningDemo(start, end, t, time):
    """ Quintic Polynomial: x = k5*t^5 + k4*t^4 + k3*t^3 + k2*t^2 + k1*t + k0
    :param start: Start point
    :param end: End point
    :param t: Current time
    :param time: Expected time spent
    :return: The value of the current time in this trajectory planning
    """
    if t < time:
        tMatrix = np.matrix([
        [         0,           0,             0,          0,        0,   1],
        [   time**5,     time**4,       time**3,    time**2,     time,   1],
        [         0,           0,             0,          0,        1,   0],
        [ 5*time**4,   4*time**3,     3*time**2,     2*time,        1,   0],
        [         0,           0,             0,          2,        0,   0],
        [20*time**3,  12*time**2,        6*time,          2,        0,   0]])

        xArray = []
        for i in range(len(start)):
            xArray.append([start[i], end[i], 0, 0, 0, 0])
        xMatrix = np.matrix(xArray).T

        kMatrix = tMatrix.I * xMatrix

        timeVector = np.matrix([t**5, t**4, t**3, t**2, t, 1]).T
        x = (kMatrix.T * timeVector).T.A[0]
```

```
    else:
        x = end

    return x
```

2.2 Linear trajectory planning

Linear trajectory planning restricts the positional changes of the robotic arm in the world coordinate system. Here we use the **segmented sampling method**, where the straight line is segmented first, and the joint angles of each point on the line can be obtained using the IK solver. Due to the large number of segments, the position change of each segment is very small, so we can directly use the `move()` function to move the robotic arm from one end of the segment to the other one by one, which in the end enables the robotic arm to move in a straight line.

```
def LineartrajPlanning(start, end, t, time):
    if t < time:
        r = int(t / time * num_line_points)
        res = spline_q_lists[r]
    else:
        res = spline_q_lists[num_line_points-1]

    if True in np.isnan(res):
        print("trajPlaning(): NAN")
        return False
    return res
```

First, we generate the set of points `spline_q_lists`, in which all the points are on the line beween the left and right points of the dyeing pool. Then we get the position (joint angles) of the robotic arm according to the segmentation interval in which the time is located. Finally, we call function `move()` to move the robotic arm from the previous point in the set to the next point. Therefore the linear trajectory planning is accomplished.

III Main Instrumentation

1) ZJU-I Desktop Manipulator
2) Robot Joint Module
3) CoppeliaSim
4) Python, Matlab, VSCode

IV Results and Analysis

According to the above procedure, we accomplished the trajectory planning task, and the final status in the simulation is shown below in figure 1. The total process of transporting the blocks took **one minute and seventeen seconds**. The curve of acceleration, velocity, and position of each joint angle is fairly smooth. In the screen recording (included in the file folder), the robotic arm moved the blocks in the order of Cube 1 - Prism 1 - Cube 2 - Prism 2.

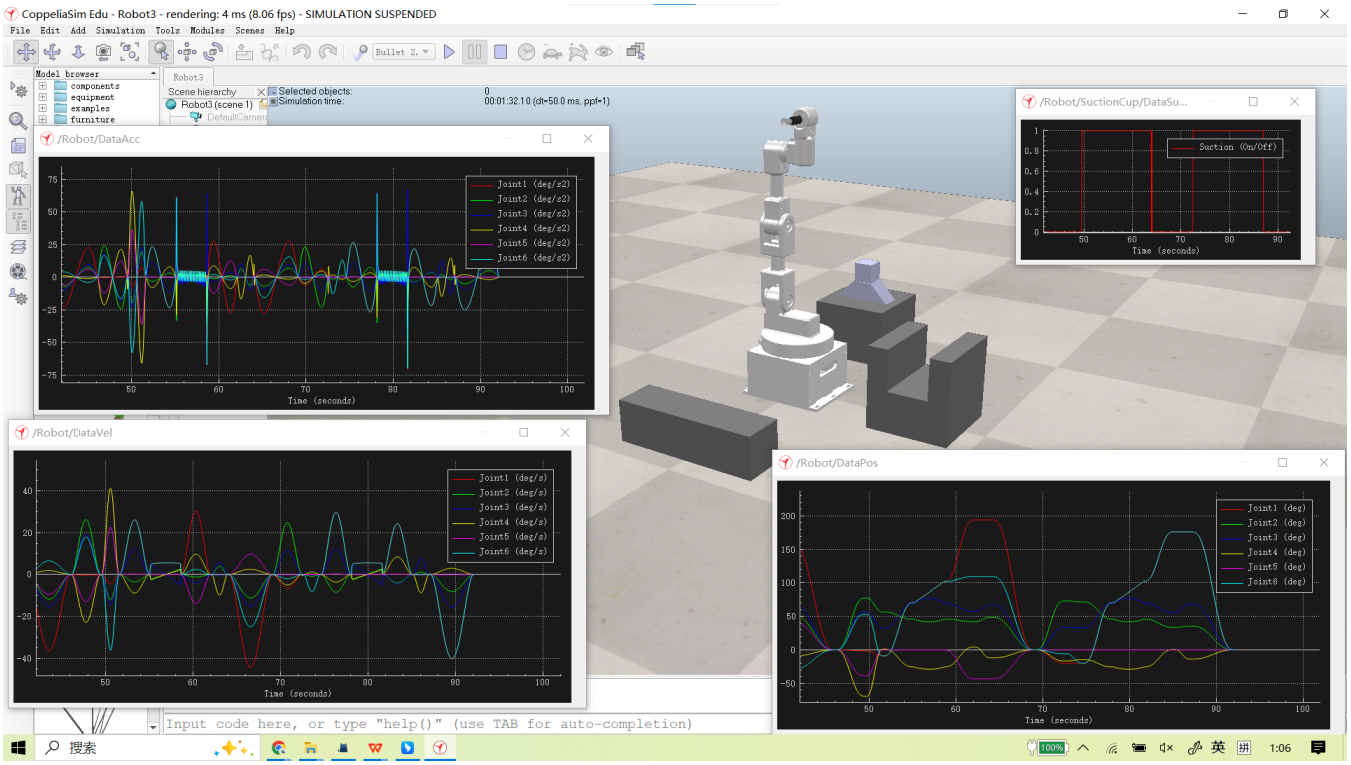


Figure 1: the final status of the simulation

The final poses of the four blocks are shown in figure 2, and we can see all the blocks are in the right place as required.

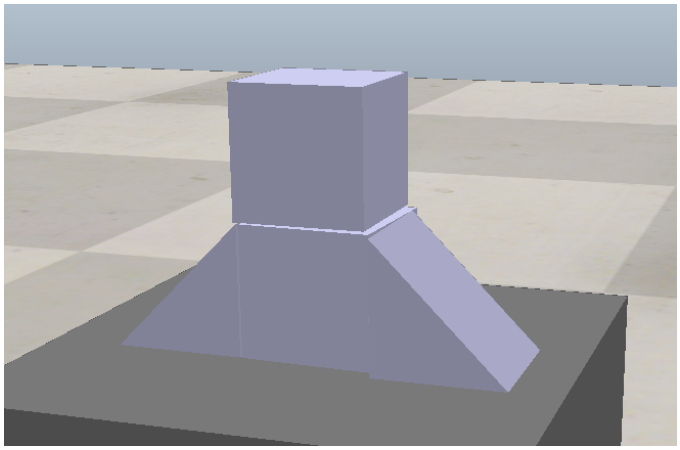


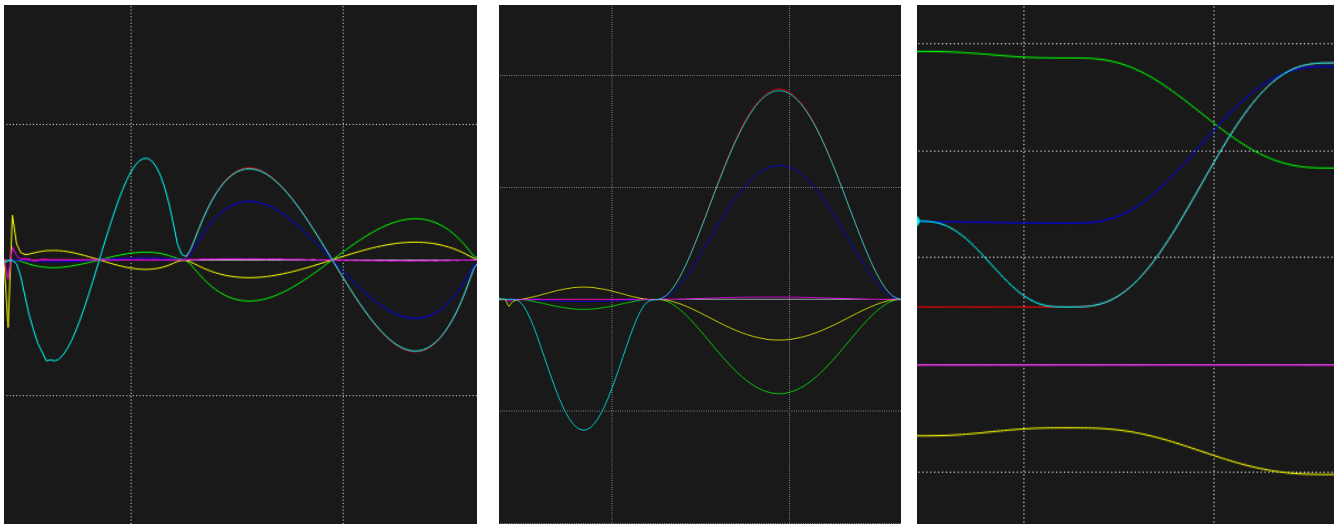
Figure 2: the blocks are placed in the right place

Also, we would like to analyze the acceleration, velocity, and position curves of the transporting process. We intercept the linear and nonlinear parts of the entire curve and analyze them respectively.

1. Non-linear Trajectory Planning

The linear segments of the curves are depicted in figure 3. Since we use the quintic polynomial to do trajectory planning, the velocity of the joint angles is a quartic polynomial while the acceleration is a cubic

polynomial. So they are both smooth curves without sharp changes.

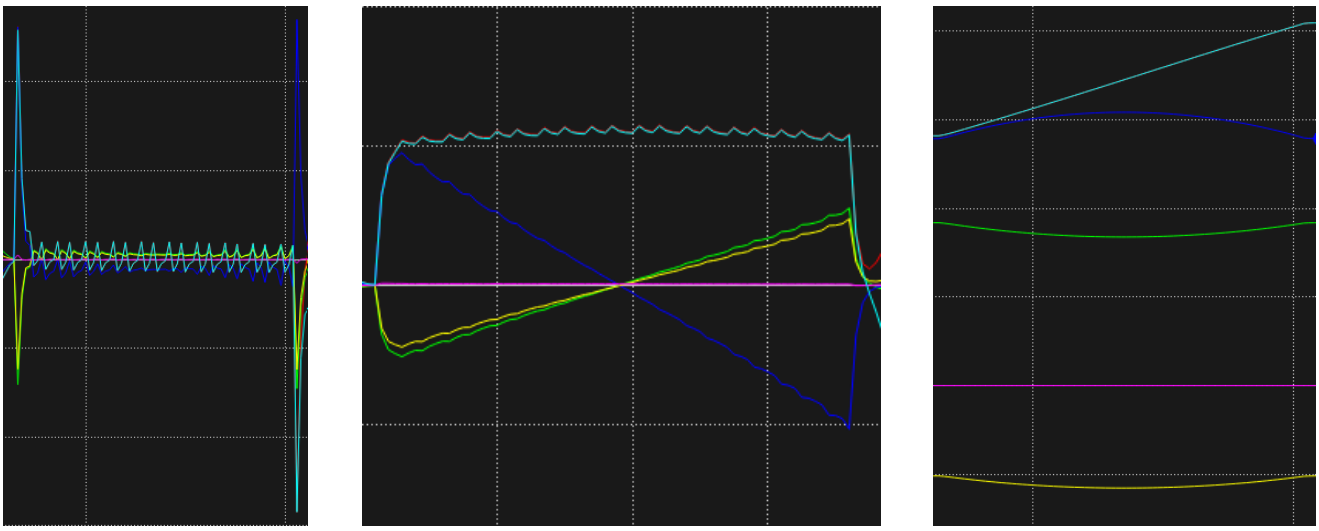


(a) Acceleration (b) Velocity (c) Position

Figure 3: the non-linear segments of the curves

2. Linear Trajectory Planning

The linear segments of the curves are depicted in figure 4. We adopted the segmented sampling method, so we can see that the velocity and acceleration curves fluctuate regularly. Since this is a uniform motion, the acceleration curve exhibits slight fluctuations near zero, while the velocity curve shows minor variations around the predetermined velocity value. The joint position transitions smoothly, resembling a roughly linear process.



(a) Acceleration (b) Velocity (c) Position

Figure 4: the linear segments of the curves

V Discussion

In this experiment, we worked together to solve the problems and gained a deeper idea of Euler angles, inverse kinematics, trajectory planning, and other concepts.

In the process of via points selection, the inverse kinematics solution can be weird for an angle out of range, in which case we need to adjust the angle by adding or subtracting times of pi. Constantly, joint angle velocity may exceed constraints, so we need to make sure the angle changes between two adjacent positions aren't too large.

VI Appendix

```python
#python
#luaExec wrapper='pythonWrapper' -- using the old wrapper for backw. compat.
# To switch to the new wrapper, simply remove above line, and add sim=require('sim')
# as the first instruction in sysCall_init() or sysCall_thread()
# from IK.IKSolver import IKSolver
import numpy as np
import sys, getpass
sys.path.append(f"C:/Users/{getpass.getuser()}/Documents/IK")
print(f"C:/Users/{getpass.getuser()}/Documents/IK")
import IK

####################################
### You Can Write Your Code Here ###
####################################

def sysCall_init():
    # initialization the simulation
    doSomeInit()    # must have

    #--------------------------------------------------------------------------
    # using the codes, you can obtain the poses and positions of four blocks
    pointHandles = []
    for i in range(2):
        pointHandles.append(sim.getObject('::/Platform1/Cuboid' + str(i+1) + '/SuckPoint'))
    for i in range(2):
        pointHandles.append(sim.getObject('::/Platform1/Prism' + str(i+1) + '/SuckPoint'))
    # get the pose of Cuboid/SuckPoint
    for i in range(4):
        print(sim.getObjectPose(pointHandles[i], -1))
    #--------------------------------------------------------------------------


    #--------------------------------------------------------------------------
    # following codes show how to call the build-in inverse kinematics solver
    # you may call the codes, or write your own IK solver
    # before you use the codes, you need to convert the above quaternions to X-Y'-Z' Euler angels
    # you may write your own codes to do the conversion, or you can use other tools (e.g. matlab)
```

```
38
39      iks = IK.IKSolver()
40      # return the joint angle vector q which belongs to [-PI, PI]
41      # Position and orientation of the end-effector are defined by [x, y, z, rx, ry, rz]
42      # x,y,z are in meters; rx,ry,rz are X-Y'-Z'Euler angles in radian
43      #-------------------------------------------------------------------------
44
45      """ this demo program shows a 3-postion picking task
46      step1: the robot stats to run from the rest position (q0)
47      step2: the robot moves to the picking position (q1) in 5s
48      step3: turn on the vacumm gripper and picking in 0.5s
49      step4: lift a block to position (q2) in 3s
50      step5: the robot moves from q2 back to the rest positon q0
51      q0 - initial joint angles of the robot
52      q1 - joint angles when the robot contacts with a block
53      q2 - final joint angels of the robot
54      """
55      global q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15, q16
56      global l, r, left, right
57      q0 = np.zeros(6) # initialize q0 with all zeros
58      # angles of joint 1-6 obtained by solving the inverse kinematics
59
60      l = np.array([0.1, 0.35, 0.2])
61      r = np.array([-0.1, 0.35, 0.2])
62      left = iks.solve(np.array([0.1, 0.35, 0.2, np.pi, 0, -np.pi/2]))[:,2]
63      print("left")
64      print(left)
65      right = iks.solve(np.array([-0.1, 0.35, 0.2, np.pi, 0, -np.pi/2]))[:,2]
66      print("right")
67      print(right)
68      '''
69      # 1st block
70      q1 = iks.solve(np.array([0.4, 0.12, 0.15, -np.pi, 0, -48/180*np.pi]))[:,2] # pick
71      q2 = iks.solve(np.array([0.4, 0.12, 0.16, np.pi, 0, -np.pi/2]))[:,2]
72      q3 = iks.solve(np.array([-0.35, 0, 0.25, np.pi, 0, -np.pi/2]))[:,2]
73      q4 = iks.solve(np.array([-0.35, 0, 0.2, np.pi, 0, -np.pi/2]))[:,2]    # place
74
75      # 2nd block
76      q5 = iks.solve(np.array([0.4, 0.04, 0.125, -142.2/180*np.pi, 26.5/180*np.pi, 120/180*np.pi]))[:,2]
77      print("q5")
78      print(q5)
79      q5[5] +=  np.pi
80      print("q5")
81      print(q5)
82      q6 = iks.solve(np.array([0.4, 0.04, 0.16, np.pi, 0, -np.pi/2]))[:,2]
83      q7 = iks.solve(np.array([-0.35, 0.05, 0.2, 0.75 * np.pi, 0, np.pi/2]))[:,2]
84      q8 =  iks.solve(np.array([-0.35, 0.05, 0.176, 0.75 * np.pi, 0, np.pi/2]))[:,2]
85
86
```

```
    # 3rd block
    q9 = iks.solve(np.array([0.4, -0.04, 0.125, 136.4/180*np.pi, 12.4/180*np.pi, -12.75/180*np.pi]))[:,2]
     ↪  # pick
    q10 = iks.solve(np.array([0.4, -0.04, 0.25, np.pi, 0, -np.pi/2]))[:,2]

    q11 = iks.solve(np.array([-0.35, -0.0515, 0.26, -0.75 * np.pi, 0, np.pi]))[:,2]
    q11[0] += 2*np.pi
    q12 = iks.solve(np.array([-0.35, -0.0515, 0.2, -0.75 * np.pi, 0, np.pi]))[:,2]    # place
    q12[0] += 2*np.pi

    #4th block
    q13 = iks.solve(np.array([0.4, -0.12, 0.15, np.pi, 0, -134/180*np.pi]))[:,2]

    q14 = iks.solve(np.array([0.4, -0.12, 0.16, np.pi, 0, -np.pi/2]))[:,2]

    q15 = iks.solve(np.array([-0.35, 0, 0.3, np.pi, 0, -np.pi/2]))[:,2]

    q16 = iks.solve(np.array([-0.35, 0, 0.255, np.pi, 0, -np.pi/2]))[:,2]
    '''

    # 1st block
    q1 = iks.solve(np.array([0.4, 0.12, 0.15, -np.pi, 0, -70/180*np.pi]))[:,2] # pick
    print("q1")
    print(q1)
    q2 = iks.solve(np.array([0.4, 0.12, 0.16, np.pi, 0, -np.pi/2]))[:,2]
    print("q2")
    print(q2)
    q3 = iks.solve(np.array([-0.35, 0, 0.25, np.pi, 0, -np.pi/2]))[:,2]
    q3[5] -=  np.pi
    print("q3")
    print(q3)
    q4 = iks.solve(np.array([-0.35, 0, 0.2, np.pi, 0, -np.pi/2]))[:,2]    # place
    q4[5] -=  np.pi
    print("q4")
    print(q4)

    # 2nd block
    #q5 = iks.solve(np.array([0.4, 0.04, 0.125, -144.108/180*np.pi, 29.21/180*np.pi, 30/180*np.pi]))[:,2]
    q5 = iks.solve(np.array([0.4, 0.04, 0.125, -144.108/180*np.pi, 29.21/180*np.pi, 120/180*np.pi]))[:,2]
    print("q5")
    print(q5)
    q5[5] +=  np.pi
    print("q5")
    print(q5)
    q6 = iks.solve(np.array([0.4, 0.04, 0.16, np.pi, 0, -np.pi/2]))[:,2]
    print(q6)
    q7 = iks.solve(np.array([-0.35, 0.05, 0.2, 0.75 * np.pi, 0, - np.pi/2]))[:,2]
    q7[5] -= np.pi
    print(q7)
```

```
    q8 =  iks.solve(np.array([-0.35, 0.05, 0.176, 0.75 * np.pi, 0, -np.pi/2]))[:,2]
    q8[5] -= np.pi
    print("q8")
    print(q8)


    # 3rd block
    q9 = iks.solve(np.array([0.4, -0.04, 0.125, 138.233/180*np.pi, 18.549/180*np.pi,
     ↪  -19.608/180*np.pi]))[:,2] # pick
    q10 = iks.solve(np.array([0.4, -0.04, 0.25, np.pi, 0, -np.pi/2]))[:,2]

    q11 = iks.solve(np.array([-0.35, -0.0515, 0.26, -0.75 * np.pi, 0, np.pi]))[:,2]
    q11[0] += 2*np.pi
    q12 = iks.solve(np.array([-0.35, -0.0515, 0.2, -0.75 * np.pi, 0, np.pi]))[:,2]    # place
    q12[0] += 2*np.pi

    #4th block
    q13 = iks.solve(np.array([0.4, -0.12, 0.15, np.pi, 0, -76/180*np.pi]))[:,2]

    q14 = iks.solve(np.array([0.4, -0.12, 0.16, np.pi, 0, -np.pi/2]))[:,2]

    q15 = iks.solve(np.array([-0.35, 0, 0.3, np.pi, 0, -np.pi/2]))[:,2]

    q16 = iks.solve(np.array([-0.35, 0, 0.255, np.pi, 0, -np.pi/2]))[:,2]



    print("!!!!!!!!!!!!!!!!!!!!!!!!!")
    print("!!!!!!!!!!!!!!!!!!!!!!!!!")

    global move_time, lift_time, wait_time, statetime, return_time
    move_time = 3.5          # move t
    lift_time = 2          # lift t
    wait_time = 0.1        # wait t
    statetime = 0.1        # state t
    return_time = 5          # q0 t

    global spline_points, spline_q_lists, num_line_points
    num_line_points = 1000
    spline_q_lists=[]
    spline_points = [(l + (r - l) * i / num_line_points) for i in range(num_line_points)]
    for i in range(num_line_points):
        spline_q_lists.append(iks.solve(np.append(spline_points[i], [np.pi, 0, -np.pi/2]))[:,2])
        if True in np.isnan(spline_q_lists[i]):
            spline_q_lists[i] = spline_q_lists[i-1]

def sysCall_actuation():
    # put your actuation code in this function

```

```
    # get absolute time, t
    t = sim.getSimulationTime()

    # 1st block
    offset = 0

    if  offset <= t:
        if t < offset + move_time * 1 - statetime:
            q = trajPlaningDemo(q0, q1, t-offset, move_time-wait_time-statetime)
            state = False
        elif t < offset + move_time * 1:
            q = q1
            state = True
        elif t < offset + move_time * 1 + lift_time * 1:
            q = trajPlaningDemo(q1, q2, t-(offset + move_time * 1), lift_time-wait_time)
            state = True
        elif t < offset + move_time * 2 + lift_time * 1:
            q = trajPlaningDemo(q2, left, t-(offset + move_time * 1 + lift_time * 1),
             ↪  move_time-wait_time)
            state = True
        elif t < offset + move_time * 3 + lift_time * 1:
            q = LineartrajPlanning(l, r, t-(offset + move_time * 2 + lift_time * 1), move_time)
            state = True
        elif t < offset + move_time * 4 + lift_time * 1:
            q = trajPlaningDemo(right, q3, t-(offset + move_time * 3 + lift_time * 1),
             ↪  move_time-wait_time)
            state = True
        elif t < offset + move_time * 4 + lift_time * 2 - statetime:
            q = trajPlaningDemo(q3, q4, t-(offset + move_time * 4 + lift_time * 1),
             ↪  lift_time-wait_time-statetime)
            state = True
        elif t < offset + move_time * 4 + lift_time * 2:
            q = q4
            state = False
        elif t < offset + move_time * 4 + lift_time * 2 + return_time:
            q = trajPlaningDemo(q4, q0, t-(offset + move_time * 4 + lift_time * 2),
             ↪  return_time-wait_time)
            state = False

    # 2nd block
    offset = (move_time * 4 + lift_time * 2 + return_time)
    if  offset <= t:
        if  t < offset + move_time * 1 - statetime:
            q = trajPlaningDemo(q0, q5, t-offset, move_time-wait_time-statetime)
            state = False
        elif t < offset + move_time * 1:
            q = q5
            state = True
        elif t < offset + move_time * 1 + lift_time * 1:
```

```
            q = trajPlaningDemo(q5, q6, t-(offset + move_time * 1), lift_time-wait_time)
            state = True
        elif t < offset + move_time * 2 + lift_time * 1:
            q = trajPlaningDemo(q6, left, t-(offset + move_time * 1 + lift_time * 1),
             ↪ move_time-wait_time)
            state = True
        elif t < offset + move_time * 3 + lift_time * 1:
            q = LineartrajPlanning(l, r, t-(offset + move_time * 2 + lift_time * 1), move_time)
            state = True
        elif t < offset + move_time * 4 + lift_time * 1:
            q = trajPlaningDemo(right, q7, t-(offset + move_time * 3 + lift_time * 1),
             ↪ move_time-wait_time)
            state = True
        elif t < offset + move_time * 4 + lift_time * 2 - statetime:
            q = trajPlaningDemo(q7, q8, t-(offset + move_time * 4 + lift_time * 1),
             ↪ lift_time-wait_time-statetime)
            state = True
        elif t < offset + move_time * 4 + lift_time * 2:
            q = q8
            state = False
        elif t < offset + move_time * 4 + lift_time * 2 + return_time:
            q = trajPlaningDemo(q8, q0, t-(offset + move_time * 4 + lift_time * 2),
             ↪ return_time-wait_time)
            state = False

    # 3rd block
    offset = (move_time * 4 + lift_time * 2 + return_time) * 2
    #offset = 0
    if  offset <= t:
        if  t < offset + move_time * 1 - statetime:
            q = trajPlaningDemo(q0, q9, t-offset, move_time-wait_time-statetime)
            state = False
        elif t < offset + move_time * 1:
            q = q9
            state = True
        elif t < offset + move_time * 1 + lift_time * 1:
            q = trajPlaningDemo(q9, q10, t-(offset + move_time * 1), lift_time-wait_time)
            state = True
        elif t < offset + move_time * 2 + lift_time * 1:
            q = trajPlaningDemo(q10, left, t-(offset + move_time * 1 + lift_time * 1),
             ↪ move_time-wait_time)
            state = True
        elif t < offset + move_time * 3 + lift_time * 1:
            q = LineartrajPlanning(l, r, t-(offset + move_time * 3 + lift_time * 1), move_time )
            state = True
        elif t < offset + move_time * 4 + lift_time * 1:
            q = trajPlaningDemo(right, q11, t-(offset + move_time * 3 + lift_time * 1),
             ↪ move_time-wait_time)
            state = True
```

```
271         elif t < offset + move_time * 4 + lift_time * 2 - statetime:
272             q = trajPlaningDemo(q11, q12, t-(offset + move_time * 4 + lift_time * 1),
                ↪ lift_time-wait_time-statetime)
273             state = True
274         elif t < offset + move_time *4 + lift_time * 2:
275             q = q12
276             state = False
277         elif t < offset + move_time * 4 + lift_time * 2 + return_time:
278             q = trajPlaningDemo(q12, q0, t-(offset + move_time * 4 + lift_time * 2),
                ↪ return_time-wait_time)
279             state = False
280 
281     # 4th block
282     offset = (move_time * 4 + lift_time * 2 + return_time) * 3
283     if  offset <= t:
284         if  t < offset + move_time * 1 - statetime:
285             q = trajPlaningDemo(q0, q13, t-offset, move_time-wait_time-statetime)
286             state = False
287         elif t < offset + move_time * 1:
288             q = q13
289             state = True
290         elif t < offset + move_time * 1 + lift_time * 1:
291             q = trajPlaningDemo(q13, q14, t-(offset + move_time * 1), lift_time-wait_time)
292             state = True
293         elif t < offset + move_time * 2 + lift_time * 1:
294             q = trajPlaningDemo(q14, left, t-(offset + move_time * 1 + lift_time * 1),
                ↪ move_time-wait_time)
295             state = True
296         elif t < offset + move_time * 3 + lift_time * 1:
297             q = LineartrajPlanning(l, r, t-(offset + move_time * 2 + lift_time * 1), move_time)
298             state = True
299         elif t < offset + move_time * 4 + lift_time * 1:
300             q = trajPlaningDemo(right, q15, t-(offset + move_time * 3 + lift_time * 1),
                ↪ move_time-wait_time)
301             state = True
302         elif t < offset + move_time * 4 + lift_time * 2 - statetime:
303             q = trajPlaningDemo(q15, q16, t-(offset + move_time * 4 + lift_time * 1),
                ↪ lift_time-wait_time-statetime)
304             state = True
305         elif t < offset + move_time * 4 + lift_time * 2:
306             q = q16
307             state = False
308         elif t < offset + move_time * 4 + lift_time * 2 + return_time:
309             q = trajPlaningDemo(q16, q0, t-(offset + move_time * 4 + lift_time * 2),
                ↪ return_time-wait_time)
310             state = False
311     if t >= (move_time * 4 + lift_time * 2 + return_time) * 4:
312         sim.pauseSimulation()
313     else:
```

```
        runState = move(q, state)

    """
    The following codes shows a procedure of trajectory planning using the 5th-order polynomial
    You may write your own code to replace this function, e.g. trapezoidal velocity planning
    """
def LineartrajPlanning(start, end, t, time):
    if t < time:
        r = int(t / time * num_line_points)
        res = spline_q_lists[r]
    else:
        res = spline_q_lists[num_line_points-1]

    if True in np.isnan(res):
        print("trajPlaning(): NAN")
        return False
    return res

def trajPlaningDemo(start, end, t, time):
    """ Quintic Polynomial: x = k5*t^5 + k4*t^4 + k3*t^3 + k2*t^2 + k1*t + k0
    :param start: Start point
    :param end: End point
    :param t: Current time
    :param time: Expected time spent
    :return: The value of the current time in this trajectory planning
    """
    if t < time:
        tMatrix = np.matrix([
        [         0,           0,             0,          0,        0,   1],
        [   time**5,     time**4,       time**3,    time**2,     time,   1],
        [         0,           0,             0,          0,        1,   0],
        [ 5*time**4,   4*time**3,     3*time**2,     2*time,        1,   0],
        [         0,           0,             0,          2,        0,   0],
        [20*time**3,  12*time**2,        6*time,          2,        0,   0]])

        xArray = []
        for i in range(len(start)):
            xArray.append([start[i], end[i], 0, 0, 0, 0])
        xMatrix = np.matrix(xArray).T

        kMatrix = tMatrix.I * xMatrix

        timeVector = np.matrix([t**5, t**4, t**3, t**2, t, 1]).T
        x = (kMatrix.T * timeVector).T.A[0]

    else:
        x = end

    return x
```

```
363
364
365 ##################################################
366 ### You Don't Have to Change the following Codes ###
367 ##################################################
368
369 def doSomeInit():
370     global Joint_limits, Vel_limits, Acc_limits
371     Joint_limits = np.array([[-200, -90, -120, -150, -150, -180],
372                              [200, 90, 120, 150, 150, 180]]).transpose()/180*np.pi
373     Vel_limits = np.array([100, 100, 100, 100, 100, 100])/180*np.pi
374     Acc_limits = np.array([500, 500, 500, 500, 500, 500])/180*np.pi
375
376     global lastPos, lastVel, sensorVel
377     lastPos = np.zeros(6)
378     lastVel = np.zeros(6)
379     sensorVel = np.zeros(6)
380
381     global robotHandle, suctionHandle, jointHandles
382     robotHandle = sim.getObject('.')
383     suctionHandle = sim.getObject('./SuctionCup')
384     jointHandles = []
385     for i in range(6):
386         jointHandles.append(sim.getObject('./Joint' + str(i+1)))
387     sim.writeCustomDataBlock(suctionHandle, 'activity', 'off')
388     sim.writeCustomDataBlock(robotHandle, 'error', '0')
389
390     global dataPos, dataVel, dataAcc, graphPos, graphVel, graphAcc
391     dataPos = []
392     dataVel = []
393     dataAcc = []
394     graphPos = sim.getObject('./DataPos')
395     graphVel = sim.getObject('./DataVel')
396     graphAcc = sim.getObject('./DataAcc')
397     color = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 1]]
398     for i in range(6):
399         dataPos.append(sim.addGraphStream(graphPos, 'Joint'+str(i+1), 'deg', 0, color[i]))
400         dataVel.append(sim.addGraphStream(graphVel, 'Joint'+str(i+1), 'deg/s', 0, color[i]))
401         dataAcc.append(sim.addGraphStream(graphAcc, 'Joint'+str(i+1), 'deg/s2', 0, color[i]))
402
403 def sysCall_sensing():
404     # put your sensing code here
405     if sim.readCustomDataBlock(robotHandle,'error') == '1':
406         return
407     global sensorVel
408     for i in range(6):
409         pos = sim.getJointPosition(jointHandles[i])
410         if i == 0:
411             if pos < -160/180*np.pi:
```

```
                pos += 2*np.pi
        vel = sim.getJointVelocity(jointHandles[i])
        acc = (vel - sensorVel[i])/sim.getSimulationTimeStep()
        if pos < Joint_limits[i, 0] or pos > Joint_limits[i, 1]:
            print("Error: Joint" + str(i+1) + " Position Out of Range!")
            sim.writeCustomDataBlock(robotHandle, 'error', '1')
            return

        if abs(vel) > Vel_limits[i]:
            print("Error: Joint" + str(i+1) + " Velocity Out of Range!")
            sim.writeCustomDataBlock(robotHandle, 'error', '1')
            return

        if abs(acc) > Acc_limits[i]:
            print("Error: Joint" + str(i+1) + " Acceleration Out of Range!")
            sim.writeCustomDataBlock(robotHandle, 'error', '1')
            return

        sim.setGraphStreamValue(graphPos,dataPos[i], pos*180/np.pi)
        sim.setGraphStreamValue(graphVel,dataVel[i], vel*180/np.pi)
        sim.setGraphStreamValue(graphAcc,dataAcc[i], acc*180/np.pi)
        sensorVel[i] = vel

def sysCall_cleanup():
    # do some clean-up here
    sim.writeCustomDataBlock(suctionHandle, 'activity', 'off')
    sim.writeCustomDataBlock(robotHandle, 'error', '0')



def move(q, state):
    if sim.readCustomDataBlock(robotHandle,'error') == '1':
        return
    global lastPos, lastVel
    for i in range(6):
        if q[i] < Joint_limits[i, 0] or q[i] > Joint_limits[i, 1]:
            print("move(): Joint" + str(i+1) + " Position Out of Range!")
            return False
        if abs(q[i] - lastPos[i])/sim.getSimulationTimeStep() > Vel_limits[i]:
            print("move(): Joint" + str(i+1) + " Velocity Out of Range!")
            return False
        if abs(lastVel[i] - (q[i] - lastPos[i]))/sim.getSimulationTimeStep() > Acc_limits[i]:
            print("move(): Joint" + str(i+1) + " Acceleration Out of Range!")
            return False

    lastPos = q
    lastVel = q - lastPos

    for i in range(6):
```

```
461         sim.setJointTargetPosition(jointHandles[i], q[i])
462 
463     if state:
464         sim.writeCustomDataBlock(suctionHandle, 'activity', 'on')
465     else:
466         sim.writeCustomDataBlock(suctionHandle, 'activity', 'off')
467 
468     return True
```