ID6040: Introduction to Robotics

Ayush Mukund Jamdar EE20B018

April 2, 2023

1 Programming Assignment 3

The assignment aims to let the learner implement a differential inverse kinematics solution to the same two link manipulator so as to move the end effector with a fixed velocity to reach a goal. I compute the joint angular velocities that vary continuously over the motion that will get the desired end effector velocity.

2 The Assignment

2.1 The Problem

The CoppeliaSim scene is as follows.

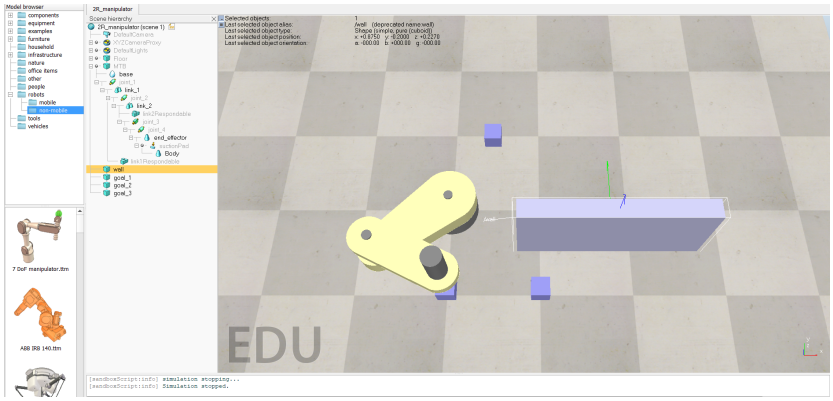

Figure 1: A-3 Scene

The end effector has to move from Goal-2 to Goal-3 (the lower purple objects) such that the effector velocity (m/s) remains a constant -

$$\vec{v_e} = 0.01\hat{i} + 0\hat{j}$$

It is, at this point, necessary to understand that it is possible to tread this path with only an x velocity because the end effector's initial and final destinations are known to be along the same y coordinate. It will not always be possible to get this solution for any final goal. For example, it cannot reach Goal-1 (top) with this velocity. Mathematically, the arm will extend until it stretches out entirely (after having moved only along x) and reaches a singularity.

What we demonstrate in this assignment is the ability to calculate rotation speeds with time that can give the constant end effector speed.

2.2 Differential Kinematics

The given system is represented in Figure 1. l_1 and l_2 are the link lengths and θ_1 and θ_2 are joint angles. Forward kinematics establishes -

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$

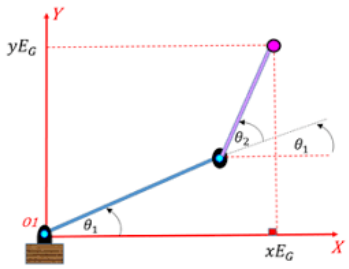

Figure 2: A graphical representation of the system.

$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$

Let's make a move to the time domain by differentiating the equations with time. LHS gives end-effector speeds and RHS joint angular speeds.

$$V_x = -l_1 \sin(\theta_1)\dot{\theta_1} - l_2 \sin(\theta_1 + \theta_2)(\dot{\theta_1} + \dot{\theta_2})$$

$$V_y = l_1 \cos(\theta_1)\dot{\theta_1} + l_2 \cos(\theta_1 + \theta_2)(\dot{\theta_1} + \dot{\theta_2})$$

In a matrix,

$$\begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} = \begin{bmatrix} -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta_1} \\ \dot{\theta_2} \end{bmatrix}$$

We write this as

$$\vec{V} = J \cdot \dot{\theta}$$

where J is the Jacobian, the matrix of partial derivatives.

$$J = \begin{bmatrix} -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \\ 0 & 0 \end{bmatrix}$$

Now the manipulator is 2D; we ignore the third row corresponding to V_z. This makes the Jacobian a $2x2$ matrix which can be inverted normally. Velocity vector is given by

$$V = \begin{bmatrix} 0.01 \\ 0 \end{bmatrix}$$

Singularity Well there isn't really a solution to the problem if the manipulator runs into a singular configuration ($\|J\| = 0$). All we can do is to make sure that no two links are extended straight out. In this case, it is easy to see that it won't have to go to the workspace boundary. Hence it is ensured that the Jacobian will always be invertible.

The final solution

$$\dot{\theta} = J^{-1} \cdot \vec{V}$$

That's it! Continue to the next section will describes the implementation.

2.3 Simulation

The above equations are implemented in the Python module `differential_kinematics.py` as follows. An exception would be raised if the Jacobian gets singular. `numpy` does the rest of the magic and we don't have to worry about linear algebra.

```
def get_desired_joint_rate(joint_angles):

    l1 = robot_params.link_1_length
    l2 = robot_params.link_2_length

    # Compute 2x2 jacobian matrix for given joint angles [in radians]
    j11 = -l1 * math.sin(joint_angles[0]) - l2 * math.sin(
        joint_angles[0] + joint_angles[1]
    )
    j12 = -l1 * math.sin(joint_angles[0] + joint_angles[1])
    j21 = l1 * math.cos(joint_angles[0]) + l2 * math.cos(
        joint_angles[0] + joint_angles[1]
    )
    j22 = l2 * math.cos(joint_angles[0] + joint_angles[1])

    jacobian = np.array([[j11, j12], [j21, j22]])

    # raise an exception if jacobian is singular
    if np.linalg.det(jacobian) == 0:
        raise Exception("Jacobian is singular")
    else:
        # Compute desired joint angle rate by inverting jacobian and
        # multiplying with desired end effector velocity
        desired_end_effector_velocity = np.array([[0.01], [0.0]])
        desired_joint_angle_rates = np.matmul(
            np.linalg.inv(jacobian), desired_end_effector_velocity
        )

    return [desired_joint_angle_rates[0], desired_joint_angle_rates[1]]
```

The `sim_interface.py` connects the Python simulation scripts to the CoppeliaSim scene and controls joint rotation speeds.

2.4 Simulation Results

In a successful run, the arm slowly moves along x to reach Goal-3. The below three screenshots show the successful run of the simulation with correct end effector position.

2.5 Conclusion

The goals the assignment was started with are now seen to have been accomplished. The inverse Jacobian works perfect to drive the end effector at a constant speed to the desired position. Joint rotation speeds are calculated at intervals of 0.1.

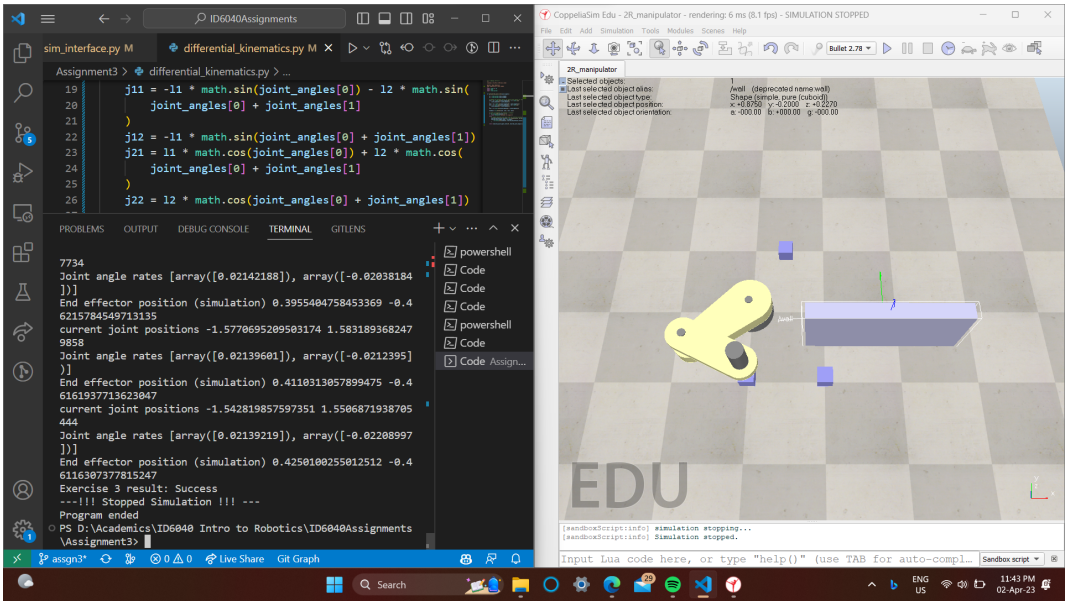

Figure 3: Results-1

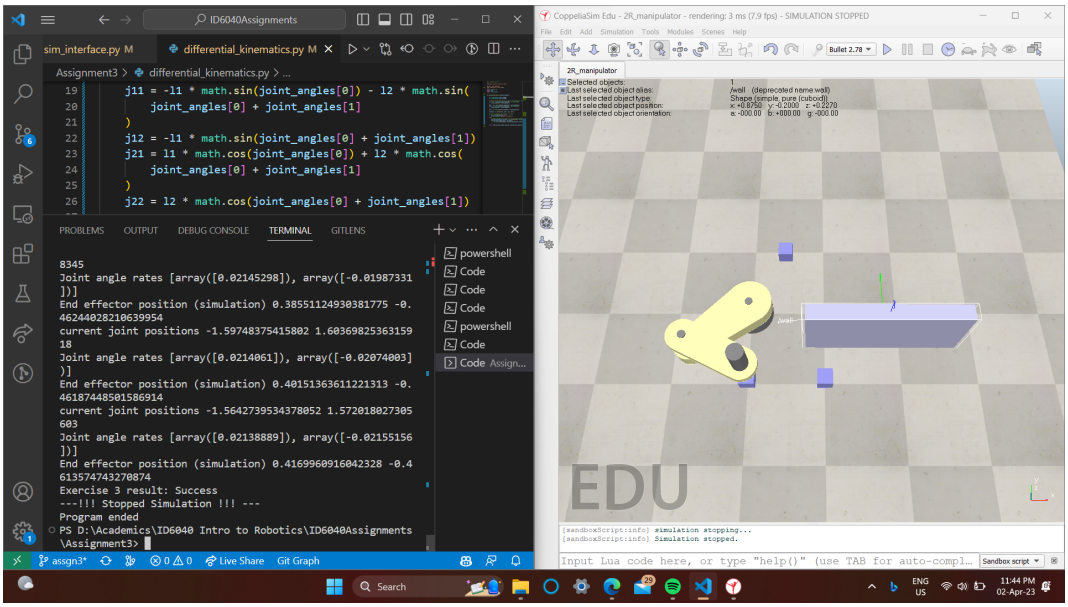

Figure 4: Results-2

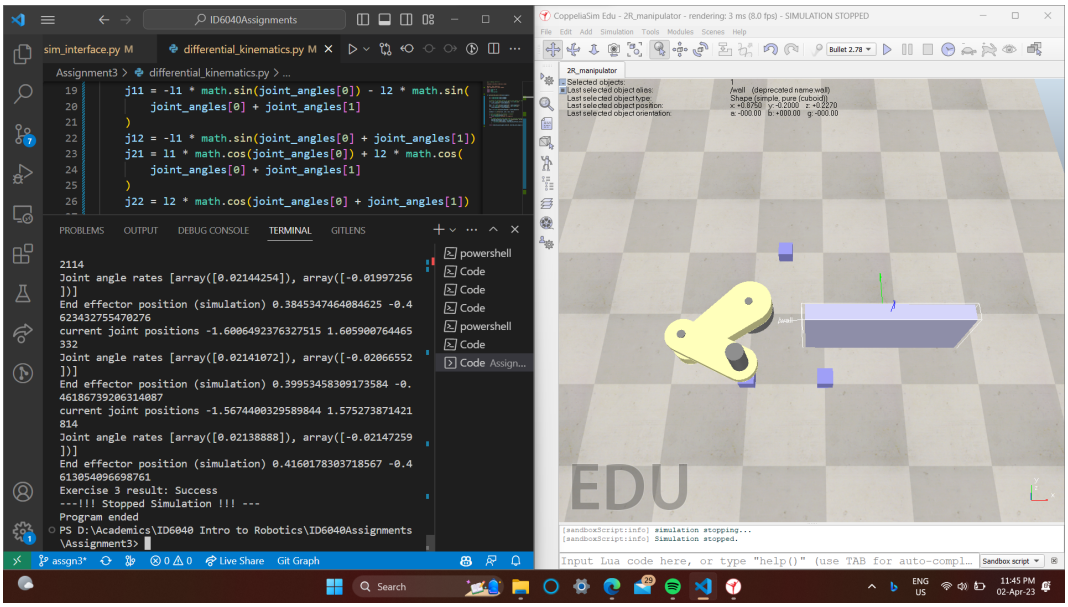

Figure 5: Results-3