MTRX5700: Experimental Robotics
Assignment 1

Group - SegFault

Student	Contribution (%)
520476363	25
520465644	25
52046527	25
510657840	25

1 Simulating and Experimentally Validating Kinematic Chains

1.1 Derivations of DH and Modified DH

In order to determine the DH Tables for the UR5e, we first assign reference axes at each of the six joints of concern, starting at joint 1. The axes are placed following two general rules:

- the z-axis is along the axis of rotation
- the x-axis is based on z_{i-1} and z_i such that the x-axis is perpendicular to both

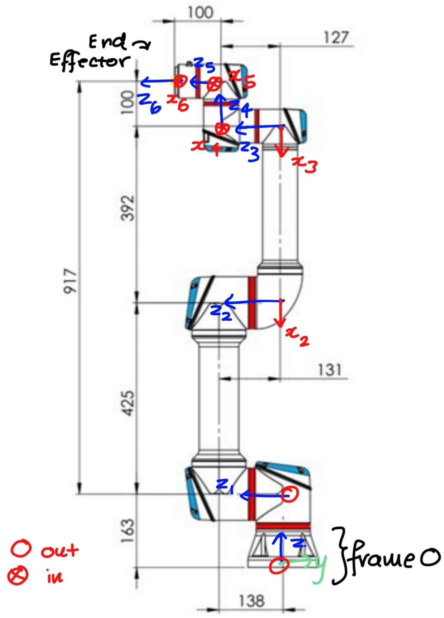


Figure 1: Coordinate Frames for UR5e

Link	d_i	θ_i	a_i	α_i
1	163	θ_1	0	$\frac{\pi}{2}$
2	0	θ_2	-425	0
3	0	θ_3	-392	0
4	127	θ_4	0	$\frac{\pi}{2}$
5	100	θ_5	0	$-\frac{\pi}{2}$
6	100	θ_6	0	0

Table 1: Standard DH Table Parameters for UR5e Arm

Considering d_i as the distance between the origin of $frame_{i-1}$ to the x_i axis along the z_{i-1} axis, θ_i as the input joint angles, a_i as the distance between the z_{i-1} and z_i axes along the x_i axis, and α_i as the angle between the z_{i-1} and z_i axes about the x_i axis, taking anti-clockwise as positive, the standard DH-table can be derived by using the reference frames shown in Figure 1.

Using the DH parameter table, the homogeneous transform matrices can be constructed following the general matrix form:

$$H_{i-1}^{i} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

By following an iterative method in the ArmKinematics class as can be seen in Appendix B, the global transforms of the coordinate frames with respect to a base reference frame can then be found by the multiplication of the homogeneous transforms. The final transformation matrix for the end effector is then given by:

$$T_0^6 = H_0^1 H_1^2 H_2^3 H_3^4 H_4^5 H_5^6 \tag{2}$$

In contrast to the standard DH table, modified DH works by assigning a new coordinate system to the robotic arm. The particular difference lies as modified DH assigns the coordinates of $frame_{i-1}$ to $joint_{i-1}$ as opposed to $joint_i$ as in standard DH. Using this new coordinate system, modified DH also alters the order in which the transformations to the frames occur. Modified DH starts with rotation and translation about the x-axis, whereas standard DH experiences its first rotation and translation about the z-axis. Hence the order in which the matrices are multiplied, ultimately creates a new transformation matrix in comparison to a standard DH table. In order to account for this difference, a new DH parameter table and homogeneous transform matrix is derived as below, with more detailed derivations in Appendix A.

Link	α_{i-1}	a_{i-1}	θ_i	d_i
1	0	0	θ_1	163
2	$-\frac{\pi}{2}$	0	θ_2	0
3	0	425	θ_3	0
4	0	392	θ_4	127
5	$-\frac{\pi}{2}$	0	θ_5	100
6	$\frac{\pi}{2}$	0	θ_6	100

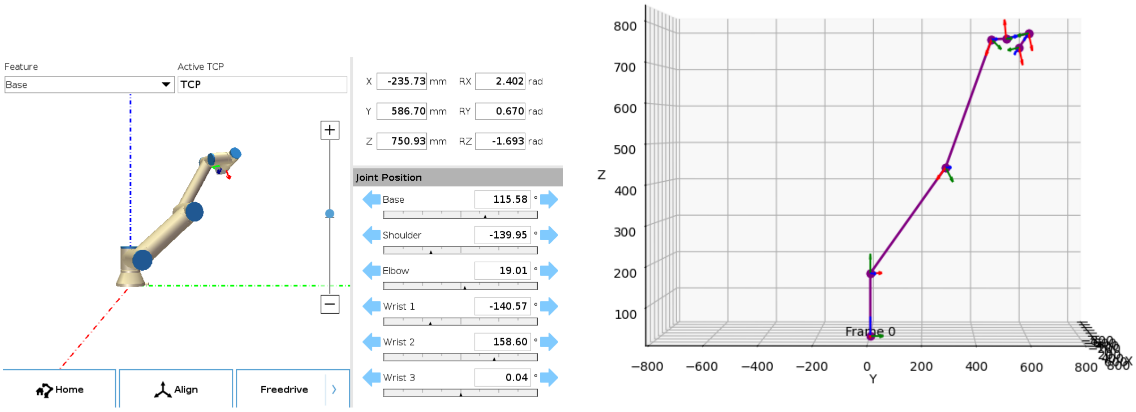
Table 2: Modified DH Table Parameters for UR5e Arm

$$\mathrm{H}_{i-1}^{i} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i\cos\alpha_{i-1} & \cos\theta_i\cos\alpha_{i-1} & -\sin\alpha_{i-1} & -\sin\alpha_{i-1}\,d_i \\ \sin\theta_i\sin\alpha_{i-1} & \cos\theta_i\sin\alpha_{i-1} & \cos\alpha_{i-1} & \cos\alpha_{i-1}\,d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

1.2 Results and Discussion

To verify the implementation of the developed forward kinematics algorithm a 3D render of the robotic arm is used. The accuracy of such algorithm is verified by testing experimentally in the lab and comparing the real output of the UR5e to the output of the 3D render. This led to the following observations.

1.2.1 Position 1



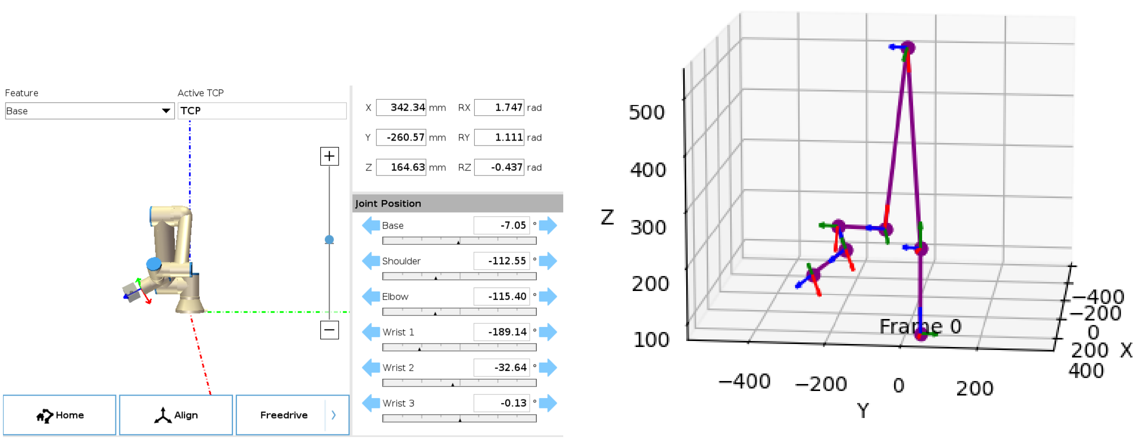
(a) Actual (b) Simulated

Figure 2: Position 1

	X	Y	Z	Rx	Ry	Rz
Calculated	-241.95	583.93	751.37	2.397	0.672	-1.695
Actual	-235.73	586.70	750.93	2.402	0.670	-1.693

Table 3: End Effector Position Values

1.2.2 Position 2



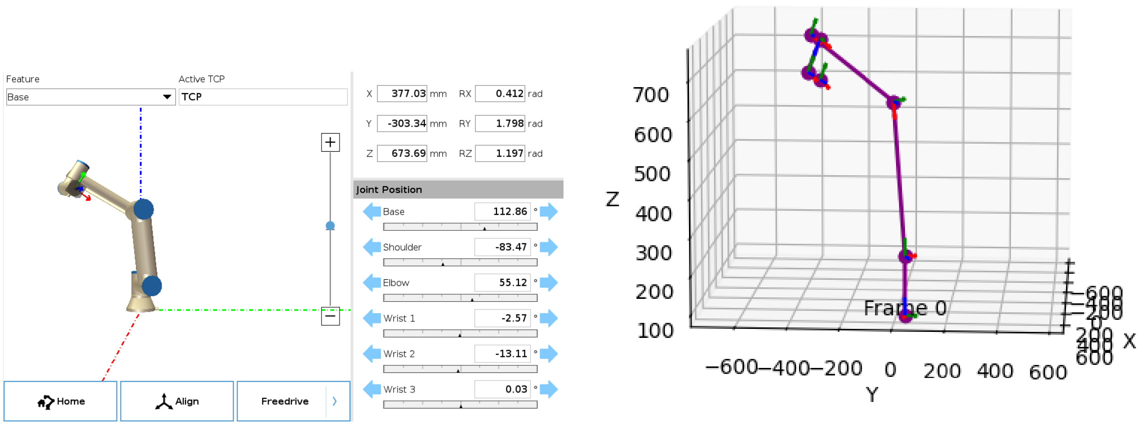
(a) Actual (b) Simulated

Figure 3: Position 2

	X	Y	Z	Rx	Ry	Rz
Calculated	342.16	-255.13	164.81	1.748	1.108	-0.439
Actual	342.34	-260.57	164.63	1.747	1.111	-0.437

Table 4: End Effector Position Values

1.2.3 Position 3



(a) Actual

(b) Simulated

Figure 4: Position 3

	X	Y	Z	Rx	Ry	Rz
Calculated	371.97	-304.67	673.94	0.409	1.799	1.199
Actual	377.03	-303.34	673.69	0.412	1.798	1.197

Table 5: End Effector Position Values

The above figures illustrate that our simulation results were accurate and consistent with the experimental values produced by the UR5e, with X, Y and Z displacement values having an error of less than 5mm (<2%). The existing errors can be attributed to inaccurate measurements in the simulated DH table, whereby some dimensions of the robotic arm provided by the reference diagram in Figure 1 differ from the real dimensions of the UR5e. The rotation angles were all within 0.01 radians between the experimental and simulated results, showing good accuracy in our simulation. We were required to convert our Euler angle output from the rotation matrix to a rotation vector because this is the default rotation representation for the UR5e arm. The angular inaccuracy is slight and can be attributed to the sensor noise experienced by the encoder wheels in the joints, which is used to calculate the angular displacement. This noise can be reduced by implementing a better band-pass filter and a window average filter. The simulation model can be considered reliable as it observed results with a consistent accuracy to the UR5e for all three positions attempted.

2 PRRP Planar Manipulator

2.1 Determining Joint Coordinates

Using the coordinate frames from Figure **??**, the configuration variables for each joint can be found using the conventions for the standard DH table, laid out in section 1.1.

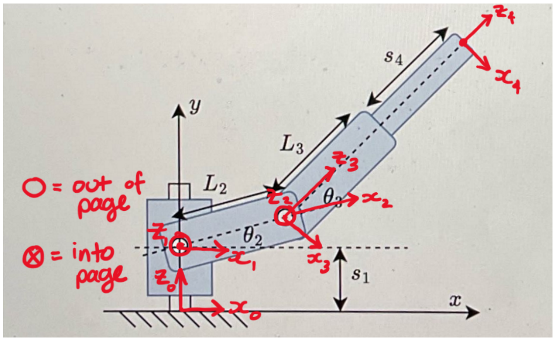


Figure 5: Planar Manipulator Frames

Link	d_i	θ_i	a_i	α_i
1	s_1	0	0	$\frac{\pi}{2}$
2	0	θ_2	600	0
3	0	$\theta_3 - \frac{\pi}{2}$	0	$-\frac{\pi}{2}$
4	$450 + s_4$	0	0	0

Table 6: Standard DH Table Parameters for PRRP Arm

2.2 Computing Transformation Matrices

The transform matrix for joint i can be found by substituting the corresponding d_i, θ_i, α_i and a_i into matrix 1. The end-effector transform can be found by multiplying the transforms from joints 0 to 4.

$$T_0^4 = H_0^1 H_1^2 H_2^3 H_3^4 \tag{3}$$

The end-effector position can be found by extracting the first 3 rows of the last column of the transform matrix. In terms of the parameters s_1, s_4, θ_2 and θ_3, the end effector position is:

$$x = (s_4 + 450)(-\sin(\theta_2)\cdot\cos(\theta_3 - \frac{\pi}{2}) - \sin(\theta_3 - \frac{\pi}{2})\cdot\cos(\theta_2)) + 600\cos(\theta_2)$$
$$y = 0$$
$$z = s_1 + (s_4 + 450)(-\sin(\theta_2)\cdot\sin(\theta_3 - \frac{\pi}{2}) + cos(\theta_2)\cdot\cos(\theta_3 - \frac{\pi}{2})) + 600\sin(\theta_2)$$

These equations were found using the Sympy mathematics library in python. s_1, s_4, θ_2 and θ_3 were declared as symbolic variables, allowing the end effector position method to find the cartesian coordinates in terms of these variables.

2.3 Plotting the Workspace

2.3.1 Without Limits

A point cloud was made using the end-effector position. Initially, there were no limits placed on the revolute joints, allowing the arm to span the full 360-degree space, which produced the following:

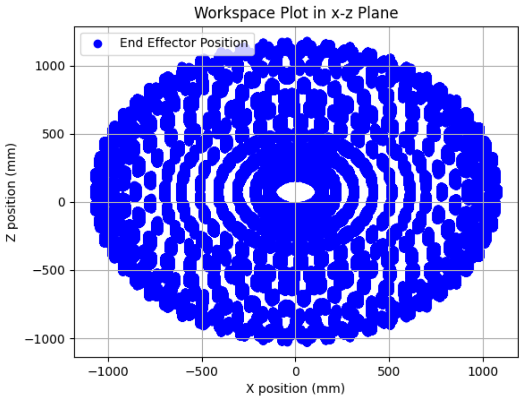


Figure 6: Workspace plot with unbound revolute joints.

2.3.2 With Limits

The following limits were placed on the revolute joints, and the workspace was re-plotted:

$$
\begin{aligned}
50 \leq & s_1 \leq 100\,\text{mm} \\
10 \leq & s_4 \leq 30\,\text{mm} \\
\frac{4\pi}{5} \leq & \theta_2 \leq \frac{\pi}{2} \\
-\frac{4\pi}{3} \leq & \theta_3 \leq \frac{4\pi}{3}
\end{aligned}
$$

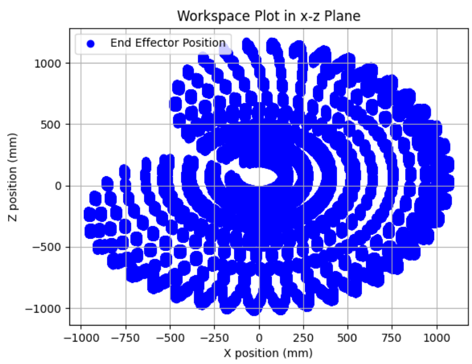


Figure 7: Workspace plot with bounded revolute joints.

2.4 Inverse Kinematics

2.4.1 Part A

Basic trigonometric principles were used to determine whether the PRRP planar arm could reach its target without interference with the obstacle. By fixing the first prismatic joint at $s_1 = 0$, we can use trigonometric functions to graph the robotic arm's configuration along with the obstacle.

By identifying the angles of both links relative to the horizontal, we apply the cosine rule as follows:

$$\theta = cos^{-1}\left(\frac{target^2 + d_1^2 - d_2^2}{2\,target\,d_1}\right) \tag{4}$$

Where,

$target = 750, d_1 = L_2, d_2 = L_3 + s_4, \theta =$ the angle of line 1 relative to the positive horizontal

This equation was also used to calculate the angle of the second link, which in turn allowed us to determine the gradients of the links for plotting.

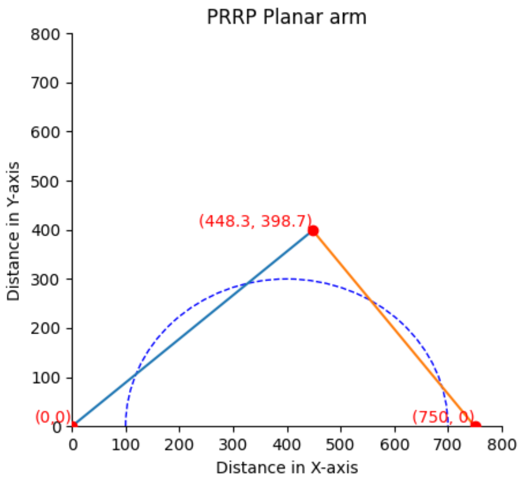


Figure 8: Diagram of PRRP Manipulator with End Effector Pose [750,0] and first Prismatic Joint at $s_1 = 0$

As shown in Figure 9, the robot's end-effector cannot reach the target point without interfering with the obstacle when $s_1 = 0$.

2.4.2 Part B

In order to determine the joint variables that cause the PRRP manipulator to obtain a specific end-effector pose, inverse kinematics can be used. Considering the state of the system, the following equations can be derived to represent the x_e and y_e end-effector positions:

$$x_e = L_2 \cos\theta_2 + (L_3 + s_4)\cos(\theta_2 + \theta_3) \tag{5}$$

$$y_e = s_1 + L_2 \sin\theta_2 + (L_3 + s_4)\sin(\theta_2 + \theta_3) \tag{6}$$

Considering the limits applied to the prismatic joints, the maximum possible extension for each joint can be arbitrarily assigned yielding $s_1 = 500, s_4 = 50$.

The two remaining unknowns, θ_2 and θ_3 can then be determined using algebraic or numerical methods. First considering the algebraic state of the system with $s_1 = 500$ and $L_3 + s_4 = 500$, the end-effector position of the target is taken as $x_e = 750, y_e = 0$. Presenting this diagrammatically gives:

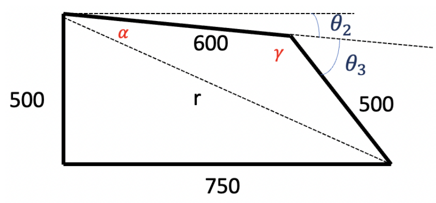


Figure 9: Diagram of PRRP Manipulator with End Effector Pose [750,0]

Performing simple trigonometry then allows for the use of inverse kinematics to solve for the two unknown joint angles. Starting with r and the use of the cosine rule to find angles α and γ we obtain:

$$r = \sqrt{500^2 + 750^2} \tag{7}$$

$$\alpha = \cos^-1\left(\frac{600^2 + r^2 - 500^2}{2(600)(r)}\right) \approx 31.48°, \gamma = \cos^-1\left(\frac{600^2 + 500^2 - r^2}{2(600)(500)}\right) \approx 109.72° \tag{8}$$

Using this, the joint angles of the PRRP manipulator, θ_2 and θ_3 can then be determined:

$$\theta_2 = \tan^-1\left(\frac{500}{750}\right) - \alpha \approx 2.21°, \theta_3 = 180° - \gamma \approx -70.28° \tag{9}$$

Verifying this with a numerical solution, Sympy is used to solve the simultaneous set of equations for x_e and y_e outlined above. This yields two solutions, with solution one returning $\theta_2 = -0.0386\text{rad} \approx -2.21°$, $\theta_3 = -1.227\text{rad} \approx -70.30°$, and solution two returning $\theta_2 = -1.137\text{rad} \approx -65.14°$, $\theta_3 = 1.227\text{rad} \approx 70.30°$. Provided that solution two implies that the robotic arm travels beneath the ground surface, solution one is examined. It is noted that solution one also verifies the algebraic solution presented above. It can then be visually verified that the arm will avoid the obstacle at the arrangement where $s_1 = 500$ and $s_4 = 50$ with $\theta_2 = -2.21°$ and $\theta_3 = -70.30°$.

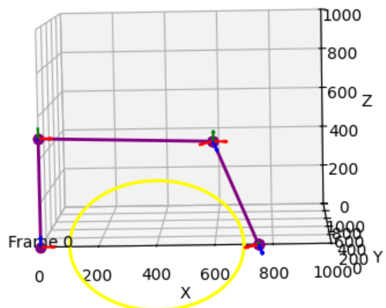


Figure 10: PRRP Manipulator with avoiding obstacle

As above, alternate arrangements of the joint variables, including the prismatic joint translation and the revolute joint rotation, which lead to an arrangement where the PRRP manipulator is capable of avoiding the obstacle, can be found by using the sympy solver and the visualiser.

Appendix A

Individual Transforms for Modified DH table:

$$H_0^1 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 163 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad H_1^2 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin\theta_2 & -\cos\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$H_2^3 = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 425 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad H_3^4 = \begin{bmatrix} \cos\theta_4 & -\sin\theta_4 & 0 & 392 \\ \sin\theta_4 & \cos\theta_4 & 0 & 0 \\ 0 & 0 & 1 & 127 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$H_4^5 = \begin{bmatrix} \cos\theta_5 & -\sin\theta_5 & 0 & 0 \\ 0 & 0 & 1 & 100 \\ -\sin\theta_5 & -\cos\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad H_5^6 = \begin{bmatrix} \cos\theta_6 & -\sin\theta_6 & 0 & 0 \\ 0 & 0 & -1 & -100 \\ \sin\theta_6 & \cos\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Appendix B

```python
import numpy as np

# *************************** cArmKinematics.py ***************************************

# Filename:       cArmVisualiser.py
# Author:         Alfie

# Description:  This file defines the cArmKinematics class which finds the
#               end effector position of the robot arm as well as the homogeneous
#               transforms from the origin to each joint. It also checks for singularities
#               using the Jacobian matrix.

# Dependencies: numpy

# ************************************************************************************

class cArmKinematics:

    TRANSFORM_DIM = 4
    WORKSPACE_DIM = 3

    # Class constructor
    def __init__(self, DHTable, NumJoints):
        self.DHTable = DHTable
        self.NumJoints = NumJoints


    #Check for singularites using Jacobian matrices
    def mCheckCorrectness(self):
        #Initialise the position and rotation vectors
        PositionVector = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])
        RotationVector = np.array([np.eye(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])
        LinearVelocity = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])
        AngularVelocity = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])

        #Obtain the position, rotation, linear velocity and angular velocity vectors from the joint pose
        for FrameNum in range(self.NumJoints):
            PositionVector[FrameNum] = self.JointPoseGlob[FrameNum][:self.WORKSPACE_DIM, -1]
            RotationVector[FrameNum] = self.JointPoseGlob[FrameNum][:self.WORKSPACE_DIM, :self.WORKSPACE_DIM]
            AngularVelocity[FrameNum] = RotationVector[FrameNum][:, -1]
            LinearVelocity[FrameNum] = np.cross(AngularVelocity[FrameNum], PositionVector[-1] - PositionVector[FrameNum])

        J = np.zeros((self.WORKSPACE_DIM + self.WORKSPACE_DIM, self.NumJoints))

        for FrameNum in range(self.NumJoints):
            J[:self.WORKSPACE_DIM, FrameNum] = LinearVelocity[FrameNum]
            J[self.WORKSPACE_DIM:, FrameNum] = AngularVelocity[FrameNum]


        #Check for singularites
        Singular = np.linalg.matrix_rank(J) < min(J.shape)

        if Singular:
            print("Singularities detected")

        else:
```

```
            print("No singularities detected")


    def mEndeffectorPosition(self):
        return self.JointPoseGlob[-1][:self.WORKSPACE_DIM, -1]


    # Determine the final end effector position
    def mGetAllJointGlobPose(self):
        self.JointPoseGlob = np.array([np.eye(self.TRANSFORM_DIM) for _ in range(self.NumJoints)])

        for FrameNum in range(self.NumJoints):
            self.JointPoseGlob[FrameNum] = self.JointPoseGlob[FrameNum-1]@self.DHTable.mConstructHT(FrameNum, Standard=True)

        return self.JointPoseGlob




```

```python
from sympy import symbols, Eq, solve, sqrt, atan2, acos, cos, sin, pprint
from cArmVisualiser import cArmVisualiser
from cArmKinematics import cArmKinematics
import numpy as np

# ************************** cInverseKinematics.py **************************************

# Filename:         cInverseKinematics.py
# Author:           Alfie

# Description:      The file defines the cInverseKinematics class which solves for the
#                   unknown joint angles Theta1 and Theta2 of the 4 degrees of freedom
#                   robotic arm, given the required end effector position defined by:
#                   (Xe, Ye) and the extent of extrusion of the prismatic joints defined
#                   by S1 and S4.
#
#                   The method mComputeIK attempts to solve the inverse kinematic equations
#                   and returns a list of possible Theta1 and Theta2 angles to satisfy the
#                   end effector position (Xe, Ye) given S1 and S4.

# Dependencies:     sympy   cArmVisualiser  cArmKinematics  numpy

# ***********************************************************************************

class cInverseKinematics:

    # Define symbolic variables
    Theta1, Theta2 = symbols('Theta1 Theta2')
    L2, L3 = 600, 450   # Link lengths

    def __init__(self, Xe, Ye, S1, S4):
        self.Xe = Xe
        self.Ye = Ye
        self.S1 = S1
        self.S4 = S4

    def mComputeIK(self):

        # Define inverse kinematics equations
        Eq1 = Eq(self.Xe, self.L2*cos(self.Theta1) + (self.L3 + self.S4)*cos(self.Theta1 + self.Theta2))
        Eq1 = Eq(self.Ye, self.S1 + self.L2*sin(self.Theta1) + (self.L3 + self.S4)*sin(self.Theta1 + self.Theta2))

        # Solve the system
        Solution = solve((Eq1, Eq1), (self.Theta1, self.Theta2))

        #   Returns a list of sets [{Theta1, Theta2}, {Theta1, Theta2}, ...]
        #   Returns an empty list [] if no solutions are found
        return Solution

```

```python
from cDHTable import cDHTable
from cArmKinematics import cArmKinematics
from cArmVisualiser import cArmVisualiser
from cWorkspacePlotter import cWorkspacePlotter
from cInverseKinematics import cInverseKinematics

import math
import numpy as np


# *************************** cWorkspacePlotter.py ***************************************

# Filename:         main.py
# Author:           Alfie

# Description:      The file is the main code of the Question2 robot arm simulator,
#                   it plots the robot’s frames in a 3D plot using the cArmVisualiser
#                   class instantiation. cArmVisualiser receives the transformation matrices
#                   it requires from the cDHTable class instantiation, which receives the
#                   required joint angles and extrution lengths.
#
#                   cWorkspacePlotter plots the possible positions of the end effector of
#                   the robot arm by obeying the joint limits provided in its constructor
#
#                   ** The robot arm is plotted in a 3D plot but only exists in the 2D plane
#                   X-Z.
#
# Dependencies:     cWorkspacePlotter.py   numpy   ArmKinematics2.py  DHTable2.py
#                   ArmVisualiser2.py       math

# ************************************************************************************

# Suppress scientific notation
np.set_printoptions(suppress=True)

<<<<<<< Updated upstream
# Perform main calculations and operations
=======
#   Performs forward kinematics calculations for the given joint parameters and plots
#   the resultant frames on a 3D figure
>>>>>>> Stashed changes
def PerformCalcs(JointAngles, S1, S4):
    DHTable = cDHTable(JointAngles, S1, S4)
    Kinematics = cArmKinematics(DHTable, len(JointAngles))
    Transforms =  Kinematics.mGetAllJointGlobPose()

    print("Joint Positions: \n", Transforms.round(2), "\n")
    print("End Effector Position: \n",Kinematics.mEndeffectorPosition().round(2), "\n")

<<<<<<< Updated upstream
#Prompt user for input
=======
    Kinematics.mCheckCorrectness()
    Visualiser = cArmVisualiser()
    Visualiser.mPlotUR5e(Transforms)

    if len(JointAngles) == 4:
        Visualiser.PlotObstacle([400,0], 300)
```

```python

    Visualiser.Show()

#   Infinite loop used to interact with the user to select the desired calculation
#   for the desired manipulator system
>>>>>>> Stashed changes
while(1):
    print("Would you like to simulate a 4 DOF or 6 DOF manipualtor? (enter 4 or 6)\n")

    ManType = int(input("DOF: "))

    #   For Question 2 manipulator
    if ManType == 4:
        print("Would you like to perform forward kinematics or inverse kinematics? (enter F or I)")
        KinType = input("Kinematic Type: ")

        #   Performs forward kinematics and plots given joint parameters
        if KinType == "F":

            print("Should I plot the workspace or the frame transformations")
            PlotSpace = input("W for workspace / T for frames: ")

            if PlotSpace == "T":
                print("Enter your manipulator parameters: \n")

                S1 = float(input("S1: "))
                S4 = float(input("S4: "))
                THETA_2  = float(input("THETA_2: "))
                THETA_3  = float(input("THETA_3: "))
                JointAngles = [0,np.radians(THETA_2),np.radians(THETA_3) - math.radians(90),0]
                PerformCalcs(JointAngles, S1, S4)

            elif PlotSpace == "W":

                Restrictions = input("Should I include joint restrictions (Y for yes / N for no): ")

                if Restrictions == "Y":
                    S1Lowlim = float(input("S1 lower limit: "))
                    S1Uplim = float(input("S1 upper limit: "))

                    S4LowLim = float(input("S4 lower limit: "))
                    S4UpLim = float(input("S4 upper limit: "))

                    Theta2LowLim = np.radians(float(input("Theta2 lower Limit: ")))
                    Theta2UpLim = np.radians(float(input("Theta2 upper Limit: ")))

                    Theta3LowLim = np.radians(float(input("Theta3 lower Limit: ")))
                    Theta3UpLim = np.radians(float(input("Theta3 upper Limit: ")))

                    Man4Workspace = cWorkspacePlotter(S1Lowlim, S1Uplim, S4LowLim, S4UpLim, Theta2LowLim, Theta2UpLim, Theta3LowLim, Theta3UpLim)
                    Man4Workspace.mPlotWorkspace()

                elif Restrictions == "N":
                    Man4Workspace = cWorkspacePlotter()
                    Man4Workspace.mPlotWorkspace()




        elif KinType == "I":
            Xe = int(input("Enter x coordinate for end effector (float/int):"))
            Ye = int(input("Enter y coordinate for end effector (float/int):"))
            S1 = int(input("Enter S1 length (float/int):"))
```

```
            S4 = int(input("Enter S4 length (float/int):"))
            IK = cInverseKinematics(Xe, Ye, S1, S4)
            Solutions = IK.mComputeIK()

            for i in range(len(Solutions)):
                print("Solution " + str(i + 1) + ":" + "[" + str(Solutions[i][0].evalf(2)) + "," + str(Solutions[i][1].evalf(2)) + "]" + "\n")

    #   For Question 1 manipulator
    elif ManType == 6:
        JointAngles = [0,0,0,0,0,0]
        S1 = 0
        S4 = 0

        print("Enter the angles for the robot arm in degrees (default = 0 degrees for all).\n")

        JointAngles[0] = np.radians(float(input("Base angle: ")))
        JointAngles[1] = np.radians(float(input("Shoulder angle: ")))
        JointAngles[2] = np.radians(float(input("Elbow angle: ")))
        JointAngles[3] = np.radians(float(input("Wrist1 angle: ")))
        JointAngles[4] = np.radians(float(input("Wrist2 angle: ")))
        JointAngles[5] = np.radians(float(input("Wrist3 angle: ")))

        PerformCalcs(JointAngles, S1, S4)

    else:
        print("Incompatible DOF.")

```

```python
import numpy as np
import math


# *************************** DHTable.py ***************************************

# Filename:       cDHTable.py
# Author:         Alfie

# Description:  This file defines a class that constructs a Denavit-Hartenberg
#               Table, which describes the joints of a robotic limb using the
#               Denavit-Hartenberg notation.

#               The cDHTable class receives joint angle values, Theta, of the robotic
#               limb whilst having pre-existing knowledge of the other dimensional
#               values of the limb required for the DH Table such as Di, Ai, and Alpha
#
#               The cDHTable class returns the transform matrices of each of the frames
#               corresponding to each robotic link through the mConstructHT function.

# Dependencies: numpy   math

# **********************************************************************************


class cDHTable:

    # Constant definitions
    TABLE_COLUMNS = 4
    D = 0
    THETA = 1
    A = 2
    ALPHA = 3

    #Define link lengths
    L2 = 600
    L3 = 450

    # Initialise DH Table
    def __init__(self, JointAngles, S1, S4):
        self.JointAngles = JointAngles
        self.NumJoints = len(JointAngles)
        self.DHTable = np.zeros((self.NumJoints, self.TABLE_COLUMNS))

        if self.NumJoints == 4:
            Di = [S1, 0, 0, self.L3 + S4]
            Ai = [0, self.L2, 0, 0]
            AlphaI = [math.pi/2,0,-(math.pi/2),0]
        elif self.NumJoints == 6:
            Di = [163, 0, 0, 127, 100, 100]
            Ai = [0, -425, -392, 0, 0, 0]
            AlphaI = [(math.pi)/2, 0, 0, (math.pi)/2, -(math.pi/2), 0]
        else:
            print("Incompatible Joint Angle Vector. Incorrect number of angles?")

        for row in range(self.NumJoints):
            self.DHTable[row][self.D] = Di[row]
            self.DHTable[row][self.THETA] = self.JointAngles[row]
            self.DHTable[row][self.A] = Ai[row]
            self.DHTable[row][self.ALPHA] = AlphaI[row]

    #Helper function to get DH parameters
    def mGetDHParameters(self, FrameNum):
```

```
        # Extract parameters
        Theta = self.DHTable[FrameNum][self.THETA]
        Di = self.DHTable[FrameNum][self.D]
        Ai = self.DHTable[FrameNum][self.A]
        Alpha = self.DHTable[FrameNum][self.ALPHA]

        # Compute trigonometric values
        Ct = math.cos(Theta)
        St = math.sin(Theta)
        Ca = math.cos(Alpha)
        Sa = math.sin(Alpha)

        return Ct, St, Ca, Sa, Di, Ai

    # Construct homogeneous transform matrix (either Standard or modified)
    def mConstructHT(self, FrameNum, Standard=True):
        Ct, St, Ca, Sa, Di, Ai = self.mGetDHParameters(FrameNum)

        if Standard:
            SHT = np.array([
                [Ct, -St * Ca, St * Sa, Ai * Ct],
                [St, Ct * Ca, -Ct * Sa, Ai * St],
                [0, Sa, Ca, Di],
                [0, 0, 0, 1]
            ])
            return SHT
        else:
            MHT = np.array([
                [Ct, -St, 0, Ai],
                [St * Ca, Ct * Ca, -Sa, -Sa * Di],
                [St * Sa, Ct * Sa, Ca, Ca * Di],
                [0, 0, 0, 1]
            ])
            return MHT
```

```python
import matplotlib.pyplot as plt
import numpy as np
from cArmKinematics import cArmKinematics
from cDHTable import cDHTable

# *************************** cWorkspacePlotter.py ***************************************

# Filename:           cWorkspacePlotter.py
# Author:             Alfie

# Description:        The file defines the cWorkspacePlotter which visualises on a 2D plot
#                     what positions the end effector of the robot limb can achieve given
#                     a set of joint limits.
#                     This is done by recursively producing transformation matrices of the
#                     robot arm using the DHTable2 class and extracting the end effector
#                     position using the ArmKinematics2 class, for every possible combination
#                     of joint angles and joint extrusions.
#
#                     All values should be in millimeters or radians
#
# Dependencies:       matplotlib.pyplot   numpy   ArmKinematics2  DHTable2

# ************************************************************************************


class cWorkspacePlotter():

    NUM_POINTS = 20

    #   cWorkspacePlotter constructor, receives joint limits
    def __init__(self, S1Lowlim = 0, S1Uplim = 500, S4LowLim = 0, S4UpLim = 50, Theta2LowLim = 0, Theta2UpLim = 2*np.pi, Theta3LowLim = 0, Theta3UpLim = 2*np.pi):
        self.S1Lowlim = S1Lowlim
        self.S1Uplim = S1Uplim

        self.S4LowLim = S4LowLim
        self.S4UpLim = S4UpLim

        self.Theta2LowLim = Theta2LowLim
        self.Theta2UpLim = Theta2UpLim

        self.Theta3LowLim = Theta3LowLim
        self.Theta3UpLim = Theta3UpLim

    def mPlotWorkspace(self):
        EndEffectorPosition = []

        # Loop over all the parameters to compute end effector positions
        for S1 in np.linspace(self.S1Lowlim, self.S1Uplim, self.NUM_POINTS):

            for S2 in np.linspace(self.S4LowLim, self.S4UpLim, self.NUM_POINTS):

                for Theta2 in np.linspace(self.Theta2LowLim, self.Theta2UpLim, self.NUM_POINTS):

                    for Theta3 in np.linspace(self.Theta3UpLim, self.Theta3LowLim, self.NUM_POINTS):

                        JointAngles = [0, Theta2, Theta3 - np.pi/2, 0]
                        Table = cDHTable(JointAngles, S1, S2)

```

```
                        Kinematics = cArmKinematics(Table, 4)
                        Kinematics.mGetAllJointGlobPose()
                        EndEffectorPosition.append(Kinematics.mEndeffectorPosition())

        EndEffectorPosition = np.array(EndEffectorPosition)

        Xvals = EndEffectorPosition[:, 0]  # x-position
        Zvals = EndEffectorPosition[:, 2]  # z-position

        plt.scatter(Xvals, Zvals, color='blue', marker='o', label="End Effector Position")

        # Add labels and grid
        plt.xlabel("X position (mm)")
        plt.ylabel("Z position (mm)")
        plt.title("Workspace Plot in x-z Plane")
        plt.grid(True)
        plt.legend()

        # Show the plot
        plt.show()
```

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# *************************** cArmVisualiser.py *************************************

# Filename:       cArmVisualiser.py
# Author:         Jestin

# Description:  This file defines the cArmVisualiser class which plots the reference
#               frames of the robot arm on a matplotlib figure.
#               Using the mPlotUR5e method it receives a list of 4x4 transform matrices
#               and extracts the rotation matrix and global position vectors to visulise the
#               frames on a 3D graph

# Dependencies: numpy   matplotlib.pyplot   mpl_toolkits.mplot3d

# *********************************************************************************


class cArmVisualiser:

    # Instantiates the cArmVisualiser class and initializes a matplotlib 3D figure
    def __init__(self):
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111, projection='3d')

    # Plots the frames
    # Transformations -> list of 4x4 transformation matrices
    def mPlotUR5e(self, Transformations):

        # Initialize origin and frame vectors (as 1D arrays)
        Origin = np.array([0, 0, 0])
        OriginPrev = np.array([0, 0, 0])

        # Frame vectors representing X, Y, Z axes
        FrameArrowI = np.array([50, 0, 0])  # X-axis (red)
        FrameArrowJ = np.array([0, 50, 0])  # Y-axis (green)
        FrameArrowK = np.array([0, 0, 50])  # Z-axis (blue)

        # Plot the axes for the global frame (origin)
        self.ax.quiver(*Origin, *FrameArrowI, color='r', label="X-axis")  # x-axis in red
        self.ax.quiver(*Origin, *FrameArrowJ, color='g', label="Y-axis")  # y-axis in green
        self.ax.quiver(*Origin, *FrameArrowK, color='b', label="Z-axis")  # z-axis in blue

        # Annotate the initial frame (frame 0) at origin
        self.ax.text(Origin[0], Origin[1], Origin[2], f"Frame 0", color='black', fontsize=10, ha='center')

        # Loop through Transformations to plot subsequent frames
        for i, T in enumerate(Transformations):
            # Extract position (Origin) from transformation matrix
            Origin = T[:3, 3]

            # Apply rotation matrix to frame vectors
            FrameArrowI = T[:3, :3] @ np.array([50, 0, 0])  # X-axis vector in this frame
            FrameArrowJ = T[:3, :3] @ np.array([0, 50, 0])  # Y-axis vector in this frame
            FrameArrowK = T[:3, :3] @ np.array([0, 0, 50])  # Z-axis vector in this
```

```
frame

            # Plot the axes for the current frame
            if i == 0:
                self.ax.quiver(*Origin, *FrameArrowI, color='r', label="X-axis")
                self.ax.quiver(*Origin, *FrameArrowJ, color='g', label="Y-axis")
                self.ax.quiver(*Origin, *FrameArrowK, color='b', label="Z-axis")
            else:
                self.ax.quiver(*Origin, *FrameArrowI, color='r')
                self.ax.quiver(*Origin, *FrameArrowJ, color='g')
                self.ax.quiver(*Origin, *FrameArrowK, color='b')

            # Plot the line connecting the previous frame to the current one
            self.ax.plot([OriginPrev[0], Origin[0]], [OriginPrev[1], Origin[1]], [OriginPrev[2], Origin[2]],
                         marker='o', linestyle='-', color='purple', linewidth=2)

            # Update previous origin for the next iteration
            OriginPrev = np.copy(Origin)

        # Set limits and labels for the 3D graph
        self.ax.set_xlim([0, 1000])
        self.ax.set_ylim([0, 1000])
        self.ax.set_zlim([0, 1000])
        self.ax.set_xlabel('X')
        self.ax.set_ylabel('Y')
        self.ax.set_zlabel('Z')


    #   Plots the circular obstacle on the X-Z plane
    def PlotObstacle(self, Position, Radius):

        Theta = np.linspace(0, 2*np.pi, 100)    #   Angle values

        #   Creating circle coordinates in the XY plane
        x = Position[0] + Radius * np.cos(Theta)
        y = np.zeros_like(x)
        z = Position[1] + Radius * np.sin(Theta)

        #   Plots the circle
        self.ax.plot(x, y, z, linestyle='-', color='yellow', linewidth=2)

        print("circle should've plotted")


    #   Shows the plots on the figure
    def Show(self):
        plt.show()


```

```python
import numpy as np

# *************************** cArmKinematics.py ***************************************

# Filename:       cArmVisualiser.py
# Author:         Alfie

# Description:  This file defines the cArmKinematics class which finds the
#               end effector position of the robot arm as well as the homogeneous
#               transforms from the origin to each joint. It also checks for singularities
#               using the Jacobian matrix.

# Dependencies: numpy

# ***********************************************************************************


class cArmKinematics:

    TRANSFORM_DIM = 4
    WORKSPACE_DIM = 3

    # Class constructor
    def __init__(self, DHTable, NumJoints):
        self.DHTable = DHTable
        self.NumJoints = NumJoints


    #Check for singularites using Jacobian matrices
    def mCheckCorrectness(self):
        #Initialise the position and rotation vectors
        PositionVector = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])
        RotationVector = np.array([np.eye(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])
        LinearVelocity = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])
        AngularVelocity = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.NumJoints)])

        #Obtain the position, rotation, linear velocity and angular velocity vectors from the joint pose
        for FrameNum in range(self.NumJoints):
            PositionVector[FrameNum] = self.JointPoseGlob[FrameNum][:self.WORKSPACE_DIM, -1]
            RotationVector[FrameNum] = self.JointPoseGlob[FrameNum][:self.WORKSPACE_DIM, :self.WORKSPACE_DIM]
            AngularVelocity[FrameNum] = RotationVector[FrameNum][:, -1]
            LinearVelocity[FrameNum] = np.cross(AngularVelocity[FrameNum], PositionVector[-1] - PositionVector[FrameNum])

        J = np.zeros((self.WORKSPACE_DIM + self.WORKSPACE_DIM, self.NumJoints))

        for FrameNum in range(self.NumJoints):
            J[:self.WORKSPACE_DIM, FrameNum] = LinearVelocity[FrameNum]
            J[self.WORKSPACE_DIM:, FrameNum] = AngularVelocity[FrameNum]


        #Check for singularites
        Singular = np.linalg.matrix_rank(J) < min(J.shape)

        if Singular:
            print("Singularities detected")

        else:
```

```python
            print("No singularities detected")


    def mEndeffectorPosition(self):
        return self.JointPoseGlob[-1][:self.WORKSPACE_DIM, -1]


    # Determine the final end effector position
    def mGetAllJointGlobPose(self):
        self.JointPoseGlob = np.array([np.eye(self.TRANSFORM_DIM) for _ in range(self.NumJoints)])

        for FrameNum in range(self.NumJoints):
            self.JointPoseGlob[FrameNum] = self.JointPoseGlob[FrameNum-1]@self.DHTable.mConstructHT(FrameNum, Standard=True)

        return self.JointPoseGlob
```

```python
from sympy import symbols, Eq, solve, sqrt, atan2, acos, cos, sin, pprint
from cArmVisualiser import cArmVisualiser
from cArmKinematics import cArmKinematics
import numpy as np

# ************************ cInverseKinematics.py ************************************

# Filename:          cInverseKinematics.py
# Author:            Alfie

# Description:       The file defines the cInverseKinematics class which solves for the
#                    unknown joint angles Theta1 and Theta2 of the 4 degrees of freedom
#                    robotic arm, given the required end effector position defined by:
#                    (Xe, Ye) and the extent of extrusion of the prismatic joints defined
#                    by S1 and S4.
#
#                    The method mComputeIK attempts to solve the inverse kinematic equations
#                    and returns a list of possible Theta1 and Theta2 angles to satisfy the
#                    end effector position (Xe, Ye) given S1 and S4.

# Dependencies:      sympy   cArmVisualiser  cArmKinematics  numpy

# ***********************************************************************************

class cInverseKinematics:

    # Define symbolic variables
    Theta1, Theta2 = symbols('Theta1 Theta2')
    L2, L3 = 600, 450    # Link lengths

    def __init__(self, Xe, Ye, S1, S4):
        self.Xe = Xe
        self.Ye = Ye
        self.S1 = S1
        self.S4 = S4

    def mComputeIK(self):

        # Define inverse kinematics equations
        Eq1 = Eq(self.Xe, self.L2*cos(self.Theta1) + (self.L3 + self.S4)*cos(self.Theta1 + self.Theta2))
        Eq1 = Eq(self.Ye, self.S1 + self.L2*sin(self.Theta1) + (self.L3 + self.S4)*sin(self.Theta1 + self.Theta2))

        # Solve the system
        Solution = solve((Eq1, Eq1), (self.Theta1, self.Theta2))

        #   Returns a list of sets [{Theta1, Theta2}, {Theta1, Theta2}, ...]
        #   Returns an empty list [] if no solutions are found
        return Solution

```

```
from cDHTable import cDHTable
from cArmKinematics import cArmKinematics
from cArmVisualiser import cArmVisualiser
from cWorkspacePlotter import cWorkspacePlotter
from cInverseKinematics import cInverseKinematics

import math
import numpy as np


# *************************** cWorkspacePlotter.py ***************************************

# Filename:         main.py
# Author:           Alfie

# Description:      The file is the main code of the Question2 robot arm simulator,
#                   it plots the robot's frames in a 3D plot using the cArmVisualiser
#                   class instantiation. cArmVisualiser receives the transformation matrices
#                   it requires from the cDHTable class instantiation, which receives the
#                   required joint angles and extrution lengths.
#
#                   cWorkspacePlotter plots the possible positions of the end effector of
#                   the robot arm by obeying the joint limits provided in its constructor
#
#                   ** The robot arm is plotted in a 3D plot but only exists in the 2D plane
#                   X-Z.
#
# Dependencies:     cWorkspacePlotter.py   numpy   ArmKinematics2.py  DHTable2.py
#                   ArmVisualiser2.py       math

# ***********************************************************************************

# Suppress scientific notation
np.set_printoptions(suppress=True)

<<<<<<< Updated upstream
# Perform main calculations and operations
=======
#   Performs forward kinematics calculations for the given joint parameters and plots
#   the resultant frames on a 3D figure
>>>>>>> Stashed changes
def PerformCalcs(JointAngles, S1, S4):
    DHTable = cDHTable(JointAngles, S1, S4)
    Kinematics = cArmKinematics(DHTable, len(JointAngles))
    Transforms =  Kinematics.mGetAllJointGlobPose()

    print("Joint Positions: \n", Transforms.round(2), "\n")
    print("End Effector Position: \n",Kinematics.mEndeffectorPosition().round(2), "\n")

<<<<<<< Updated upstream
#Prompt user for input
=======
    Kinematics.mCheckCorrectness()
    Visualiser = cArmVisualiser()
    Visualiser.mPlotUR5e(Transforms)

    if len(JointAngles) == 4:
        Visualiser.PlotObstacle([400,0], 300)
```

```

    Visualiser.Show()

#   Infinite loop used to interact with the user to select the desired calculation
#   for the desired manipulator system
>>>>>>> Stashed changes
while(1):
    print("Would you like to simulate a 4 DOF or 6 DOF manipualtor? (enter 4 or 6)\n")

    ManType = int(input("DOF: "))

    #   For Question 2 manipulator
    if ManType == 4:
        print("Would you like to perform forward kinematics or inverse kinematics? (enter F or I)")
        KinType = input("Kinematic Type: ")

        #   Performs forward kinematics and plots given joint parameters
        if KinType == "F":

            print("Should I plot the workspace or the frame transformations")
            PlotSpace = input("W for workspace / T for frames: ")

            if PlotSpace == "T":
                print("Enter your manipulator parameters: \n")

                S1 = float(input("S1: "))
                S4 = float(input("S4: "))
                THETA_2  = float(input("THETA_2: "))
                THETA_3  = float(input("THETA_3: "))
                JointAngles = [0,np.radians(THETA_2),np.radians(THETA_3) - math.radians(90),0]
                PerformCalcs(JointAngles, S1, S4)

            elif PlotSpace == "W":

                Restrictions = input("Should I include joint restrictions (Y for yes / N for no): ")

                if Restrictions == "Y":
                    S1Lowlim = float(input("S1 lower limit: "))
                    S1Uplim = float(input("S1 upper limit: "))

                    S4LowLim = float(input("S4 lower limit: "))
                    S4UpLim = float(input("S4 upper limit: "))

                    Theta2LowLim = np.radians(float(input("Theta2 lower Limit: ")))
                    Theta2UpLim = np.radians(float(input("Theta2 upper Limit: ")))

                    Theta3LowLim = np.radians(float(input("Theta3 lower Limit: ")))
                    Theta3UpLim = np.radians(float(input("Theta3 upper Limit: ")))

                    Man4Workspace = cWorkspacePlotter(S1Lowlim, S1Uplim, S4LowLim, S4UpLim, Theta2LowLim, Theta2UpLim, Theta3LowLim, Theta3UpLim)
                    Man4Workspace.mPlotWorkspace()

                elif Restrictions == "N":
                    Man4Workspace = cWorkspacePlotter()
                    Man4Workspace.mPlotWorkspace()




        elif KinType == "I":
            Xe = int(input("Enter x coordinate for end effector (float/int):"))
            Ye = int(input("Enter y coordinate for end effector (float/int):"))
            S1 = int(input("Enter S1 length (float/int):"))
```

```
            S4 = int(input("Enter S4 length (float/int):"))
            IK = cInverseKinematics(Xe, Ye, S1, S4)
            Solutions = IK.mComputeIK()

            for i in range(len(Solutions)):
                print("Solution " + str(i + 1) + ":" + "[" + str(Solutions[i][0].evalf(2)) + "," + str(Solutions[i][1].evalf(2)) + "]" + "\n")

    #   For Question 1 manipulator
    elif ManType == 6:
        JointAngles = [0,0,0,0,0,0]
        S1 = 0
        S4 = 0

        print("Enter the angles for the robot arm in degrees (default = 0 degrees for all).\n")

        JointAngles[0] = np.radians(float(input("Base angle: ")))
        JointAngles[1] = np.radians(float(input("Shoulder angle: ")))
        JointAngles[2] = np.radians(float(input("Elbow angle: ")))
        JointAngles[3] = np.radians(float(input("Wrist1 angle: ")))
        JointAngles[4] = np.radians(float(input("Wrist2 angle: ")))
        JointAngles[5] = np.radians(float(input("Wrist3 angle: ")))

        PerformCalcs(JointAngles, S1, S4)

    else:
        print("Incompatible DOF.")

```

```python
import numpy as np
import math


# ************************** DHTable.py **************************************

# Filename:       cDHTable.py
# Author:         Alfie

# Description:  This file defines a class that constructs a Denavit-Hartenberg
#               Table, which describes the joints of a robotic limb using the
#               Denavit-Hartenberg notation.

#               The cDHTable class receives joint angle values, Theta, of the robotic
#               limb whilst having pre-existing knowledge of the other dimensional
#               values of the limb required for the DH Table such as Di, Ai, and Alpha
#
#               The cDHTable class returns the transform matrices of each of the frames
#               corresponding to each robotic link through the mConstructHT function.

# Dependencies: numpy   math

# *****************************************************************************


class cDHTable:

    # Constant definitions
    TABLE_COLUMNS = 4
    D = 0
    THETA = 1
    A = 2
    ALPHA = 3

    #Define link lengths
    L2 = 600
    L3 = 450

    # Initialise DH Table
    def __init__(self, JointAngles, S1, S4):
        self.JointAngles = JointAngles
        self.NumJoints = len(JointAngles)
        self.DHTable = np.zeros((self.NumJoints, self.TABLE_COLUMNS))

        if self.NumJoints == 4:
            Di = [S1, 0, 0, self.L3 + S4]
            Ai = [0, self.L2, 0, 0]
            AlphaI = [math.pi/2,0,-(math.pi/2),0]
        elif self.NumJoints == 6:
            Di = [163, 0, 0, 127, 100, 100]
            Ai = [0, -425, -392, 0, 0, 0]
            AlphaI = [(math.pi)/2, 0, 0, (math.pi)/2, -(math.pi/2), 0]
        else:
            print("Incompatible Joint Angle Vector. Incorrect number of angles?")

        for row in range(self.NumJoints):
            self.DHTable[row][self.D] = Di[row]
            self.DHTable[row][self.THETA] = self.JointAngles[row]
            self.DHTable[row][self.A] = Ai[row]
            self.DHTable[row][self.ALPHA] = AlphaI[row]

    #Helper function to get DH parameters
    def mGetDHParameters(self, FrameNum):
```

```
        # Extract parameters
        Theta = self.DHTable[FrameNum][self.THETA]
        Di = self.DHTable[FrameNum][self.D]
        Ai = self.DHTable[FrameNum][self.A]
        Alpha = self.DHTable[FrameNum][self.ALPHA]

        # Compute trigonometric values
        Ct = math.cos(Theta)
        St = math.sin(Theta)
        Ca = math.cos(Alpha)
        Sa = math.sin(Alpha)

        return Ct, St, Ca, Sa, Di, Ai

    # Construct homogeneous transform matrix (either Standard or modified)
    def mConstructHT(self, FrameNum, Standard=True):
        Ct, St, Ca, Sa, Di, Ai = self.mGetDHParameters(FrameNum)

        if Standard:
            SHT = np.array([
                [Ct, -St * Ca, St * Sa, Ai * Ct],
                [St, Ct * Ca, -Ct * Sa, Ai * St],
                [0, Sa, Ca, Di],
                [0, 0, 0, 1]
            ])
            return SHT
        else:
            MHT = np.array([
                [Ct, -St, 0, Ai],
                [St * Ca, Ct * Ca, -Sa, -Sa * Di],
                [St * Sa, Ct * Sa, Ca, Ca * Di],
                [0, 0, 0, 1]
            ])
            return MHT
```

```
import matplotlib.pyplot as plt
import numpy as np
from cArmKinematics import cArmKinematics
from cDHTable import cDHTable

# ************************** cWorkspacePlotter.py ****************************************

# Filename:          cWorkspacePlotter.py
# Author:            Alfie

# Description:       The file defines the cWorkspacePlotter which visualises on a 2D plot
#                    what positions the end effector of the robot limb can achieve given
#                    a set of joint limits.
#                    This is done by recursively producing transformation matrices of the
#                    robot arm using the DHTable2 class and extracting the end effector
#                    position using the ArmKinematics2 class, for every possible combination
#                    of joint angles and joint extrusions.
#
#                    All values should be in millimeters or radians
#
# Dependencies:      matplotlib.pyplot   numpy   ArmKinematics2  DHTable2

# **********************************************************************************

class cWorkspacePlotter():

    NUM_POINTS = 20

    #   cWorkspacePlotter constructor, receives joint limits
    def __init__(self, S1Lowlim = 0, S1Uplim = 500, S4LowLim = 0, S4UpLim = 50, Theta2LowLim = 0, Theta2UpLim = 2*np.pi, Theta3LowLim = 0, Theta3UpLim = 2*np.pi):
        self.S1Lowlim = S1Lowlim
        self.S1Uplim = S1Uplim

        self.S4LowLim = S4LowLim
        self.S4UpLim = S4UpLim

        self.Theta2LowLim = Theta2LowLim
        self.Theta2UpLim = Theta2UpLim

        self.Theta3LowLim = Theta3LowLim
        self.Theta3UpLim = Theta3UpLim

    def mPlotWorkspace(self):
        EndEffectorPosition = []

        # Loop over all the parameters to compute end effector positions
        for S1 in np.linspace(self.S1Lowlim, self.S1Uplim, self.NUM_POINTS):

            for S2 in np.linspace(self.S4LowLim, self.S4UpLim, self.NUM_POINTS):

                for Theta2 in np.linspace(self.Theta2LowLim, self.Theta2UpLim, self.NUM_POINTS):

                    for Theta3 in np.linspace(self.Theta3UpLim, self.Theta3LowLim, self.NUM_POINTS):

                        JointAngles = [0, Theta2, Theta3 - np.pi/2, 0]
                        Table = cDHTable(JointAngles, S1, S2)

```

```
                        Kinematics = cArmKinematics(Table, 4)
                        Kinematics.mGetAllJointGlobPose()
                        EndEffectorPosition.append(Kinematics.mEndeffectorPosition())

        EndEffectorPosition = np.array(EndEffectorPosition)

        Xvals = EndEffectorPosition[:, 0]  # x-position
        Zvals = EndEffectorPosition[:, 2]  # z-position

        plt.scatter(Xvals, Zvals, color='blue', marker='o', label="End Effector Position")

        # Add labels and grid
        plt.xlabel("X position (mm)")
        plt.ylabel("Z position (mm)")
        plt.title("Workspace Plot in x-z Plane")
        plt.grid(True)
        plt.legend()

        # Show the plot
        plt.show()
```

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# *************************** cArmVisualiser.py *****************************************

# Filename:       cArmVisualiser.py
# Author:         Jestin

# Description:  This file defines the cArmVisualiser class which plots the reference
#               frames of the robot arm on a matplotlib figure.
#               Using the mPlotUR5e method it receives a list of 4x4 transform matrices
#               and extracts the rotation matrix and global position vectors to visulise the
#               frames on a 3D graph

# Dependencies: numpy   matplotlib.pyplot   mpl_toolkits.mplot3d

# *******************************************************************************************


class cArmVisualiser:

    # Instantiates the cArmVisualiser class and initializes a matplotlib 3D figure
    def __init__(self):
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111, projection='3d')

    # Plots the frames
    # Transformations -> list of 4x4 transformation matrices
    def mPlotUR5e(self, Transformations):

        # Initialize origin and frame vectors (as 1D arrays)
        Origin = np.array([0, 0, 0])
        OriginPrev = np.array([0, 0, 0])

        # Frame vectors representing X, Y, Z axes
        FrameArrowI = np.array([50, 0, 0])  # X-axis (red)
        FrameArrowJ = np.array([0, 50, 0])  # Y-axis (green)
        FrameArrowK = np.array([0, 0, 50])  # Z-axis (blue)

        # Plot the axes for the global frame (origin)
        self.ax.quiver(*Origin, *FrameArrowI, color='r', label="X-axis")  # x-axis in red
        self.ax.quiver(*Origin, *FrameArrowJ, color='g', label="Y-axis")  # y-axis in green
        self.ax.quiver(*Origin, *FrameArrowK, color='b', label="Z-axis")  # z-axis in blue

        # Annotate the initial frame (frame 0) at origin
        self.ax.text(Origin[0], Origin[1], Origin[2], f"Frame 0", color='black', fontsize=10, ha='center')

        # Loop through Transformations to plot subsequent frames
        for i, T in enumerate(Transformations):
            # Extract position (Origin) from transformation matrix
            Origin = T[:3, 3]

            # Apply rotation matrix to frame vectors
            FrameArrowI = T[:3, :3] @ np.array([50, 0, 0])  # X-axis vector in this frame
            FrameArrowJ = T[:3, :3] @ np.array([0, 50, 0])  # Y-axis vector in this frame
            FrameArrowK = T[:3, :3] @ np.array([0, 0, 50])  # Z-axis vector in this
```

```
frame

            # Plot the axes for the current frame
            if i == 0:
                self.ax.quiver(*Origin, *FrameArrowI, color='r', label="X-axis")
                self.ax.quiver(*Origin, *FrameArrowJ, color='g', label="Y-axis")
                self.ax.quiver(*Origin, *FrameArrowK, color='b', label="Z-axis")
            else:
                self.ax.quiver(*Origin, *FrameArrowI, color='r')
                self.ax.quiver(*Origin, *FrameArrowJ, color='g')
                self.ax.quiver(*Origin, *FrameArrowK, color='b')

            # Plot the line connecting the previous frame to the current one
            self.ax.plot([OriginPrev[0], Origin[0]], [OriginPrev[1], Origin[1]], [OriginPrev[2], Origin[2]],
                         marker='o', linestyle='-', color='purple', linewidth=2)

            # Update previous origin for the next iteration
            OriginPrev = np.copy(Origin)

        # Set limits and labels for the 3D graph
        self.ax.set_xlim([0, 1000])
        self.ax.set_ylim([0, 1000])
        self.ax.set_zlim([0, 1000])
        self.ax.set_xlabel('X')
        self.ax.set_ylabel('Y')
        self.ax.set_zlabel('Z')


    #   Plots the circular obstacle on the X-Z plane
    def PlotObstacle(self, Position, Radius):

        Theta = np.linspace(0, 2*np.pi, 100)    #   Angle values

        #   Creating circle coordinates in the XY plane
        x = Position[0] + Radius * np.cos(Theta)
        y = np.zeros_like(x)
        z = Position[1] + Radius * np.sin(Theta)

        #   Plots the circle
        self.ax.plot(x, y, z, linestyle='-', color='yellow', linewidth=2)

        print("circle should've plotted")


    #   Shows the plots on the figure
    def Show(self):
        plt.show()
```