```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, num_joints):
        self.DHTable = DHTable
        self.num_joints = num_joints


    #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.num_joints)])
        RotationVector = np.array([np.eye(self.WORKSPACE_DIM) for _ in range(self.num_joints)])
        LinearVelocity = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.num_joints)])
        AngularVelocity = np.array([np.zeros(self.WORKSPACE_DIM) for _ in range(self.num_joints)])

        #Obtain the position, rotation, linear velocity and angular velocity vectors from the joint pose
        for FrameNum in range(self.num_joints):
            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.num_joints))
        for FrameNum in range(self.num_joints):
            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(sel
f.num_joints)])

        for FrameNum in range(self.num_joints):
            self.jointPoseGlob[FrameNum] = self.jointPoseGlob[FrameNum-1]@self.DHTab
le.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

class cInverseKinematics:

    # Define symbolic variables
    theta_1, theta_2 = symbols('theta_1 theta_2')
    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 ComputeIK(self):

        # Define inverse kinematics equations
        eq1 = Eq(self.xe, self.L2*cos(self.theta_1) + (self.L3 + self.s4)*cos(self.theta_1 + self.theta_2))
        eq2 = Eq(self.ye, self.s1 + self.L2*sin(self.theta_1) + (self.L3 + self.s4)*sin(self.theta_1 + self.theta_2))

        # Solve the system
        solution = solve((eq1, eq2), (self.theta_1, self.theta_2))

        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:         main2.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)

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")
    Kinematics.mCheckCorrectness()
    visualiser = cArmVisualiser()
    visualiser.mPlotUR5e(Transforms)
    if len(JointAngles) == 4:
        visualiser.PlotObstacle([400,0], 300)
    visualiser.Show()

while(1):
    print("Would you like to simulate a 4 DOF or 6 DOF manipualtor? (enter 4 or 6)\n")
    ManType = int(input("DOF: "))

    if ManType == 4:
        print("Would you like to perform forward kinematics or inverse kinematics? (enter F or I)")
        KinType = input("Kinematic Type: ")
        if KinType == "F":
            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 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.ComputeIK()
            for i in range(len(solutions)):
                print("Solution " + str(i + 1) + ":" + "[" + str(solutions[i][0].evalf(2)) + "," + str(solutions[i][1].evalf(2)) + "]" + "\n")

    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.num_joints = len(JointAngles)
        self.DHTable = np.zeros((self.num_joints, self.TABLE_COLUMNS))

        if self.num_joints == 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.num_joints == 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.num_joints):
            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):
```

```python
        # 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.
#
#
# Dependencies:      matplotlib.pyplot   numpy   ArmKinematics2  DHTable2

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


class cWorkspacePlotter():

    NUM_POINTS = 20

    #   cWorkspacePlotter constructor, receives joint limits
    def __init__(self, S1Lowlim, S1Uplim, S4LowLim, S4UpLim, Theta2LowLim, Theta2UpLim, Theta3LowLim, Theta3UpLim):
        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)
```

```
                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()

```