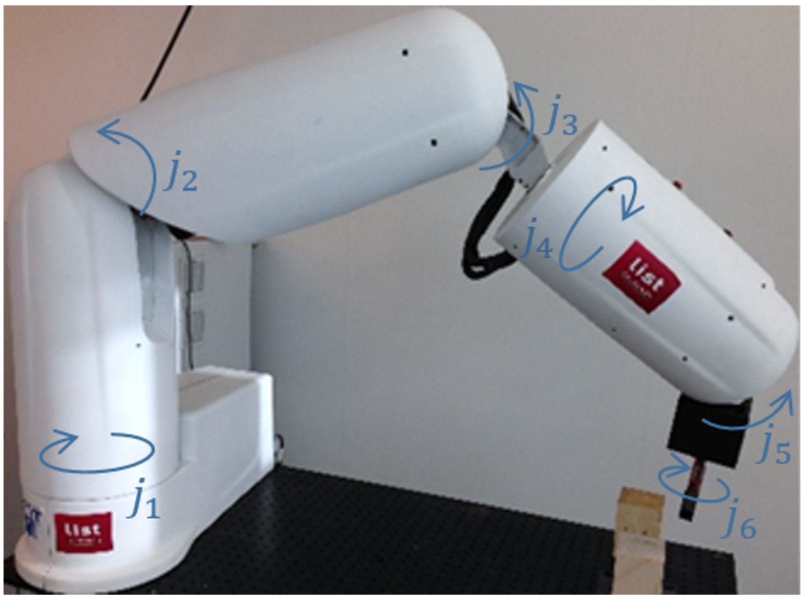
ROBOTICS - Tutorial 1 : Direct and inverse kinematics

Introduction

We propose to study the **geometric** and **kinematic** modeling of a manipulator arm developed by the *Interactive Robotics Laboratory* of the *CEA List*. This robot, which kinematic chain is of serial type, has 6 revolute joints (j_i with $i = 1, \ldots, 6$).



The numerical values of the robot parameters, required for the completion of this tutorial, are specified in the following table.

Table. Numerical values of the robot parameters.

Parameters	Numerical values	Type of parameter
d_3	$0.7m$	Geometric parameter
r_1	$0.5m$	Geometric parameter
r_4	$0.2m$	Geometric parameter
r_E	$0.1m$	Geometric parameter

The use of *Python* is required to perform the tutorial. Please import the following required mathematical libraires to start the tutorial.

In [1...

```python
import numpy as np
import math as m
import functools as fu
from numpy.linalg import eig
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random

%matplotlib inline
```

In the following, you will progressively update a *Dictionnary* in Python containing the robot parameters, named **robotParameters**.

Please initialize it as follows: *robotParameters = { 'nJoints': 6, 'jointsType': ['R','R','R','R','R','R']}*

In [1...
```
robotParameters = { 'nJoints': 6, 'jointsType': ['R','R','R','R','R','R']}
```

You will also progressively build a *Class* containing some *attributes* related to the robot. To do so, you will be asked to program some of its *methods* in the tutorial. This class is named **RobotModel** and is defined in the file *ClassRobotModel*.

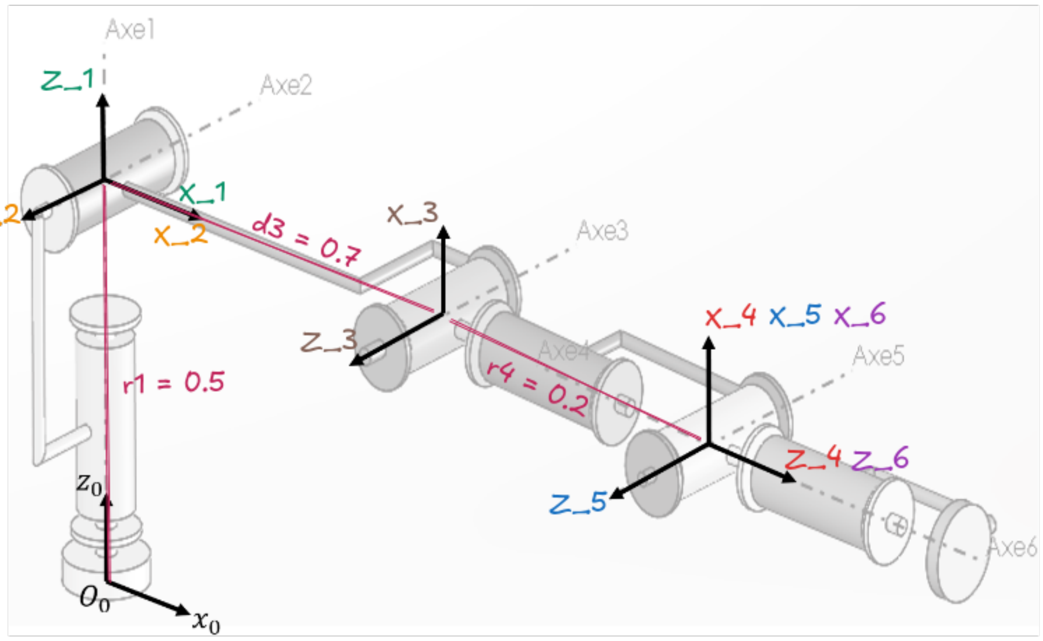
Please initialize it as follows. You will see printing the resulting *attributes* of the *Class* **RobotModel**.

In [1...
```
from ClassRobotModel import RobotModel
RobotTutorials = RobotModel( **robotParameters )
```

```
Attribute (int): self.numberJoints =  6
Attribute (list): self.jointsType =  ['R', 'R', 'R', 'R', 'R', 'R']
Attribute (list - 0 if self.jointsType[i] == 'R' / 1 if self.jointsType[i]
== 'P'): self.sigma =  [0, 0, 0, 0, 0, 0]
```

Direct geometric model

Q1. *Modified Denavit-Hartenberg (MDH)* parameters defining the spatial arrangement of the robot structure.



Q2. Geometric parameters of the robot:

i	α_i	d_i	θ_i	r_i
1	0	0	0	0.5
2	$pi/2$	0	0	0
3	0	0.7	$pi/2$	0
4	$pi/2$	0	0	0.2
5	$-pi/2$	0	0	0
6	$pi/2$	0	0	0

also in *DHM_parameters.txt*

In [1...
```
robotParameters['fileDHM'] = "DHM_parameters.txt"
RobotTutorials = RobotModel( **robotParameters )
```

```
Attribute (int): self.numberJoints =  6
Attribute (list): self.jointsType =  ['R', 'R', 'R', 'R', 'R', 'R']
Attribute (list - 0 if self.jointsType[i] == 'R' / 1 if self.jointsType[i]
== 'P'): self.sigma =  [0, 0, 0, 0, 0, 0]
Attribute (list - float): self.tableDHM =  [[1.0, 0.0, 0.0, 0.0, 0.5],
[2.0, 1.5707963267948966, 0.0, 0.0, 0.0], [3.0, 0.0, 0.7, 1.57079632679489
66, 0.0], [4.0, 1.5707963267948966, 0.0, 0.0, 0.2], [5.0, -1.5707963267948
966, 0.0, 0.0, 0.0], [6.0, 1.5707963267948966, 0.0, 0.0, 0.0]]
```

Q3-a. Write a generic function $TransformMatElem(\alpha_i, d_i, \theta_i, r_i)$ which output argument is the homogeneous transform matrix *g* between two successive frames.

In [1...
```
@staticmethod
def TransformMatElem(alpha_i, d_i, theta_i, r_i):
    """
    Computation of the homogeneous transform matrix between two successive

    Input:
        - Four scalar parameters given by the Modified Denavit-Hartenberg

    Output:
        - Homogeneous transform matrix g_(i-1,i) as a "np.array"
    """

    ca = np.cos(alpha_i)
    sa = np.sin(alpha_i)
    ct = np.cos(theta_i)
    st = np.sin(theta_i)

    return np.array([
        [ct,      -st,     0,      d_i],
        [st*ca,   ct*ca,   -sa,    -sa*r_i],
        [st*sa,   ct*sa,   ca,     ca*r_i],
        [0,       0,       0,      1]
    ])

RobotModel.TransformMatElem = TransformMatElem
```

Q3-b. Write a function $ComputeDGM(\text{self}, q)$ which computes the direct geometric model of any robot with series open kinematic chain, taking as input arguments the current configuration.

In [1...
```
def ComputeDGM(self, q_cur):
    """
    Computation of the Direct Geometric Model (DGM) of the robot given by

    Inputs:
        - List of robot's geometric parameters "self.tableDHM" given by th
        - Number of joints of the robot "self.numberJoints"
        - List of type of joints of the robot: "self.sigma"
        - Current joint configuration "q_cur"

    Outputs:
        - List of the successive homogeneous transform matrices: "self.lis
        - List of the successive resulting homogeneous transform matrices
    """

    self.list_g_i_1_i = []

    for i, alpha, d, theta, r in self.tableDHM:
        idx = int(i) - 1
        if self.sigma[idx] == 0:
            d_i = d
            theta_i = theta + q_cur[idx]
        else:
            d_i = d + q_cur[idx]
            theta_i = theta

        self.list_g_i_1_i.append(self.TransformMatElem(alpha, d_i, theta_i

    self.list_g_0i = [self.list_g_i_1_i[0]]
    for g_i_1_i in self.list_g_i_1_i[1:]:
        self.list_g_0i.append(self.list_g_0i[-1] @ g_i_1_i)

    return self.list_g_i_1_i, self.list_g_0i

RobotModel.ComputeDGM = ComputeDGM
```

Q3-c. We consider an end-effector mounted at the end of the robot arm. The frame $\mathcal{R}_E$ attached to the end-effector of the robot is defined by a translation of the frame $\mathcal{R}_6$ by a distance r_E along the z_6 axis.

Specify the four DHM parameters for the tool frame description in the field below.

In [1...
```
robotParameters['toolFrameDHM'] = [0, 0, 0, 0.1]
RobotTutorials = RobotModel( **robotParameters )
```

```
Attribute (int): self.numberJoints =  6
Attribute (list): self.jointsType =  ['R', 'R', 'R', 'R', 'R', 'R']
Attribute (list - 0 if self.jointsType[i] == 'R' / 1 if self.jointsType[i]
== 'P'): self.sigma =  [0, 0, 0, 0, 0, 0]
Attribute (list - float): self.tableDHM =  [[1.0, 0.0, 0.0, 0.0, 0.5],
[2.0, 1.5707963267948966, 0.0, 0.0, 0.0], [3.0, 0.0, 0.7, 1.57079632679489
66, 0.0], [4.0, 1.5707963267948966, 0.0, 0.0, 0.2], [5.0, -1.5707963267948
966, 0.0, 0.0, 0.0], [6.0, 1.5707963267948966, 0.0, 0.0, 0.0]]
Attribute (list - float): self.toolDHM =  [0, 0, 0, 0.1]
```

Using the results of previous questions, write a function $ComputeToolPose(\text{self})$ that computes the homogeneous transform matrix $\bar{g}_{0E}$. This matrix gives the position and the orientation of the frame $\mathcal{R}_E$ attached to the end-effector of the robot, expressed in the base frame $\mathcal{R}_0$.

In [1...
```python
def ComputeToolPose(self):
    """
    Computation of the homogeneous transform matrix g0E which gives the pos
    
    Inputs:
       - List of the successive homogeneous transform matrices "self.list_g
       - Number of joints of the robot "self.numberJoints"
       - List of the geometric parameters of the tool "self.toolDHM" given
    
    Output:
       - Homogeneous transform matrix "self.g_0E"
    """

    alpha, d, theta, r = self.toolDHM
    self.g_0E = self.list_g_0i[-1] @ self.TransformMatElem(alpha, d, theta

    return self.g_0E

RobotModel.ComputeToolPose = ComputeToolPose
```

In the following, we consider two joint configurations $q = [q_1, \dots, q_6]^t$ of the robot: $q_i = \left[-\frac{\pi}{2}, 0, -\frac{\pi}{2}, -\frac{\pi}{2}, -\frac{\pi}{2}, -\frac{\pi}{2}\right]^t$ and $q_f = \left[0, \frac{\pi}{4}, 0, \frac{\pi}{2}, \frac{\pi}{2}, 0\right]^t$.

Indicate what are the homogeneous transform matrices $\bar{g}_{0E}$ evaluated in these two confugrations.

In [1...
```python
# Will be used in future cells

q_i = [-np.pi/2,
       0,
       -np.pi/2,
       -np.pi/2,
       -np.pi/2,
       -np.pi/2]

q_f = [0,
       np.pi/4,
       0,
       np.pi/2,
       np.pi/2,
       0]
```

In [1...
```
RobotTutorials.ComputeDGM(q_i)
print(f'g_0E(q_i) = \n{RobotTutorials.ComputeToolPose()}')

RobotTutorials.ComputeDGM(q_f)
print(f'g_0E(q_f) = \n{RobotTutorials.ComputeToolPose()}')
```

```
g_0E(q_i) =
[[-6.12323400e-17 -6.12323400e-17 -1.00000000e+00 -1.00000000e-01]
 [ 1.00000000e+00 -1.23259516e-32 -6.12323400e-17 -7.00000000e-01]
 [-1.23259516e-32 -1.00000000e+00  6.12323400e-17  3.00000000e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
g_0E(q_f) =
[[-7.07106781e-01  7.07106781e-01  3.01573569e-33  6.36396103e-01]
 [-4.32978028e-17 -4.32978028e-17 -1.00000000e+00 -1.00000000e-01]
 [-7.07106781e-01 -7.07106781e-01  6.12323400e-17  1.13639610e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

Q4. What are the values of positions P_x, P_y, P_z and the parameters related to the orientation $R_{n,q}$ (n being the direction vector and $q \in [0, \pi]$ the rotation angle such that $R_{n,q} = R_{0E}$) of the end-effector frame for the two joint configurations $q_i = \left[-\frac{\pi}{2}, 0, -\frac{\pi}{2}, -\frac{\pi}{2}, -\frac{\pi}{2}, -\frac{\pi}{2}\right]^t$ and $q_f = \left[0, \frac{\pi}{4}, 0, \frac{\pi}{2}, \frac{\pi}{2}, 0\right]^t$ ($q = [q_1, \ldots, q_6]^t$)? To do so, write a function $DescribeToolFrame(\text{self})$ that computes the position vector and the parameters related to the orientation of the end-effector frame for the current configuration.

In [1...
```
def DescribeToolFrame(self):
        """
        Computation of the position vector and the parameters related to t

        Input:
                - Direct Geometric Model (DGM) of the robot including its

        Outputs:
                - Values of positions P=[Px, Py, Pz]' (in m) of the origin
                - Orientation parameters R_n,q, as follows:
                        - "self.n": being the direction vector
                        - "self.q" in [0,pi] the rotation angle in rad suc
        """

        g_0E = self.g_0E
        self.P = g_0E[0:3, 3]
        self.n = g_0E[0:3, 0]
        self.q = np.arccos((np.trace(g_0E[0:3,0:3]) - 1)/2)

        return self.P, self.n, self.q

RobotModel.DescribeToolFrame = DescribeToolFrame
```

In [1...
```
def printToolFrame(joint, desc):
    print(f"{desc} = {joint}")
    RobotTutorials.ComputeDGM(joint)
    RobotTutorials.ComputeToolPose()
    P, n, q = RobotTutorials.DescribeToolFrame()

    print(f"P = {P}\nn = {n}\nq = {q}")
```

```
printToolFrame(q_i, "Qi")
print("="*75)
printToolFrame(q_f, "Qf")
```

```
Qi = [-1.5707963267948966, 0, -1.5707963267948966, -1.5707963267948966,
-1.5707963267948966, -1.5707963267948966]
P = [-0.1 -0.7  0.3]
n = [-6.12323400e-17  1.00000000e+00 -1.23259516e-32]
q = 2.0943951023931957
===========================================================================
=
Qf = [0, 0.7853981633974483, 0, 1.5707963267948966, 1.5707963267948966, 0]
P = [ 0.6363961 -0.1        1.1363961]
n = [-7.07106781e-01 -4.32978028e-17 -7.07106781e-01]
q = 2.5935642459694805
```

Direct kinematic model

Q5. Write a function $ComputeJac\left(self, q\right)$ which output is the Jacobian matrix ${}^{0}J(q)$ (computed by the method of velocities composition).

Reminder: the Jacobian matrix relates the velocities in the task coordinates of the end-effector frame in $\mathcal{R}_0$, for a given joint configuration q, to the joint velocities:

$$ {}^{0}\mathcal{V}_{0,E} = \begin{bmatrix} {}^{0}V_{0,E}\left(O_E\right) \\ {}^{0}\omega_{0,E} \end{bmatrix} = \begin{bmatrix} {}^{0}J_v\left(q\right) \\ {}^{0}J_\omega\left(q\right) \end{bmatrix} \dot{q} = {}^{0}J\left(q\right)\dot{q} $$

In [1...

```
def ComputeJac(self, q_cur):
    """
    Computation of the Jacobian matrix mapping the joint velocities to the

    Inputs:
        - List defining the types of joints : "self.jointsType"
        - Number of joints of the robot: "self.numberJoints"
        - Current configuration "q_cur"

    Output:
        - Jacobian matrix 0_J in R_0: "self.oJ" as np.array
    """

    self.ComputeDGM(q_cur)
    p_0E, _, _ = self.DescribeToolFrame()

    z_i = np.array([0, 0, 1]).T

    self.oJ = np.zeros((6, self.numberJoints))

    for i in range(self.numberJoints):
        R_0i = self.list_g_0i[i][0:3, 0:3]
        p_0i = self.list_g_0i[i][0:3, 3]

        p_iE = p_0E - p_0i

        if self.jointsType[i] == 'R':
            self.oJ[0:3, i] = np.cross(R_0i.dot(z_i), p_iE)
            self.oJ[3:6, i] = R_0i.dot(z_i)
        elif self.jointsType[i] == 'P':
            self.oJ[0:3, i] = R_0i.dot(z_i)
            self.oJ[3:6, i] = np.array([[0], [0], [0]])

    return self.oJ

RobotModel.ComputeJac = ComputeJac
```

What are the values of the twists at O_E evaluated with $q = q_i$ and $q = q_f$ with the joint velocities $\dot{q} = [0.5, 1.0, -0.5, 0.5, 1.0, -0.5]^t$?

In [1...

```
def compute_and_print_twist(robot, q, q_dot, config_name):
    twist = robot.ComputeJac(q) @ q_dot

    print(f"Jac({config_name})=\n{np.round(robot.oJ,3)}")

    print(f"\nTwist at O_E for {config_name}")
    print(f"V_0E = [{twist[0]:.4f}, {twist[1]:.4f}, {twist[2]:.4f}]^T m/s
    return twist

q_dot = np.array([0.5, 1.0, -0.5, 0.5, 1.0, -0.5])
q_dot_str = ', '.join([f'{v:.1f}' for v in q_dot])

print(f"Using q_dot = [{q_dot_str}]^T:")

twist_qi = compute_and_print_twist(RobotTutorials, q_i, q_dot, "Qi")
twist_qf = compute_and_print_twist(RobotTutorials, q_f, q_dot, "Qf")

#print(twist_qi)
#print(twist_qf)
```

```
Using q_dot = [0.5, 1.0, -0.5, 0.5, 1.0, -0.5]^T:
Jac(Qi)=
[[ 0.1   -0.    -0.     0.6   -0.836 -0.   ]
 [ 0.636  0.636  0.636 -0.636 -0.     0.836]
 [-0.     0.1   -0.6   -0.     0.636 -0.6  ]
 [ 0.    -1.    -1.    -0.    -0.    -1.   ]
 [ 0.    -0.    -0.    -0.    -1.    -0.   ]
 [ 1.     0.     0.    -1.    -0.     0.   ]]

Twist at O_E for Qi
V_0E = [-0.4864, -0.1000, 1.3364]^T m/s | omega_0E = [0.0000, -1.0000,
-0.0000]^T rad/s
Jac(Qf)=
[[ 0.1   -0.636 -0.141  0.071 -0.071  0.   ]
 [ 0.636  0.     0.     0.    -0.     0.   ]
 [-0.     0.636  0.141 -0.071 -0.071 -0.   ]
 [ 0.     0.     0.     0.707  0.707  0.   ]
 [ 0.    -1.    -1.    -0.    -0.    -1.   ]
 [ 1.     0.     0.     0.707 -0.707  0.   ]]

Twist at O_E for Qf
V_0E = [-0.5510, 0.3182, 0.4596]^T m/s | omega_0E = [1.0607, -0.0000, 0.14
64]^T rad/s
```

Q6. In the rest of the study, we restrict the analysis of operational end-effector velocities to translational velocities via ${}^{0}J_v\left(q\right)$.

Qualify the transmission of velocities between the joint and task spaces for the corresponding q_i and q_f configurations:

- what is the preferred direction to transmit velocity in the task space when the manipulator configuration is q_i? Same question for q_f?
- What are the corresponding velocity manipulabilities?

To help, you can program a function $QualifyVelocityTransmission(self)$ that analyses the property of the Jacobian matrix. Explain your results.

```
In [ ]: def QualifyVelocityTransmission(self, q_cur):
            """
            Qualifying the transmission of velocities

            Input:
               - Jacobian matrix "self.oJ" to be analysed
               - Configuration q_to_analyse to compute the Jacobian at
            """

            J_v = self.ComputeJac(q_cur)[0:3, :]

            U, Sigma, Vt = np.linalg.svd(J_v)
            V = Vt.T

            sigma_max = Sigma[0]       # Maximum singular value
            sigma_min = Sigma[-1]      # Minimum singular value
            u_max = U[:, 0]            # Best direction (task space)
            u_min = U[:, -1]           # Worst direction (task space)

            W = np.prod(Sigma)  # Velocity manipulability measure

            # Singularity analysis
            singularity_threshold = 1e-4  # Threshold for singularity detection
            is_singular = W < singularity_threshold

            q_to_analyse_str = '[' + ', '.join([f'{v:.1f}' for v in q_cur]) + ']'
            print(f"=== Velocity Transmission Analysis @ {q_to_analyse_str} ===\n'
            print(f"Singular values: {Sigma}")
            print(f"Velocity manipulability (w): {W:.6f}")

            if is_singular:
                print("Configuration is near singularity!")
                print(f"Manipulability w = {W:.2e} < threshold = {singularity_thre
            else:
                print(f"Configuration is well-conditioned (w > {singularity_thresh

            print("Principal directions in task space (columns of U):")
            for i in range(3):
                print(f"  Direction u{i+1}: [{U[0,i]:7.4f}, {U[1,i]:7.4f}, {U[2,i]

            print(f"\nBest transmission direction: [{u_max[0]:7.4f}, {u_max[1]:7.4
            print(f"Worst transmission direction: [{u_min[0]:7.4f}, {u_min[1]:7.4

            fig = plt.figure(figsize=(12, 10))
            ax = fig.add_subplot(111, projection='3d')

            u_sphere = np.linspace(0, 2 * np.pi, 50)
            v_sphere = np.linspace(0, np.pi, 50)
            x_sphere = np.outer(np.cos(u_sphere), np.sin(v_sphere))
```

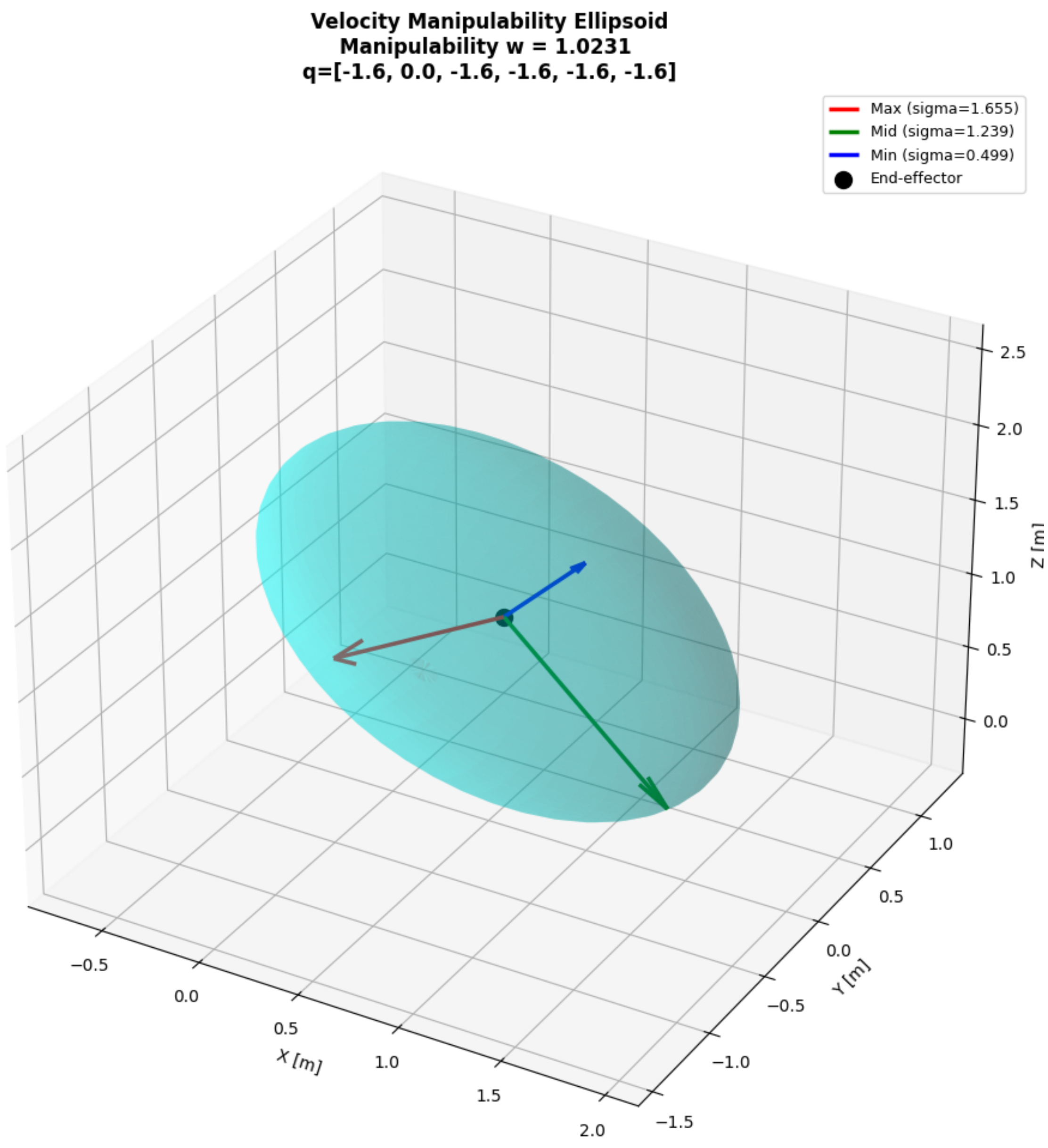
```
y_sphere = np.outer(np.sin(u_sphere), np.sin(v_sphere))
z_sphere = np.outer(np.ones(np.size(u_sphere)), np.cos(v_sphere))

ellipsoid_points = []
for i in range(len(u_sphere)):
    for j in range(len(v_sphere)):
        sphere_pt = np.array([x_sphere[i,j], y_sphere[i,j], z_sphere[i
        ellipsoid_pt = U @ np.diag(Sigma) @ sphere_pt
        ellipsoid_points.append(ellipsoid_pt)

ellipsoid_points = np.array(ellipsoid_points)
x_ellipsoid = ellipsoid_points[:, 0].reshape(len(u_sphere), len(v_sphe
y_ellipsoid = ellipsoid_points[:, 1].reshape(len(u_sphere), len(v_sphe
z_ellipsoid = ellipsoid_points[:, 2].reshape(len(u_sphere), len(v_sphe

P_E = self.g_0E[0:3, 3]

ax.plot_surface(x_ellipsoid + P_E[0], y_ellipsoid + P_E[1], z_ellipso
                alpha=0.3, color='cyan', edgecolor='none')

colors = ['red', 'green', 'blue']
labels = ['Max', 'Mid', 'Min']
for i in range(3):
    direction = U[:, i] * Sigma[i]
    ax.quiver(P_E[0], P_E[1], P_E[2],
              direction[0], direction[1], direction[2],
              color=colors[i], arrow_length_ratio=0.15, linewidth=2.5,
              label=f'{labels[i]} (sigma={Sigma[i]:.3f})')

ax.scatter([P_E[0]], [P_E[1]], [P_E[2]], color='black', s=100,
          label='End-effector', marker='o')

ax.set_xlabel('X [m]', fontsize=10)
ax.set_ylabel('Y [m]', fontsize=10)
ax.set_zlabel('Z [m]', fontsize=10)

title = 'Velocity Manipulability Ellipsoid\n' + f'Manipulability w = 
if is_singular:
    title += ' SINGULARITY'

ax.set_title(title, fontsize=12, fontweight='bold')

max_range = np.array([x_ellipsoid.max()-x_ellipsoid.min(),
                      y_ellipsoid.max()-y_ellipsoid.min(),
                      z_ellipsoid.max()-z_ellipsoid.min()]).max() / 2

mid_x = (x_ellipsoid.max()+x_ellipsoid.min()) * 0.5 + P_E[0]
mid_y = (y_ellipsoid.max()+y_ellipsoid.min()) * 0.5 + P_E[1]
mid_z = (z_ellipsoid.max()+z_ellipsoid.min()) * 0.5 + P_E[2]

ax.set_xlim(mid_x - max_range, mid_x + max_range)
ax.set_ylim(mid_y - max_range, mid_y + max_range)
ax.set_zlim(mid_z - max_range, mid_z + max_range)

ax.legend(loc='upper right', fontsize=9)
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
RobotModel.QualifyVelocityTransmission = QualifyVelocityTransmission
```

To answear the previous question we run the QualifyVelocityTransmission function for each joint configuation:

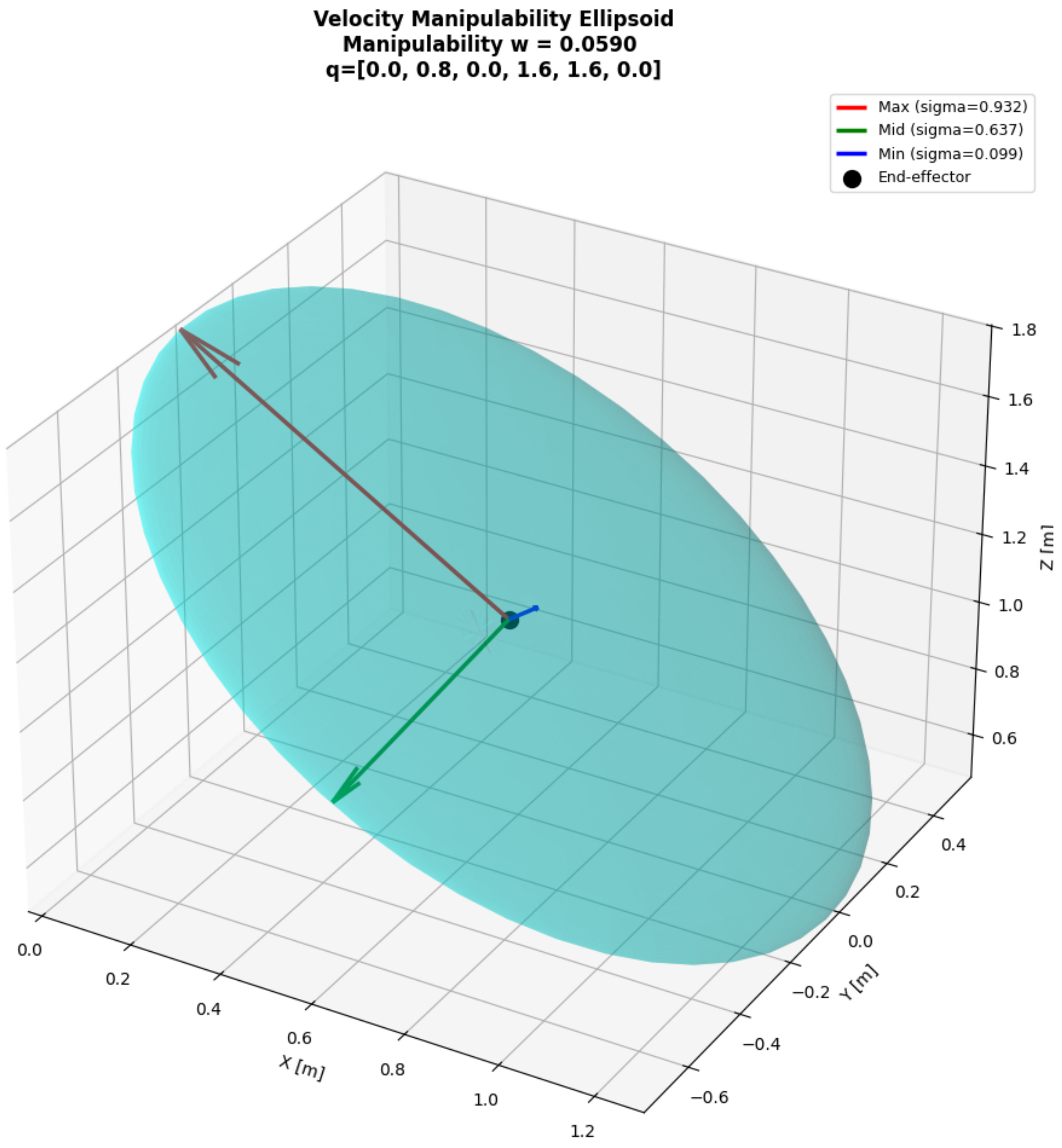
```
In [ ]: RobotTutorials.QualifyVelocityTransmission(q_i)
```

```
=== Velocity Transmission Analysis @ [-1.6, 0.0, -1.6, -1.6, -1.6, -1.6] =
==

Singular values: [1.65528661 1.23908257 0.4988161 ]
Velocity manipulability (w): 1.023090
Configuration is well-conditioned (w > 1.00e-04)

Principal directions in task space (columns of U):
  Direction u1: [ 0.0277, -0.8938,  0.4475]^T (sigma1 = 1.6553)
  Direction u2: [ 0.7978, -0.2500, -0.5487]^T (sigma2 = 1.2391)
  Direction u3: [ 0.6023,  0.3722,  0.7062]^T (sigma3 = 0.4988)

Best transmission direction: [ 0.0277, -0.8938,  0.4475]^T
Worst transmission direction: [ 0.6023,  0.3722,  0.7062]^T
```



```
In [ ]: RobotTutorials.QualifyVelocityTransmission(q_f)
```

```
=== Velocity Transmission Analysis @ [0.0, 0.8, 0.0, 1.6, 1.6, 0.0] ===

Singular values: [0.93244996 0.63691605 0.09937314]
Velocity manipulability (w): 0.059017
Configuration is well-conditioned (w > 1.00e-04)

Principal directions in task space (columns of U):
  Direction u1: [-0.7114, -0.0975,  0.6959]^T (sigma1 = 0.9324)
  Direction u2: [-0.0103, -0.9888, -0.1490]^T (sigma2 = 0.6369)
  Direction u3: [ 0.7027, -0.1132,  0.7025]^T (sigma3 = 0.0994)

Best transmission direction: [-0.7114, -0.0975,  0.6959]^T
Worst transmission direction: [ 0.7027, -0.1132,  0.7025]^T
```



Inverse geometric model

Q7. In this study, the resolution of the inverse geometric model is considered numerically by exploiting the inverse differential model. Moreover, the study is restricted to the position only of the robot's end-effector frame in the task space (no constraint on the orientation of the end-effector frame).

Using an iterative procedure exploiting the pseudo-inverse of the Jacobian matrix, program a function $ComputeIGM(\text{self}, X_d, q_0, k_{max}, \epsilon_x)$ having as input arguments the desired task position X_d and the initial condition q_0. Both the maximum number of iterations k_{max} of the algorithm and the norm of the tolerated Cartesian error $|X_d - DGM(q_k)| < \epsilon_x$, define the stopping criteria of the algorithm.

```
In [ ]: def ComputeIGM(self, X_d, q_0, k_max, eps_x):
           """
           Computation of the Inverse Geometric Model (IGM) mapping the Cartesian

           Inputs:
              - Desired Cartesian vector "X_d" as a np.array to be reached by the
              - Initial condition "q_0" as a np.array
              - Number "k_max" of maximal iteration in the recursive algorithm
              - Norm of the tolerated Cartesian error "eps_x"

           Outputs:
              - List "self.list_q_IGM" of the joint vectors computed at each itera
              - Returned "self.list_q_IGM[-1]" of the final found joint vector, so

           """
           self.list_q_IGM = []
           qk = q_0

           for k in range(k_max):
              self.list_q_IGM.append(qk)
              # update error ###########
              self.ComputeDGM(qk)
              self.ComputeToolPose()
              Xk = self.g_0E[0:3, 3]
              error =  X_d - Xk
              error_norm = np.linalg.norm(error)
              ###########################
              self.ComputeJac(qk)

              if error_norm < eps_x:
                 #print(f"Converged in {k} iterations! Final error = {error_norm*
                 return self.list_q_IGM[-1], k

              qk = qk + np.linalg.pinv(self.oJ[0:3, :]) @ error

              #qk = qk + self.oJ[0:3, :].T @ error

           #print(f"no convergence (error = {error_norm*1000:.1f} mm)")
           return self.list_q_IGM[-1], -1

        RobotModel.ComputeIGM = ComputeIGM
```

Compute q^* when the function is called with the following arguments:

a) $X_d = X_{d_i} = (-0.1, -0.7, 0.3)^t$,
$q_0 = [-1.57, 0.00, -1.47, -1.47, -1.47, -1.47]$, $k_{max} = 100$, $\epsilon_x = 1mm$?

b) $X_d = X_{d_f} = (0.64, -0.10, 1.14)^t$, $q_0 = [0, 0.80, 0.00, 1.00, 2.00, 0.00]$, $k_{max} = 100$, $\epsilon_x = 1mm$?

Check the accuracy of the results using the function calculated in **Q3.**

```
In [ ]: k_max = 100
        eps_x = 0.001

        def calculate_igm(X_d, q_0):
            igm, steps = RobotTutorials.ComputeIGM(X_d, q_0, k_max, eps_x)
            pos = RobotTutorials.g_0E[0:3, 3]
            error = np.linalg.norm(X_d - pos)
            print(f"")
            print(f'q_0 = {q_0}\nq*= {igm}\nX_d: {X_d}\nX*: {pos}\nerror norm {erro


        print("---- IGM tests ----")
        calculate_igm(X_d = np.array([-0.1, -0.7, 0.3]), q_0 = np.array([-1.57, 0
        print("-------------------")
        calculate_igm(X_d = np.array([0.64, -0.1, 1.14]), q_0 = np.array([0, 0.8,
```

```
---- IGM tests ----

q_0 = [-1.57  0.   -1.47 -1.47 -1.47 -1.47]
q*= [-1.57239062  0.01358612 -1.52368211 -1.44494862 -1.48204603 -1.47
]
X_d: [-0.1 -0.7  0.3]
X*: [-0.09993478 -0.6999681   0.30027285]
error norm 0.282mm (in steps=1)
-------------------

q_0 = [0.  0.8 0.  1.  2.  0. ]
q*= [-2.44142745e-02  7.64266396e-01 -1.78033951e-01  1.00185485e+00
  1.57056629e+00  2.94879089e-17]
X_d: [ 0.64 -0.1   1.14]
X*: [ 0.63989594 -0.09989798  1.1399421 ]
error norm 0.157mm (in steps=2)
```

Inverse kinematic model

In this question, the trajectory of the end-effector to be followed in the task space must allow the desired final position X_{d_f} to be reached by following a straight line in the task space starting at the initial position X_{d_i}. This rectilinear motion is carried out at a constant speed $V = 1m.s^{-1}$ and is sampled at a period $T_e = 1ms$. The position of the end effector at the time instant kT_e is noted X_{d_k}. The initial configuration of the robot is given by q_i (found in question **Q4.**).

Q8. Using the inverse differential kinematic model, write a function entitled $ComputeIKM(\text{self}, X_{d_i}, X_{d_f}, V, T_e, q_i)$ realizing the coordinate transform to provide the series of setpoint values q_{d_k} corresponding to the X_{d_k} to the joint drivers. To do this, after having programmed the time law corresponding to the required motion, you can use the function developed in question **Q7** capable of calculating the iterative MGI from the pseudo-inverse of the Jacobian matrix.

In [ ]:
```
def ComputeIKM(self, X_d_i, X_d_f, V, Te, q_i, k_max=10, eps_x=0.001):
    # Discretization of movement
    distance = np.linalg.norm(X_d_f - X_d_i)
    duration = distance / V
    steps = int(duration / Te)

    self.discreteTime = np.array([k * Te for k in range(steps + 1)])
    self.list_X_d_k = [X_d_i + k / steps * (X_d_f - X_d_i) for k in range

    # Iterate over discretization
    q_k = q_i
    self.list_q_d_k = []

    for X_d_k in self.list_X_d_k:
        q_k, steps = self.ComputeIGM(X_d_k, q_k, k_max, eps_x)
        self.list_q_d_k.append(q_k)

RobotModel.ComputeIKM = ComputeIKM
```

Check that the successively reached positions of the end-effector is following the desired trajectory. To do so, you can plot the error between the sequence of positions reached by the end device and the position set points at each time step.

In [ ]:
```
RobotTutorials.ComputeDGM(q_i)
RobotTutorials.ComputeToolPose()
X_di, _ ,_ = RobotTutorials.DescribeToolFrame()

RobotTutorials.ComputeDGM(q_f)
RobotTutorials.ComputeToolPose()
X_df, _ ,_ = RobotTutorials.DescribeToolFrame()

print(f"X_di = {X_di}\nX_df = {X_df}")

RobotTutorials.ComputeIKM(X_di, X_df, 1.0, 0.001, q_i, 1000)
```

```
X_di = [-0.1 -0.7  0.3]
X_df = [ 0.6363961 -0.1        1.1363961]
```

In [ ]:
```
def plot_limits(self, discrete_time, q, q_min, q_max):
    plt.figure(figsize=(8, 16))

    for i in range(self.numberJoints):
        ax = plt.subplot(6, 1, i+1)
        plt.plot(discrete_time, q[i], label=f'q_{i+1}')
        plt.axhline(y=q_min[i], color='r', linestyle='--', label='q_min' :
        plt.axhline(y=q_max[i], color='g', linestyle='--', label='q_max' :

        if i == 5:
            plt.xlabel('time [s]')

        plt.yticks([
            -4*np.pi/4, -3*np.pi/4, -2*np.pi/4, -1*np.pi/4, 0,
            +1*np.pi/4, +2*np.pi/4, +3*np.pi/4, +4*np.pi/4]
        )
        plt.xlim([0, 1.4])
        plt.ylim([-np.pi - 0.1, +np.pi + 0.1])
        plt.ylabel(f'q_{i+1} [rad]')
        plt.legend(loc="upper left")
        plt.grid(True)
        # Make axis limits visible
        ax.spines['top'].set_visible(True)
        ax.spines['right'].set_visible(True)
        ax.spines['bottom'].set_visible(True)
        ax.spines['left'].set_visible(True)
        ax.tick_params(axis='both', which='both', direction='in', top=Tru

    plt.tight_layout()
    plt.show()
```

Q9. Plot the temporal evolution of the joint variables q_1 to q_6 calculated in the previous question. For each joint variable, graphically overlay the allowable extreme values corresponding to the joint limits:

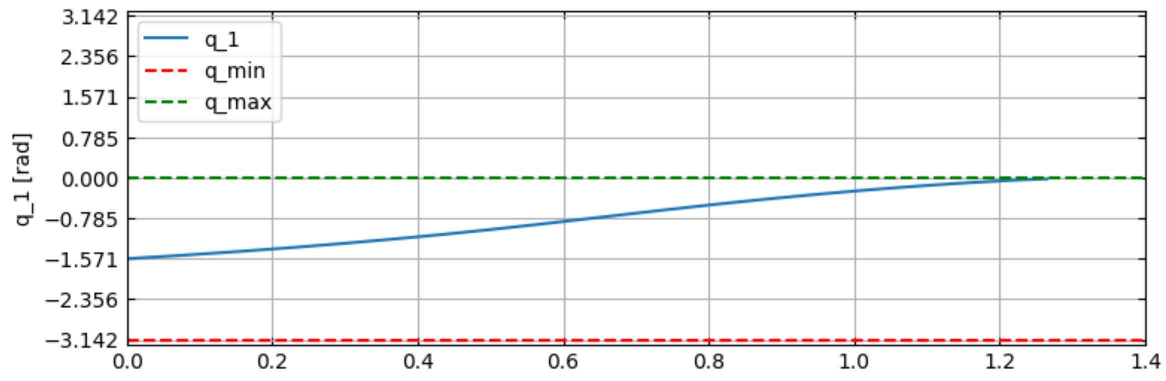
$$q_{min} = \left[-\pi, -\frac{\pi}{2}, -\pi, -\pi, -\frac{\pi}{2}, -\pi\right]$$

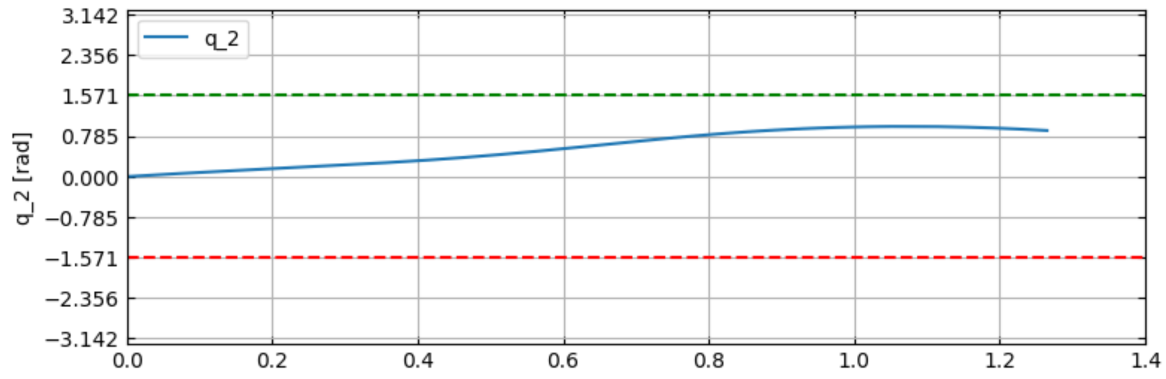
and

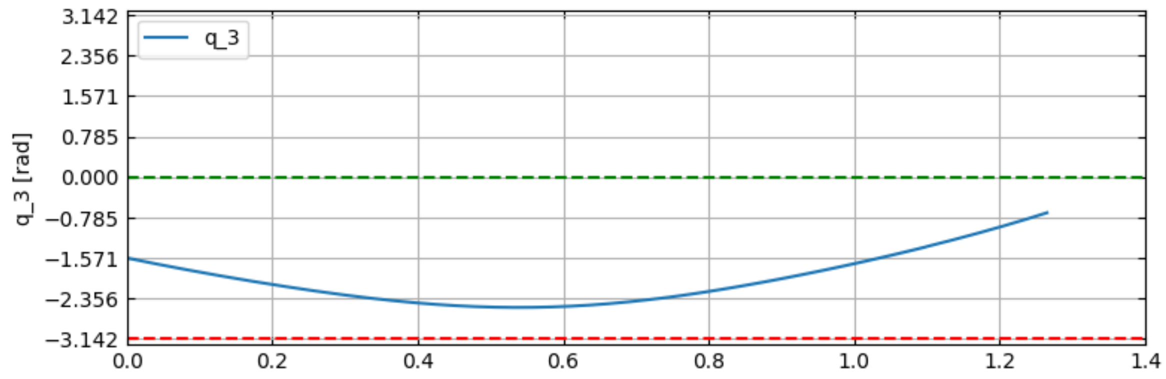
$$q_{max} = \left[0, \frac{\pi}{2}, 0, \frac{\pi}{2}, \frac{\pi}{2}, \frac{\pi}{2}\right]$$

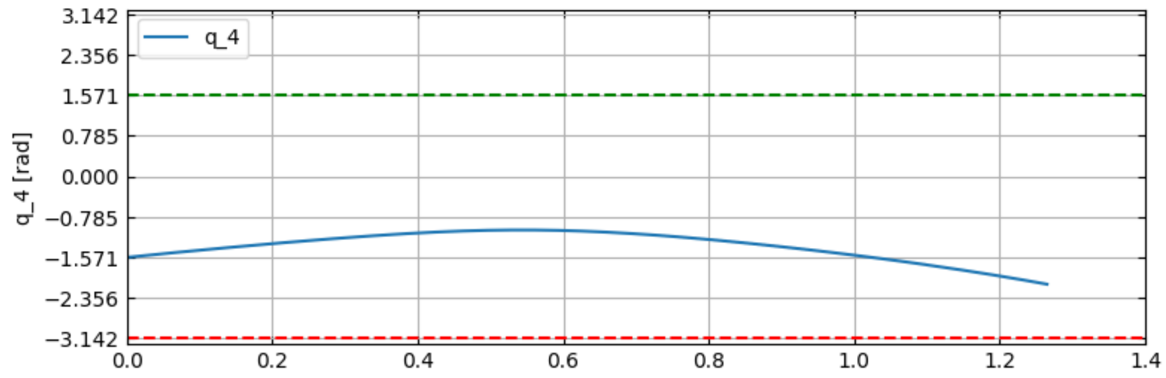
In [ ]:
```
q_min = np.array([-np.pi, -np.pi/2, -np.pi, -np.pi, -np.pi/2, -np.pi])
q_max = np.array([0, np.pi/2, 0, np.pi/2, np.pi/2, np.pi/2])
```

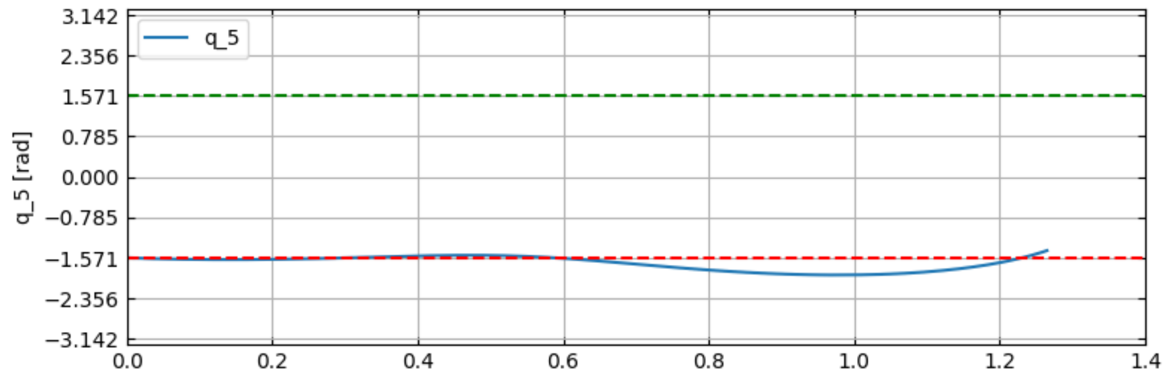
In [ ]:
```
plot_limits(RobotTutorials,
    RobotTutorials.discreteTime,
    np.array(RobotTutorials.list_q_d_k).T,
    q_min,
    q_max,
)
```

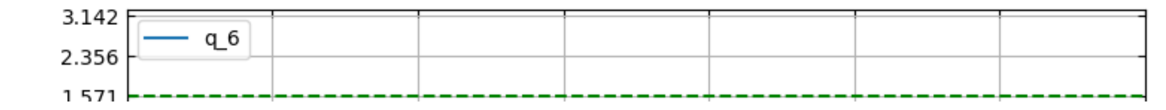

q_1
q_min
q_max
q__1 [rad]

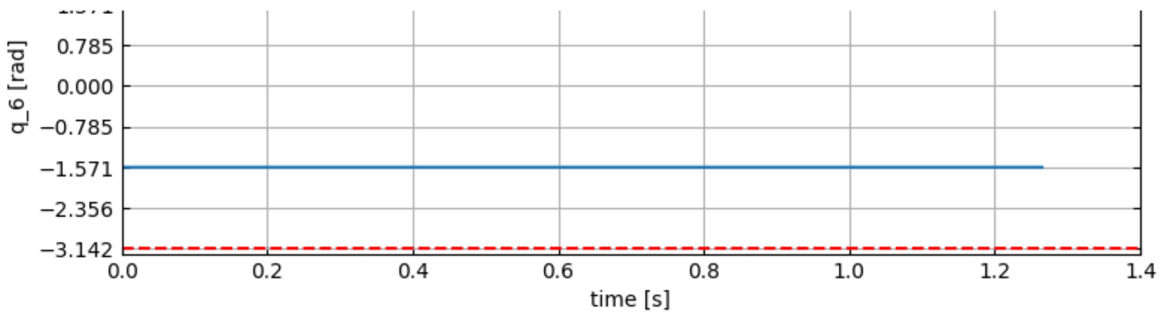

q_2
q_2 [rad]


q_3
q_3 [rad]


q_4
q_4 [rad]


q_5
q_5 [rad]


q_6



Comment on the evolution of the joint variables obtained previously.

Answer **Q9**:

Within the plots above we can see that most joints do evolve in between their desired limits except for "q_5". It is expected and by pute luck it is just one as we have not specifed the limits in any part of the algorithm (which was left for next question (Q10)).

Q10. In this question, we modify the algorithm developed in question **Q8**. We wish to take into account the distance of the values taken by the joint variables from their limits in the computation of the inverse kinematic model.

To do so, you will need to consider a secondary task aiming at keeping some distance from the articular stops q_{min} and q_{max}. By the technique of the gradient projected into the null space of ${}^{0}J_{v}\left(q\right)$, you will consider minimizing the following potential function:

$$H_{lim}\left(q\right)=\sum_{i=1}^{n}\left(\frac{q_i-\bar{q}_i}{q_{max}-q_{min}}\right)^2 \text{ where } \bar{q}_i=\frac{q_{max}+q_{min}}{2}$$

First, provide below the theoretical analytical solution for the joint variables to this problem.

Answer: Theoretical Solution for **Q10**

The potential function $H_{lim}(q)=\sum_{i=1}^{n}\left(\frac{q_i-\bar{q}_i}{q_{max}-q_{min}}\right)^2$ where $\bar{q}_i=\frac{q_{max}+q_{min}}{2}$ acts as a gravitational attractor toward the center of the joint limits. This quadratic function is minimized when all joints are at their central positions and increases as they approach their boundaries. To minimize this potential and track the desired Cartesian trajectory. The negative gradient direction naturally pulls joints toward their center positions.

The primary task ensures the end-effector reaches the desired Cartesian position X_d by applying the pseudo-inverse update $J_v^{\dagger}(X_d-X_k)$. The secondary task minimizes the potential function by projecting the negative gradient $-\alpha\nabla H_{lim}(q)$ into the null space of the Jacobian using the projector $N=I-J_v^{\dagger}J_v$.

The complete iterative update law becomes:

$q_{k+1} = q_k + J_v^\dagger(q_k)\left(X_d - X_k\right) - \alpha\left(I - J_v^\dagger(q_k)J_v(q_k)\right)\nabla H_{lim}(q_k)$, where $\alpha > 0$ is a tunable gain controlling how hard the algorithm avoids the joint limits. The pseudo-inverse $J_v^\dagger = J_v^T(J_vJ_v^T)^{-1}$ can be computed using a damped least-squares formula.

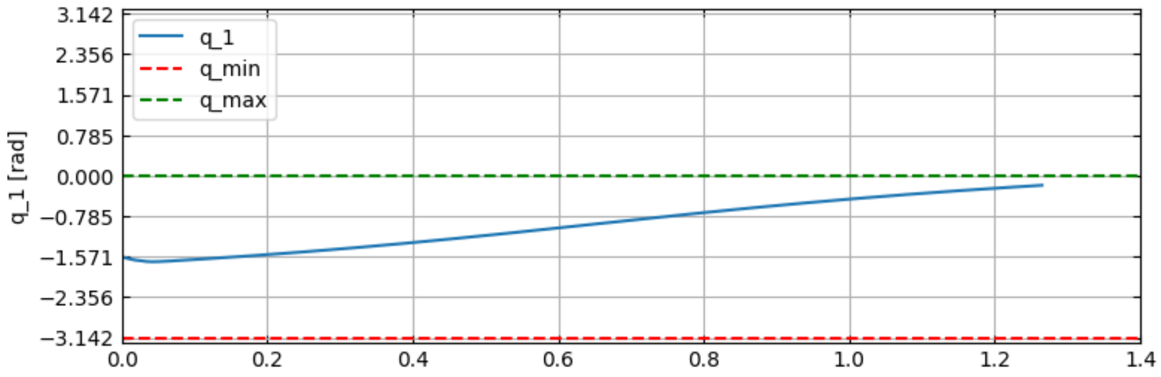
Then, develop a new function $ComputeIKMlimits(\text{self}, X_{d_i}, X_{d_f}, V, T_e, q_i, q_{min}, q_{max})$ which implements the inverse kinematic model able to take into account the previous secondary task.

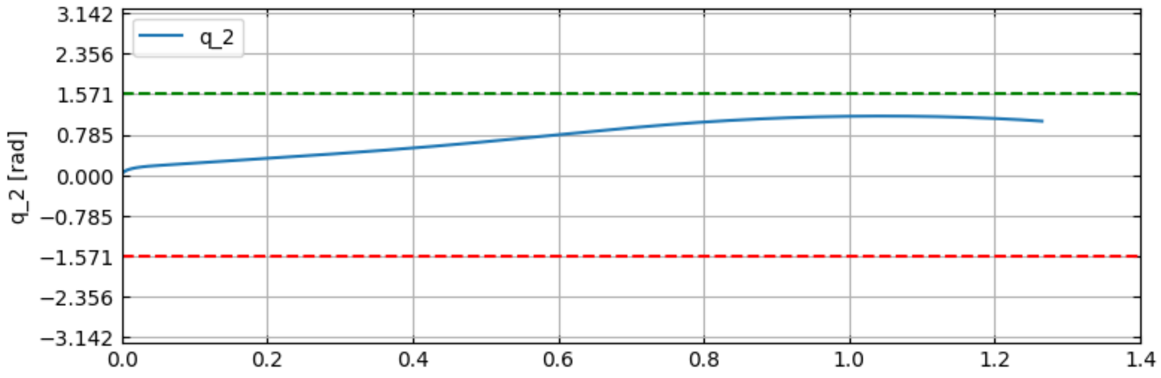
```
In [ ]: def ComputeIKMlimits(self, X_d_i, X_d_f, V, Te, q_i, k_max, eps_x, q_min,
                """
                Computation of the Inverse differential Kinematic Model (IKM) mak

                Inputs:
                   - Trajectory of the end effector to be followed in the task spa
                       - the initial position "X_d_i"
                       - the desired final position "X_d_f" to be reached.
                   - Rectilinear motion carried out :
                       - at a constant speed "V"
                       - sampled at a period "Te"
                   - Initial configuration of the robot "q_i"
                   - Number "k_max" of maximal iteration in the recursive algorit
                   - Norm of the tolerated Cartesian error "eps_x"
                   - Vector of lower bound of joint variable "q_min"
                   - Vector of upper bound of joint variable "q_max"

                Outputs:
                   - List "self.list_q_dk_limits" of the joint vectors computed a
                   - List "self.list_X_d_k" of the intermediate Cartesian poses t
                """

                distance = np.linalg.norm(X_d_f - X_d_i)
                duration = distance / V
                num_steps = int(duration / Te)

                self.discreteTime = np.array([k * Te for k in range(num_steps + 1
                self.list_X_d_k = [X_d_i + k / num_steps * (X_d_f - X_d_i) for k
                self.list_q_d_k_limits = []

                q_mid = (q_max + q_min) / 2
                alpha = 0.5

                q_k = np.array(q_i, dtype=float)

                for X_d_k in self.list_X_d_k:
                    for iteration in range(k_max):
                        self.ComputeDGM(q_k)
                        self.ComputeToolPose()
                        X_k = self.g_0E[0:3, 3]

                        error = X_d_k - X_k
                        error_norm = np.linalg.norm(error)

                        if error_norm < eps_x:
                            break
                        self.ComputeJac(q_k)
                        J_v = self.oJ[0:3, :]

                        grad_H = 2 * (q_k - q_mid) / ((q_max - q_min)**2)

                        J_pinv = J_v.T @ np.linalg.inv(J_v @ J_v.T + 1e-6 * np.eye

                        N = np.eye(len(q_k)) - J_pinv @ J_v

                        q_k = q_k + J_pinv @ error - alpha * N @ grad_H

                    self.list_q_d_k_limits.append(q_k.copy())

        RobotModel.ComputeIKMlimits = ComputeIKMlimits
```

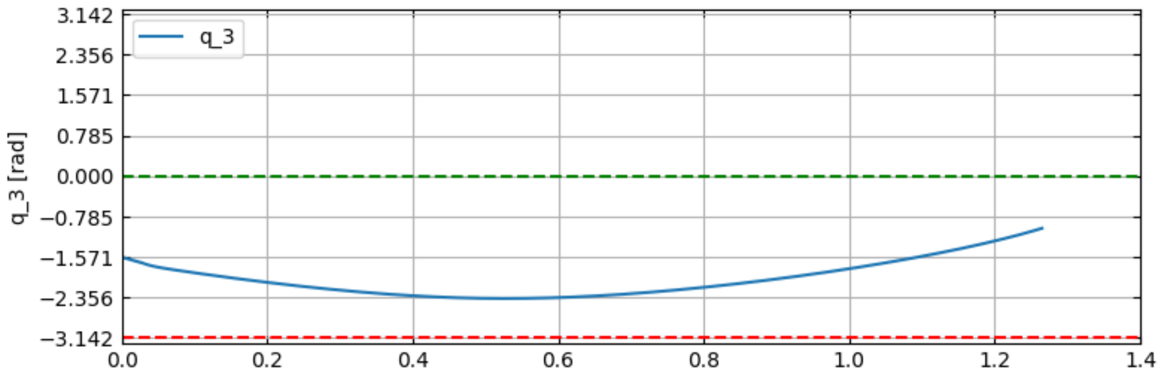
Plot the new temporal evolution of the joint variables q_1 to q_6 for the reference trajectory given in the question **Q9**.

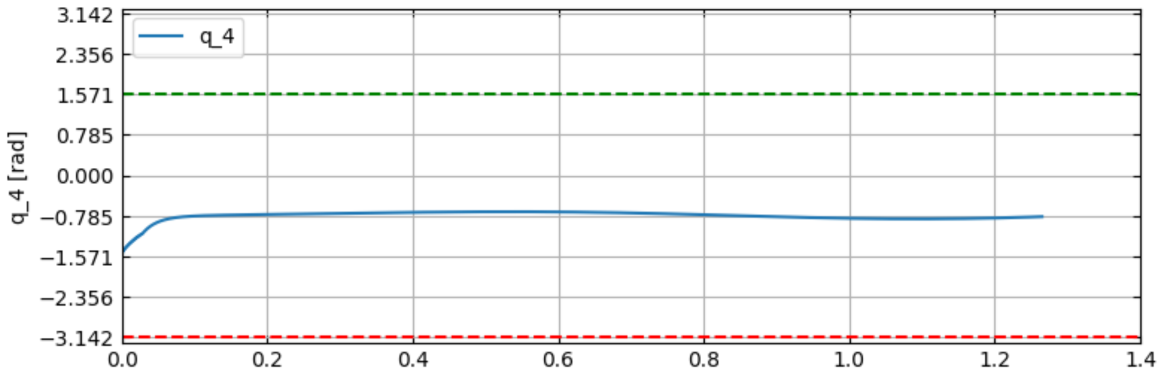
In [ ]:

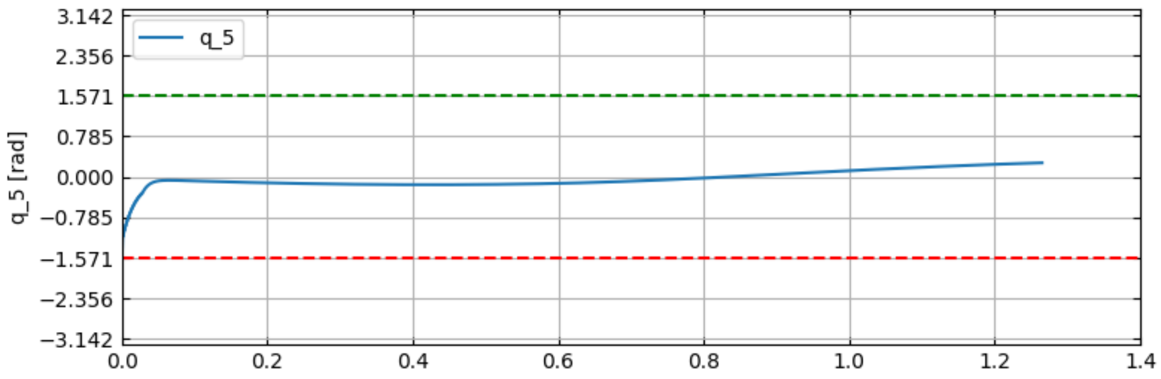
```
RobotTutorials.ComputeIKMlimits(X_di, X_df, 1.0, 0.001, q_i, 1000, 0.001,

plot_limits(
    RobotTutorials,
    RobotTutorials.discreteTime,
    np.array(RobotTutorials.list_q_d_k_limits).T,
    q_min,
    q_max,
)
```

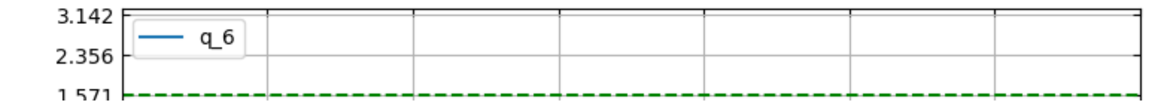

q_1
q_min
q_max
q_1 [rad]
3.142
2.356
1.571
0.785
0.000
−0.785
−1.571
−2.356
−3.142
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

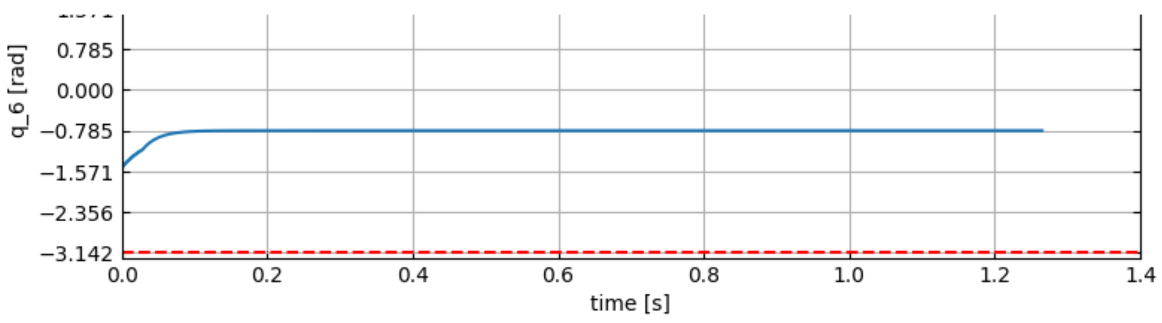

q_2
q_2 [rad]


q_3
q_3 [rad]


q_4
q_4 [rad]


q_5
q_5 [rad]


q_6
3.142
2.356
1.571



Comment on the values taken by the joint variables.

All the 6 joints seems are safelly controlled, avoiding their respective limites letting the end effector move safelly. Even considering noisy implementation, the example with tested make it look like there is room for small "overshoots" as there is a big boundary around almost all joint.