Final Project Report Kinematics Module Diploma in Robotics

Brayan Gerson Duran Toconas

November 27, 2025

Contents

List of Figures

1. Introduction

This report presents the formulation of the forward and inverse kinematics equations for the Yaskawa MH12 robot, using the Product of Exponentials (PoE) method with global reference frames. It also includes the installation sections and the procedures for each kinematic model, along with detailed instructions for running both programs.

Installation Instructions

Download the developed package folder `kinematics_mh12` (included in the same directory as this document) and place it, along with the `yaskawa_robot` package specified in the final project guidelines, inside the `src/` directory of your workspace. The `src/` directory should have the following structure:

Listing 1: Tree of packages

```
tree -L 1 src/
src/
 kinematics_mh12
 yaskawa_robot
```

Next, from the root directory of the workspace, verify the installation of all dependencies using the following command: `rosdep update && rosdep install --from-paths src`

Alternatively, you can install the dependencies manually using `pip`: `pip3 install sympy numpy scipy pyyaml` adding the `--break-system-packages` flag if required by your system. Finally, compile the workspace using: `colcon build && source install/setup.bash`

2. Forward Kinematics

Forward kinematics determines the position of the end-effector as a function of the robot's joint angles. This position and orientation corresponds to the total homogeneous transformation matrix T, also referred to as M in the PoE method.

The following section presents the interpretation of the reference frames and rotations used in the kinematic model.

2.1. References Axis

Figure 1 presents a comprehensive view of the robot's kinematic model, including all the annotations and reference frames necessary for applying the PoE method. This figure serves as a visual guide for understanding the orientation and position of each joint as well as the end-effector, facilitating the implementation of both forward and inverse kinematics calculations.

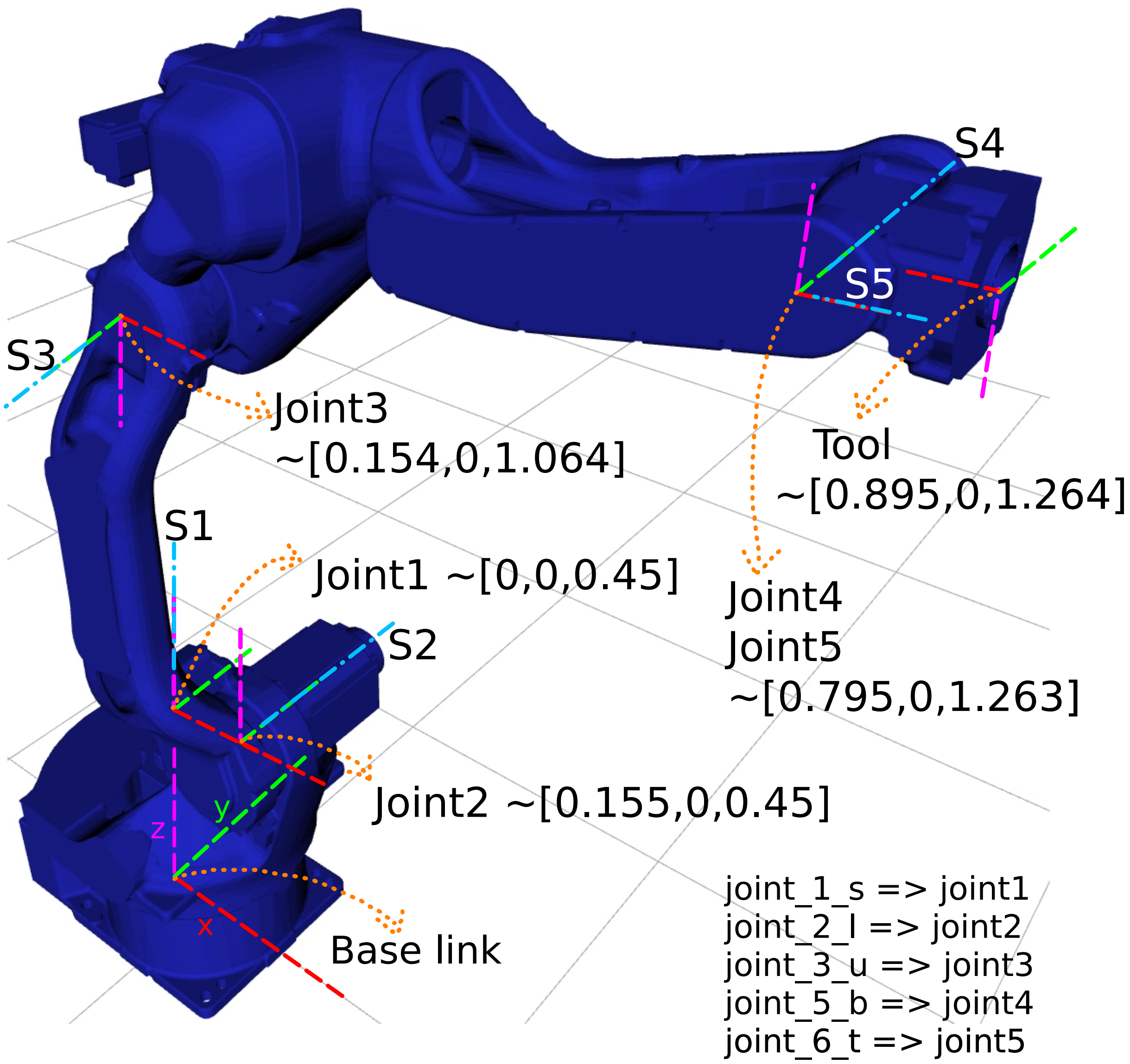


Figure 1: Robot with the reference frames illustrated

2.2. Screw Axes

$$s_1: \quad \mathbf{w} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{q} = \begin{bmatrix} 0 \\ 0 \\ 0.44999998807907104 \end{bmatrix} \tag{1}$$

$$s_2: \quad \mathbf{w} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{q} = \begin{bmatrix} 0.1550000011920929 \\ 0 \\ 0.44999998807907104 \end{bmatrix} \tag{2}$$

$$s_3: \quad \mathbf{w} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, \quad \mathbf{q} = \begin{bmatrix} 0.15487676858901978 \\ 0 \\ 1.0640000104904175 \end{bmatrix} \tag{3}$$

$$s_4: \quad \mathbf{w} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{q} = \begin{bmatrix} 0.7949622273445129 \\ 0 \\ 1.2637263536453247 \end{bmatrix} \tag{4}$$

$$s_5: \quad \mathbf{w} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{q} = \begin{bmatrix} 0.7949622273445129 \\ 0 \\ 1.2637263536453247 \end{bmatrix} \tag{5}$$

$$M = \begin{bmatrix} 0 & 0 & 1 & 0.8949621915817261 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1.263683557510376 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

2.3. Procedure to Execute the Forward Kinematics Program

To run the forward kinematics, execute the following command. After running it, the position and orientation values of the end-effector will start printing in the same terminal.

Listing 2: Command to run forward kinematics

```
$ ros2 launch kinematics_mh12 launch_mode.launch.py mode:=forward
```

The previous command will launch a preconfigured `RViz2` session along with the `Joint State Publisher` graphical interface. Additionally, the position and orientation of the end-effector will be continuously displayed in the same terminal from which the launch was executed.

3. Inverse Kinematics

The inverse kinematics was solved using the total homogeneous transformation matrix T, whose last column ($T_{0,3}$, $T_{1,3}$, $T_{2,3}$) represents the translation components p_x, p_y, and p_z. These elements define the position equations of the end-effector as functions of the robots joint variables (θ_1, θ_2, θ_3, θ_4, θ_5).

From these equations, the Jacobian matrix and its inverse are derived and used in an iterative gradient-based approach to reach the target position by minimizing the error. The joint update control law is defined as:

$$\theta_i = \theta_i + \alpha\, \Delta\theta$$

While minimizing the approximation error, the real-time joint state of the robot is published as follows:

Listing 3: Real-time joint state publishing

```
joint_msg.name = ['joint_1_s', 'joint_2_l', 'joint_3_u', 'joint_4_r',
    'joint_5_b', 'joint_6_t']

joint_angles = [ti[0], ti[1], ti[2], 0.0, -ti[3], ti[4]]
```

In this configuration, `joint_4_r` is fixed to zero since it has no mobility. The `joint_5_b` angle is inverted due to its opposite orientation in the model, while the last joint, `joint_6_t`, maintains rotation because there is a small displacement along the z-axis between joints `j5` and `j4`.

3.1. Procedure to Execute the Inverse Kinematics Program

To execute the inverse kinematics, the same package uses the same launch file with an additional argument to specify the mode. It can be started as follows:

Listing 4: Command to run inverse kinematics

```
$ ros2 launch kinematics_mh12 launch_mode.launch.py mode:=inverse
```

To define the target position, open another terminal within the same `ROS_DOMAIN_ID` and execute the following command:

Listing 5: Command to publish a target point

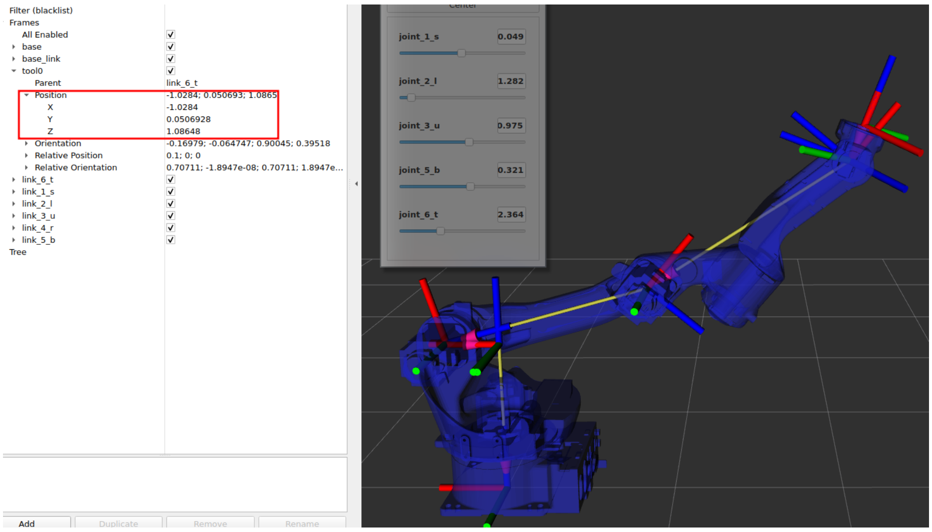
```
$ ros2 topic pub /target geometry_msgs/msg/Point "{x: 0.3, y: 0.3, z: 0.3}" --once
```

If the message is published correctly, the first terminal (where the inverse kinematics was launched) will begin displaying the real-time approximation of the joint values as the error decreases. This process typically takes about 2 to 3 minutes. Once the iteration converges, an information message will indicate that the target has been successfully reached, after which a new target can be sent.

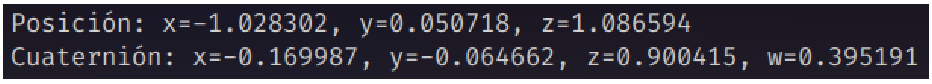
4. Discussion

The analysis is considered valid, as the results obtained from both forward and inverse kinematics demonstrate high accuracy. For forward kinematics, the difference between the estimated values and those visualized in `RViz2` is approximately ± 0.0001, while for inverse kinematics the difference is even smaller, around ± 0.0001.

The following Figure 2a, 2b, 3a, 3b provide visual evidence supporting the correct operation of both methods:

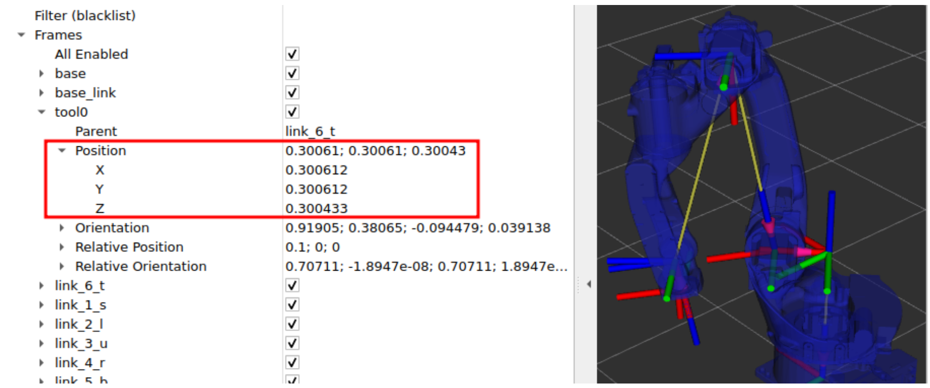


(a) RViz2

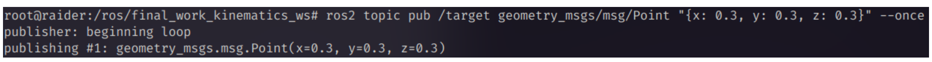


(b) Forward kinematics

Figure 2: Comparison of forward kinematics values



(a) RViz2



(b) Inverse kinematics

Figure 3: Comparison of inverse kinematics values

5. Conclusion

In conclusion, the implementation of the PoE method for determining the robot's kinematics was successful. This is supported by the low errors obtained in both forward and inverse kinematics, as well as by meeting all the deliverable requirements specified in the project guidelines.

A. Codes

Listing 6: Forward code of Python

```
#ROS 2 PoE Node.

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import JointState

import sympy as sp
from sympy import symbols, Matrix, eye, sin, cos, pprint, pi
import os
import yaml
import numpy as np
from scipy.spatial.transform import Rotation as R_quat

# -- Symbols definition
theta = symbols('theta', real=True)
w0, w1, w2 = symbols('omega_0 omega_1 omega_2', real=True)
qx, qy, qz = symbols('q_x q_y q_z', real=True)
L1, L2, L3 = symbols('L_1 L_2 L_3', real=True)
t1, t2, t3, t4, t5 = symbols('theta_1 theta_2 theta_3 theta_4 theta5', real=True)

# Symbolic twist axis
w = Matrix([w0, w1, w2])

# Skew-symmetric matrix
skew_w = Matrix([
    [0, -w[2], w[1]],
    [w[2], 0, -w[0]],
    [-w[1], w[0], 0]
])

# Rodrigues / Exponential map pieces
R = eye(3) + sin(theta)*skew_w + (1 - cos(theta))*(skew_w**2)
v = -skew_w*Matrix([[qx], [qy], [qz]])
Rv = (eye(3)*theta + (1 - cos(theta))*(skew_w) + (theta - sin(theta))*(skew_w**2)) * v

# Homogeneous Transform
T_elem = Matrix([[R[0,0], R[0,1], R[0,2], Rv[0]],
                 [R[1,0], R[1,1], R[1,2], Rv[1]],
                 [R[2,0], R[2,1], R[2,2], Rv[2]],
                 [0, 0, 0, 1]])

# Symbolic quaternion components (tool)
qtx, qty, qtz, qtw = symbols('q_tool_x q_tool_y q_tool_z q_tool_w', real=True)

# Normalize quaternion (to be safe if the numeric inputs are not normalized)
norm_q = sp.sqrt(qtx**2 + qty**2 + qtz**2 + qtw**2)
qx_n = qtx / norm_q
qy_n = qty / norm_q
qz_n = qtz / norm_q
qw_n = qtw / norm_q

# Rotation matrix from normalized quaternion (qw is scalar part)
Rot_tool = Matrix([
```

```python
    [1 - 2*(qy_n**2 + qz_n**2),     2*(qx_n*qy_n - qz_n*qw_n),     2*(qx_n*qz_n + qy_n*qw_n)],
    [2*(qx_n*qy_n + qz_n*qw_n),     1 - 2*(qx_n**2 + qz_n**2),     2*(qy_n*qz_n - qx_n*qw_n)],
    [2*(qx_n*qz_n - qy_n*qw_n),     2*(qy_n*qz_n + qx_n*qw_n),     1 - 2*(qx_n**2 + qy_n**2)]
])

# Definitions of the tool quaternion and position values
# Numeric defaults for the tool quaternion (qx,qy,qz,qw) - unit quaternion
# represents no rotation.
TOOL_QX = -1.89428046581952e-08
TOOL_QY = 0.7072578072547913
TOOL_QZ = 1.8950899516312347e-08
TOOL_QW = 0.7069556713104248

# Build the 4x4 M with the tool rotation and the previously used translation
M_sym = sp.eye(4)
for i in range(3):
    for j in range(3):
        M_sym[i, j] = Rot_tool[i, j]
# translation values from original M
M_sym[0, 3] = 0.8949621915817261
M_sym[1, 3] = 0.0
M_sym[2, 3] = 1.263683557510376

# Definition of M (pose end-effector at  home)
# Substitute numeric tool quaternion so FK depends only on joint angles
M = sp.N(M_sym.subs({qtx: TOOL_QX, qty: TOOL_QY, qtz: TOOL_QZ, qtw: TOOL_QW}))
#pprint(M)

# Transfrorms for each joint and screws (axis and point)
T1 = T_elem.subs({theta: t1, w0: 0, w1: 0, w2: 1, qx: 0, qy: 0, qz: 0.44999998807907104})
T2 = T_elem.subs({theta: t2, w0: 0, w1: 1, w2: 0, qx: 0.1550000011920929, qy: 0, qz: 0.44999998807907104})
T3 = T_elem.subs({theta: t3, w0: 0, w1: -1, w2: 0, qx: 0.15487676858901978, qy: 0, qz: 1.0640000104904175})
T4 = T_elem.subs({theta: t4, w0: 0, w1: 1, w2: 0, qx: 0.7949622273445129, qy: 0, qz: 1.2637263536453247})
T5 = T_elem.subs({theta: t5, w0: -1, w1: 0, w2: 0, qx: 0.7949622273445129, qy: 0, qz: 1.2637263536453247})

# Final Transform for the robot
T_sym = sp.expand(T1 * T2 * T3 * T4 * T5 * M)

# Save symbolic matrix to a YAML file
def save_symbolic_matrix(matrix, filename):
    matrix_data = {
        'shape': matrix.shape,
        'elements': [[str(matrix[i,j]) for j in range(matrix.shape[1])]
                     for i in range(matrix.shape[0])]
    }
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    with open(filename, 'w') as f:
        yaml.dump(matrix_data, f, default_flow_style=False)

matrix_file = os.path.join('data', 'T_sym_matrix.yaml')
os.makedirs('data', exist_ok=True)
```

```python

save_symbolic_matrix(T_sym, matrix_file)

pprint(T_sym)

# Lambdify for numeric evaluation
fk_func = sp.lambdify((t1, t2, t3, t4, t5), T_sym, modules=['numpy'])

class PoEfwMH12Node(Node):
    def __init__(self):
        super().__init__('poe_mh12_node')
        # Default joint values
        self.j1 = 0.0
        self.j2 = 0.0
        self.j3 = 0.0
        self.j4 = 0.0
        self.j5 = 0.0

        # Subscription `joint_states`
        self.sub = self.create_subscription(JointState, 'joint_states', self.cb_joint_states, 10)

        self.get_logger().info('Nodo PoE RRR iniciado. Suscrito a: /joint_states')

    def cb_joint_states(self, msg: JointState):
        #Callback for search JointState names.
        try:
            names = list(msg.name)
            positions = list(msg.position)
        except Exception:
            self.get_logger().warn('Mensaje JointState sin campos name/position válidos')
            return
        def get_by_name(n):
            if n in names:
                idx = names.index(n)
                if idx < len(positions):
                    return float(positions[idx])
            return None
        # correct joint name (URDF uses 'joint_1_s')
        v1s = get_by_name('joint_1_s')
        v2l = get_by_name('joint_2_l')
        v3u = get_by_name('joint_3_u')
        v5b = get_by_name('joint_5_b')
        v6t = get_by_name('joint_6_t')

        # Fallback:
        if v1s is None or v2l is None or v3u is None or v5b is None or v6t is None:
            if len(positions) >= 5:
                if v1s is None:
                    v1s = float(positions[0])
                if v2l is None:
                    v2l = float(positions[1])
                if v3u is None:
                    v3u = float(positions[2])
                if v5b is None:
                    v5b = float(positions[3])
                if v6t is None:
                    v6t = float(positions[4])
```

```
            else:
                self.get_logger().warn('JointState no contiene suficientes posiciones y no se encontraron nombres esperados')
                return
        self.j1 = v1s
        self.j2_= v2l
        self.j3 = v3u
        self.j4 = v5b
        self.j5 = v6t
        self._recompute_and_print()

    def _recompute_and_print(self):
        # Call to fk_func for the numeric evaluation

        j1 = getattr(self, 'j1', 0.0)
        j2 = getattr(self, 'j2_',0.0)
        j3 = getattr(self, 'j3', 0.0)
        j4 = getattr(self, 'j4', 0.0)
        j5 = getattr(self, 'j5', 0.0)
        try:
            T_num = fk_func(j1, j2, j3, j4, j5)
        except Exception as e:
            self.get_logger().error(f'Error al evaluar fk_func: {e}')
            return

        # numpy 4x4
        T_np = np.array(T_num, dtype=float)
        pos = T_np[0:3, 3]

        # Rotations 3x3 -> cuaternión (x, y, z, w)
        rot_mat = T_np[0:3, 0:3]
        try:
            rotation = R_quat.from_matrix(rot_mat)
            quat = rotation.as_quat()
        except Exception as e:
            self.get_logger().error(f'Error al convertir matriz a cuaternión: {e}')
            return

        #Position: [x, y, z]
        self.get_logger().info(f'Posición: x={pos[0]:.6f}, y={pos[1]:.6f}, z={pos[2]:.6f}')
        # Quaternion: [x, y, z, w]
        self.get_logger().info(f'Cuaternión: x={quat[0]:.6f}, y={quat[1]:.6f}, z={quat[2]:.6f}, w={quat[3]:.6f}')

def main(args=None):
    rclpy.init(args=args)
    node = PoEfwMH12Node()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()
if __name__ == '__main__':
    main()
```

Listing 7: Inverse code of Python

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Header
from sensor_msgs.msg import JointState
from geometry_msgs.msg import Point

from sympy import Matrix, symbols, cos, sin, pi, diff, lambdify, pprint
import numpy as np
import math
from random import random
from kinematics_mh12.fw_kinematics_mh12 import T_sym

#-Old symbols definitions
#t1=symbols('t1')
#t2=symbols('t2')
#t3=symbols('t3')
#t4=symbols('t4')
#t5=symbols('t5')

# New symbols definitions
t1, t2, t3, t4, t5 = symbols('theta_1 theta_2 theta_3 theta_4 theta5', real=True)

# Learning parameters
alpha=0.06  #learning rate 0.07
iterations = 255  #350

def create_jacobian(T_sym, thetas):
    # Get position elements from the transformation matrix T_sym
    px = T_sym[0,3]
    py = T_sym[1,3]
    pz = T_sym[2,3]

    # Create a dinamic Jacobian matrix
    J_rows = []
    # For each position component
    for p in [px, py, pz]:
        # Parcial derivatives w.r.t. each theta
        row = [diff(p, theta) for theta in thetas]
        J_rows.append(row)
    return Matrix(J_rows)

# Get the list of joint variables
thetas = [t1, t2, t3, t4, t5]

# Create jacobian symbolically
px = T_sym[0,3]
py = T_sym[1,3]
pz = T_sym[2,3]
J = create_jacobian(T_sym, thetas)

class invKinematicsMH12Node(Node):
    def __init__(self):
        super().__init__('inverse_mh12_node')
        self.subscription = self.create_subscription(Point, 'target', self.sub_callback,10)
        self.publisher_ = self.create_publisher(JointState, 'joint_states', 10)
        self.subscription
        self.get_logger().info('Node inverse started')
```

```python

    def sub_callback(self,msg):
        driffz=0.0004
        driffxy=0.0003
        target=Matrix([msg.x,msg.y,msg.z])
        self.get_logger().info('Received target: x={:.3f}, y={:.3f}, z={:.3f}'.format(msg.x, msg.y, msg.z))
        ##target += np.sign(target) * driffxy

        for i in [0, 1]:
            target[i] += np.sign(target[i]) * driffxy
            '''
            if target[i]>0:
                target[i] += driffxy
            else:
                target[i] -= driffxy
        '''
        target[2] += np.sign(target[2]) * driffz

        ti=Matrix([random(),random(),random(),random(),random()])
        if hasattr(self, 'last_ti'):
            ti = self.last_ti
        else:
            ti = Matrix([0, 0, 0, 0, 0])

        for i in range(iterations):
            # Evaluate forward position and Jacobian numerically
            try:

                cp = Matrix([
                px.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])]),
                py.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])]),
                pz.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])])])
                Jsubs = J.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])])

                e = (target - cp)

                #e = e / max(1.0, e.norm())

                Jinv=Jsubs.H*(Jsubs*Jsubs.H)**-1
                #Jinv = (Jsubs.T * Jsubs + 0.01 * Matrix.eye(5))**-1 * Jsubs.T

                dt=Jinv*e

                ti=ti+alpha*dt

            except Exception as exc:
                self.get_logger().error(f'Numeric evaluation failed at iter {i}: {exc}')
                break
            # Build and publish final JointState (6 joints expected by other parts; joint_4_r fixed to 0)
            header = Header()
            header.stamp = self.get_clock().now().to_msg()
            joint_msg = JointState()

            joint_msg.name = ['joint_1_s', 'joint_2_l', 'joint_3_u', 'joint_4_r', 'joint_5_b', 'joint_6_t']
```

```
                joint_angles = [ti[0], ti[1], ti[2], 0.0, -ti[3], ti[4]]

                joint_msg.position = joint_angles
                joint_msg.header = header
                self.publisher_.publish(joint_msg)
                self.get_logger().info('Published joint angles: {}'.format(joint_angles))
                self.get_logger().info('Error: {:.6f}'.format(e.norm()))
            self.get_logger().info('Finished IK iterations for target.')
            self.last_ti = ti

def main(args=None):
    rclpy.init(args=args)
    node = invKinematicsMH12Node()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()
if __name__ == '__main__':
    main()
```