

# **Task 1 Perception**

## **Planning and Perception Module**

**Diploma in Robotics**

Brayan Gerson Duran Toconas

November 27, 2025

## **Contents**

<b>1. Introduction</b>	<b>2</b>
<b>2. Forward Kinematics</b>	<b>4</b>
2.1. References Axis . . . . .	4
2.2. Screw Axes . . . . .	5
2.3. Procedure to Execute the Forward Kinematics Program . . . . .	5
<b>3. Inverse Kinematics</b>	<b>5</b>
3.1. Procedure to Execute the Inverse Kinematics Program . . . . .	6
<b>4. Discussion and Results</b>	<b>6</b>
<b>5. Conclusion</b>	<b>7</b>
<b>A. Codes</b>	<b>8</b>

## **List of Figures**

1. Robot with the reference frames illustrated . . . . .	4
2. Comparison of forward kinematics values . . . . .	7
3. Comparison of inverse kinematics values . . . . .	7

## 1. Introduction

This report presents the imageprocessing pipeline used to detect and track the color markers of an autonomous soccer robot. The procedure includes Gaussian-based color thresholding, masking operations, circle detection, and the application of a Kalman filter to obtain a stable and reliable estimation of the robot's position.

## Procedure

Only the most relevant and significant parts of the code will be described, focusing on their functionality.

Listing 1: Read the video

```
1 cap=cv2.VideoCapture(args.video_path)
```

Listing 2: Select the colour to calculate mean and covariance

```
1 def select_color_model(frame):
2     r = cv2.selectROI("Seleccione la pelota", frame, fromCenter=False, showCrosshair=True)
3     x, y, w, h = r
4     roi = frame[y:y+h, x:x+w]
5
6     hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
7     pixels = hsv_roi.reshape((-1, 3))
8
9     mean = np.mean(pixels, axis=0)
10    cov = np.cov(pixels, rowvar=False)
11
12    print("Media HSV:", mean)
13    print("Covarianza HSV:", cov)
14
15    cv2.destroyAllWindows("Seleccione la pelota")
16
17    return mean, cov
18
19 print("Mark the colors...")
20 for c in range(len(colors)):
21     print(f"Color {c+1}: {colors[c]}")
22
23     for i in range(argscaptures):
24         print(f"Captura {i+1} de {argscaptures}")
25         frame_number=int(input(f"Frame number of
{int(cap.get(cv2.CAP_PROP_FRAME_COUNT))}: "))
26         try:
27             cap.set(cv2.CAP_PROP_POS_FRAMES, frame_number)
28             ret, frame=cap.read()
29         except:
30             print("Error: Not valid frame number")
31             continue
32
33         if not ret:
34             print("Error: No se pudo leer el frame")
35             continue
36         men_i,cov_i=ball_tracker.select_color_model(frame)
```

```

37     means[c].append(men_i)
38     covs[c].append(cov_i)

```

Listing 3: Training the gaussian mask from mean and covariance

```

1 mean_hsv[c] = np.mean(means[c], axis=0)
2 cov_hsv[c] = np.mean(covs[c], axis=0)
3 hsv=cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
4 combined_mask = np.zeros(hsv.shape[:2], dtype=np.uint8)
5 for c in range(len(colors)):
6     masks[c] = ball_tracker.gaussian_mask(hsv, mean_hsv[c], cov_hsv[c], args.threshold)
7     masks[c]=cv2.erode(masks[c],kernel,iterations=1)
8     masks[c]=cv2.dilate(masks[c],kernel,iterations=2)
9     combined_mask = cv2.bitwise_or(combined_mask, masks[c])
10    frame_masked=cv2.bitwise_and(frame,frame,mask=combined_mask)

```

Listing 4: Individual color contour detections: all green contours are shown, along with one representative contour for blue and red.

```

1 for i, (name, color) in enumerate(draw_info):
2     contours, _ = cv2.findContours(masks[i], cv2.RETR_EXTERNAL,
3                                     cv2.CHAIN_APPROX_SIMPLE)
4     if contours:
5         if name != "Green":
6             c = max(contours, key=cv2.contourArea) # max
7             ((x, y), radius) = cv2.minEnclosingCircle(c)
8             print(radius)
9             if radius > 9.0 and radius < 15.5: # 0.005
10                detected[name] = True
11                measured_x[name] = int(x)
12                measured_y[name] = int(y)
13                # Dibujo en pantalla
14                cv2.circle(frame, (measured_x[name], measured_y[name]),
15                           int(radius), color, 2)
16                cv2.putText(frame, name, (measured_x[name] + 10,
17                           measured_y[name]), cv2.FONT_HERSHEY_SIMPLEX, 0.4, color, 1)
18            else:
19                for c in contours:
20                    ((x, y), radius) = cv2.minEnclosingCircle(c)
21                    if radius > 8.0 and radius < 15.2:
22                        detected[name] = True
23                        if measured_x[name] is None:
24                            measured_x[name] = []
25                            measured_y[name] = []
26                            measured_x[name].append(int(x))
27                            measured_y[name].append(int(y))
28                            cv2.circle(frame, (int(x), int(y)), int(radius), color, 2)
29                            cv2.putText(frame, name, (int(x) + 10, int(y)),
30                                       cv2.FONT_HERSHEY_SIMPLEX, 0.4, color, 1)

```

Listing 5: Prediction and marking of the color and robot positions using a Kalman filter.

```

1 kalman_positions = {}
2     for name, tracker in trackers.items():
3         if name in ["Red", "Blue"]:
4             pred_x, pred_y = tracker.predict()
5             if detected[name]:
6                 mx = measured_x[name]

```

```

7         my = measured_y[name]
8         tracker.correct(mx, my)
9         current_pos = (mx, my)
10        else:
11            current_pos = (pred_x, pred_y)
12            cv2.putText(frame, f"{name} P(Ocuido)", (int(pred_x) + 10,
13                         int(pred_y)),
14                         cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
15            kalman_positions[name] = current_pos
16            # Dibujo del marcador del Kalman
17            cv2.drawMarker(frame, (int(current_pos[0]), int(current_pos[1])), (0, 0,
18                         255), cv2.MARKER_CROSS, 20, 2)
19            elif name == "Green":
20                if detected[name]:
21                    mx_list = measured_x[name]
22                    my_list = measured_y[name]
23                    if isinstance(mx_list, list):
24                        for x, y in zip(mx_list, my_list):
25                            cv2.drawMarker(frame, (x, y), (0, 0, 255),
26                                          cv2.MARKER_TILTED_CROSS, 15, 2)

```

Listing 6: 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

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

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

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

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

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

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

## 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 7: 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 ( $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$ ).

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 8: Real-time joint state publishing**

```

1 joint_msg.name = ['joint_1_s', 'joint_2_l', 'joint_3_u', 'joint_4_r',
2   'joint_5_b', 'joint_6_t']
3 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 9: 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 10: 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 and Results

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  $\pm 0.0001$ , while for inverse kinematics the difference is even smaller, around  $\pm 0.0001$ .

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 11: Forward code of Python

```

1 #ROS 2 PoE Node.
2
3 import rclpy
4 from rclpy.node import Node
5 from sensor_msgs.msg import JointState
6
7 import sympy as sp
8 from sympy import symbols, Matrix, eye, sin, cos, pprint, pi
9 import os
10 import yaml
11 import numpy as np
12 from scipy.spatial.transform import Rotation as R_quat
13
14 # -- Symbols definition
15 theta = symbols('theta', real=True)
16 w0, w1, w2 = symbols('omega_0 omega_1 omega_2', real=True)
17 qx, qy, qz = symbols('q_x q_y q_z', real=True)
18 L1, L2, L3 = symbols('L_1 L_2 L_3', real=True)
19 t1, t2, t3, t4, t5 = symbols('theta_1 theta_2 theta_3 theta_4 theta5', real=True)
20
21 # Symbolic twist axis
22 w = Matrix([w0, w1, w2])
23
24 # Skew-symmetric matrix
25 skew_w = Matrix([
26     [0, -w[2], w[1]],
27     [w[2], 0, -w[0]],
28     [-w[1], w[0], 0]
29 ])
30
31 # Rodrigues / Exponential map pieces
32 R = eye(3) + sin(theta)*skew_w + (1 - cos(theta))*(skew_w**2)
33 v = -skew_w*Matrix([qx, qy, qz])
34 Rv = (eye(3)*theta + (1 - cos(theta))*(skew_w) + (theta - sin(theta))*(skew_w**2)) * v
35
36 # Homogeneous Transform
37 T_elem = Matrix([[R[0,0], R[0,1], R[0,2], Rv[0]],
38                  [R[1,0], R[1,1], R[1,2], Rv[1]],
39                  [R[2,0], R[2,1], R[2,2], Rv[2]],
40                  [0, 0, 0, 1]])
41
42 # Symbolic quaternion components (tool)
43 qtx, qty, qtz, qtw = symbols('q_tool_x q_tool_y q_tool_z q_tool_w', real=True)
44
45 # Normalize quaternion (to be safe if the numeric inputs are not normalized)
46 norm_q = sp.sqrt(qtx**2 + qty**2 + qtz**2 + qtw**2)
47 qx_n = qtx / norm_q
48 qy_n = qty / norm_q
49 qz_n = qtz / norm_q
50 qw_n = qtw / norm_q
51
52 # Rotation matrix from normalized quaternion (qw is scalar part)
53 Rot_tool = Matrix([

```

```

54     [1 - 2*(qy_n**2 + qz_n**2),      2*(qx_n*qy_n - qz_n*qw_n),      2*(qx_n*qz_n +
55      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 -
56      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 +
57      qy_n**2)]                   ])
58
59 # Definitions of the tool quaternion and position values
60 # Numeric defaults for the tool quaternion (qx,qy,qz,qw) - unit quaternion
61 # represents no rotation.
62 TOOL_QX = -1.89428046581952e-08
63 TOOL_QY = 0.7072578072547913
64 TOOL_QZ = 1.8950899516312347e-08
65 TOOL_QW = 0.7069556713104248
66
67 # Build the 4x4 M with the tool rotation and the previously used translation
68 M_sym = sp.eye(4)
69 for i in range(3):
70     for j in range(3):
71         M_sym[i, j] = Rot_tool[i, j]
72 # translation values from original M
73 M_sym[0, 3] = 0.8949621915817261
74 M_sym[1, 3] = 0.0
75 M_sym[2, 3] = 1.263683557510376
76
77 # Definition of M (pose end-effector at home)
78 # Substitute numeric tool quaternion so FK depends only on joint angles
79 M = sp.N(M_sym.subs({qtx: TOOL_QX, qty: TOOL_QY, qtz: TOOL_QZ, qtw: TOOL_QW}))
80 #pprint(M)
81
82 # Transfrorms for each joint and screws (axis and point)
83 T1 = T_elem.subs({theta: t1, w0: 0, w1: 0, w2: 1, qx: 0, qy: 0, qz: 0.44999998807907104})
84 T2 = T_elem.subs({theta: t2, w0: 0, w1: 1, w2: 0, qx: 0.1550000011920929, qy: 0, qz:
85     0.44999998807907104})
86 T3 = T_elem.subs({theta: t3, w0: 0, w1: -1, w2: 0, qx: 0.15487676858901978, qy: 0, qz:
87     1.0640000104904175})
88 T4 = T_elem.subs({theta: t4, w0: 0, w1: 1, w2: 0, qx: 0.7949622273445129, qy: 0, qz:
89     1.2637263536453247})
90 T5 = T_elem.subs({theta: t5, w0: -1, w1: 0, w2: 0, qx: 0.7949622273445129, qy: 0, qz:
91     1.2637263536453247})
92
93 # Final Transform for the robot
94 T_sym = sp.expand(T1 * T2 * T3 * T4 * T5 * M)
95
96 # Save symbolic matrix to a YAML file
97 def save_symbolic_matrix(matrix, filename):
98     matrix_data = {
99         'shape': matrix.shape,
100        'elements': [[str(matrix[i,j]) for j in range(matrix.shape[1])] for i in range(matrix.shape[0])]}
101
102    os.makedirs(os.path.dirname(filename), exist_ok=True)
103    with open(filename, 'w') as f:
104        yaml.dump(matrix_data, f, default_flow_style=False)
105
106    matrix_file = os.path.join('data', 'T_sym_matrix.yaml')
107    os.makedirs('data', exist_ok=True)

```

```
105     save_symbolic_matrix(T_sym, matrix_file)
106
107     pprint(T_sym)
108
109     # Lambdify for numeric evaluation
110     fk_func = sp.lambdify((t1, t2, t3, t4, t5), T_sym, modules=['numpy'])
111
112
113     class PoEfwMH12Node(Node):
114         def __init__(self):
115             super().__init__('poe_mh12_node')
116             # Default joint values
117             self.j1 = 0.0
118             self.j2 = 0.0
119             self.j3 = 0.0
120             self.j4 = 0.0
121             self.j5 = 0.0
122
123             # Subscription `joint_states`
124             self.sub = self.create_subscription(JointState, 'joint_states',
125             self.cb_joint_states, 10)
126
127             self.get_logger().info('Nodo PoE RRR iniciado. Suscrito a: /joint_states')
128
129         def cb_joint_states(self, msg: JointState):
130             #Callback for search JointState names.
131             try:
132                 names = list(msg.name)
133                 positions = list(msg.position)
134             except Exception:
135                 self.get_logger().warn('Mensaje JointState sin campos name/position válidos')
136             return
137         def get_by_name(n):
138             if n in names:
139                 idx = names.index(n)
140                 if idx < len(positions):
141                     return float(positions[idx])
142             return None
143             # correct joint name (URDF uses 'joint_1_s')
144             v1s = get_by_name('joint_1_s')
145             v2l = get_by_name('joint_2_l')
146             v3u = get_by_name('joint_3_u')
147             v5b = get_by_name('joint_5_b')
148             v6t = get_by_name('joint_6_t')
149
150             # Fallback:
151             if v1s is None or v2l is None or v3u is None or v5b is None or v6t is None:
152                 if len(positions) >= 5:
153                     if v1s is None:
154                         v1s = float(positions[0])
155                     if v2l is None:
156                         v2l = float(positions[1])
157                     if v3u is None:
158                         v3u = float(positions[2])
159                     if v5b is None:
160                         v5b = float(positions[3])
161                     if v6t is None:
162                         v6t = float(positions[4])
```

```
162     else:
163         self.get_logger().warn('JointState no contiene suficientes posiciones y no
164         se encontraron nombres esperados')
165         return
166     self.j1 = v1s
167     self.j2_= v2l
168     self.j3 = v3u
169     self.j4 = v5b
170     self.j5 = v6t
171     self._recompute_and_print()
172
173     def _recompute_and_print(self):
174         # Call to fk_func for the numeric evaluation
175
175         j1 = getattr(self, 'j1', 0.0)
176         j2 = getattr(self, 'j2_', 0.0)
177         j3 = getattr(self, 'j3', 0.0)
178         j4 = getattr(self, 'j4', 0.0)
179         j5 = getattr(self, 'j5', 0.0)
180         try:
181             T_num = fk_func(j1, j2, j3, j4, j5)
182         except Exception as e:
183             self.get_logger().error(f'Error al evaluar fk_func: {e}')
184             return
185
186         # numpy 4x4
187         T_np = np.array(T_num, dtype=float)
188         pos = T_np[0:3, 3]
189
190         # Rotations 3x3 -> cuaternión (x, y, z, w)
191         rot_mat = T_np[0:3, 0:3]
192         try:
193             rotation = R_quat.from_matrix(rot_mat)
194             quat = rotation.as_quat()
195         except Exception as e:
196             self.get_logger().error(f'Error al convertir matriz a cuaternión: {e}')
197             return
198
199         #Position: [x, y, z]
200         self.get_logger().info(f'Posición: x={pos[0]:.6f}, y={pos[1]:.6f}, z={pos[2]:.6f}')
201         # Quaternion: [x, y, z, w]
202         self.get_logger().info(f'Cuaternión: x={quat[0]:.6f}, y={quat[1]:.6f},
203         z={quat[2]:.6f}, w={quat[3]:.6f}')
204
205     def main(args=None):
206         rclpy.init(args=args)
207         node = PoEfwMH12Node()
208         try:
209             rclpy.spin(node)
210         except KeyboardInterrupt:
211             pass
212         finally:
213             node.destroy_node()
214             rclpy.shutdown()
215
216     if __name__ == '__main__':
217         main()
```

Listing 12: Inverse code of Python

```

1 import rclpy
2 from rclpy.node import Node
3 from std_msgs.msg import Header
4 from sensor_msgs.msg import JointState
5 from geometry_msgs.msg import Point
6
7 from sympy import Matrix, symbols, cos, sin, pi, diff, lambdify, pprint
8 import numpy as np
9 import math
10 from random import random
11 from kinematics_mh12.fw_kinematics_mh12 import T_sym
12
13 #-Old symbols definitions
14 #t1=symbols('t1')
15 #t2=symbols('t2')
16 #t3=symbols('t3')
17 #t4=symbols('t4')
18 #t5=symbols('t5')
19
20 # New symbols definitions
21 t1, t2, t3, t4, t5 = symbols('theta_1 theta_2 theta_3 theta_4 theta5', real=True)
22
23 # Learning parameters
24 alpha=0.06 #learning rate 0.07
25 iterations = 255 #350
26
27 def create_jacobian(T_sym, thetas):
28     # Get position elements from the transformation matrix T_sym
29     px = T_sym[0,3]
30     py = T_sym[1,3]
31     pz = T_sym[2,3]
32
33     # Create a dinamic Jacobian matrix
34     J_rows = []
35     # For each position component
36     for p in [px, py, pz]:
37         # Parcial derivatives w.r.t. each theta
38         row = [diff(p, theta) for theta in thetas]
39         J_rows.append(row)
40     return Matrix(J_rows)
41
42 # Get the list of joint variables
43 thetas = [t1, t2, t3, t4, t5]
44
45 # Create jacobian symbolically
46 px = T_sym[0,3]
47 py = T_sym[1,3]
48 pz = T_sym[2,3]
49 J = create_jacobian(T_sym, thetas)
50
51 class invKinematicsMH12Node(Node):
52     def __init__(self):
53         super().__init__('inverse_mh12_node')
54         self.subscription = self.create_subscription(Point, 'target', self.sub_callback, 10)
55         self.publisher_ = self.create_publisher(JointState, 'joint_states', 10)
56         self.subscription
57         self.get_logger().info('Node inverse started')

```

```

58
59     def sub_callback(self,msg):
60         driffz=0.0004
61         driffxy=0.0003
62         target=Matrix([msg.x,msg.y,msg.z])
63         self.get_logger().info('Received target: x={:.3f}, y={:.3f},
64 z={:.3f}'.format(msg.x, msg.y, msg.z))
65         ##target += np.sign(target) * driffxy
66
67         for i in [0, 1]:
68             target[i] += np.sign(target[i]) * driffxy
69             '''
70             if target[i]>0:
71                 target[i] += driffxy
72             else:
73                 target[i] -= driffxy
74             '''
75
76         target[2] += np.sign(target[2]) * driffz
77
78         ti=Matrix([random(),random(),random(),random(),random()])
79         if hasattr(self, 'last_ti'):
80             ti = self.last_ti
81         else:
82             ti = Matrix([0, 0, 0, 0, 0])
83
84         for i in range(iterations):
85             # Evaluate forward position and Jacobian numerically
86             try:
87
88                 cp = Matrix([
89                     px.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])]),
90                     py.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])]),
91                     pz.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])])])
92
93                 Jsubs = J.subs([(t1, ti[0]), (t2, ti[1]), (t3, ti[2]), (t4, ti[3]), (t5, ti[4])])
94
95                 e = (target - cp)
96
97                 #e = e / max(1.0, e.norm())
98
99                 Jinv=Jsubs.H*(Jsubs*Jsubs.H)**-1
100                #Jinv = (Jsubs.T * Jsubs + 0.01 * Matrix.eye(5))**-1 * Jsubs.T
101
102                 dt=Jinv*e
103
104                 ti=ti+alpha*dt
105
106             except Exception as exc:
107                 self.get_logger().error(f'Numeric evaluation failed at iter {i}: {exc}')
108                 break
109
110             # Build and publish final JointState (6 joints expected by other parts;
111             joint_4_r fixed to 0)
112             header = Header()
113             header.stamp = self.get_clock().now().to_msg()
114             joint_msg = JointState()
115
116             joint_msg.name = ['joint_1_s', 'joint_2_l', 'joint_3_u', 'joint_4_r',
117 'joint_5_b', 'joint_6_t']

```

```
112     joint_angles = [ti[0], ti[1], ti[2], 0.0, -ti[3], ti[4]]  
113  
114     joint_msg.position = joint_angles  
115     joint_msg.header = header  
116     self.publisher_.publish(joint_msg)  
117     self.get_logger().info('Published joint angles: {}'.format(joint_angles))  
118     self.get_logger().info('Error: {:.6f}'.format(e.norm()))  
119     self.get_logger().info('Finished IK iterations for target.')  
120     self.last_ti = ti  
121  
122  
123 def main(args=None):  
124     rclpy.init(args=args)  
125     node = invKinematicsMH12Node()  
126     try:  
127         rclpy.spin(node)  
128     except KeyboardInterrupt:  
129         pass  
130     finally:  
131         node.destroy_node()  
132         rclpy.shutdown()  
133 if __name__ == '__main__':  
134     main()  
135
```