Group 5 - Final Presentation

Group Number：5

Group members：宋梓冬 邵钲然 徐景骐 吴俊阳

Course：SI190C

Date：2025.7.16

Mechanical Assembly Section

Basic Assembly

Objective

Complete the fundamental physical assembly of the robotic system and motor serial connection

Process

1. To achieve sufficient torque for joint actuation, we first assembled 6 reducers (1 harmonic reducer and 5 cycloidal reducers)
2. Organized the provided 3D-printed components as required, integrating the pre-assembled motors and reducers
3. Connected the motors in series using terminal wires

Testing

Connected motors to a computer and used Zhang Datou's visualization interface for command debugging; all joints demonstrated normal rotation

Result

Obtained a basic 6-DOF robotic arm without end-effector integration

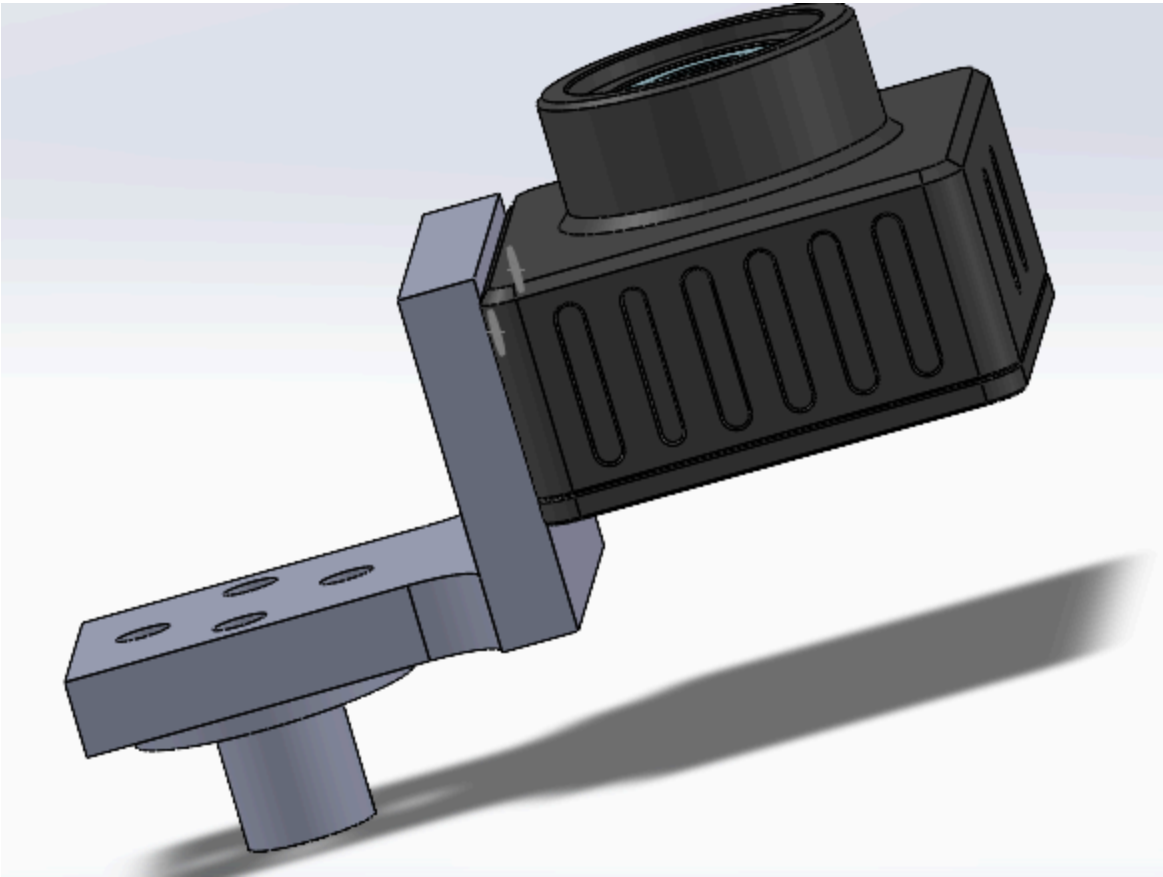
Modeling Section (Major workload component, constrained by subsequent calibration)

Objective

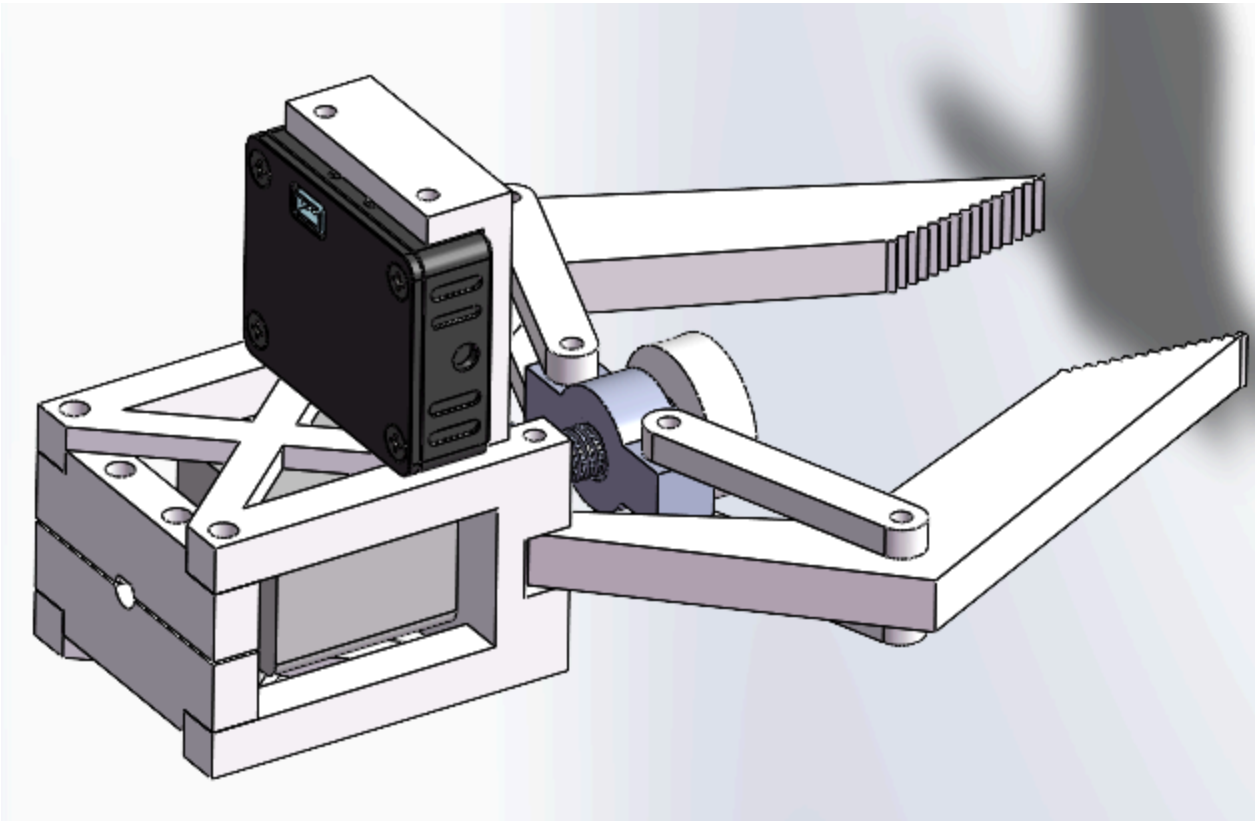
Design the robotic end-effector assembly and complete the modeling, 3D printing, and integration process

Detailed Process

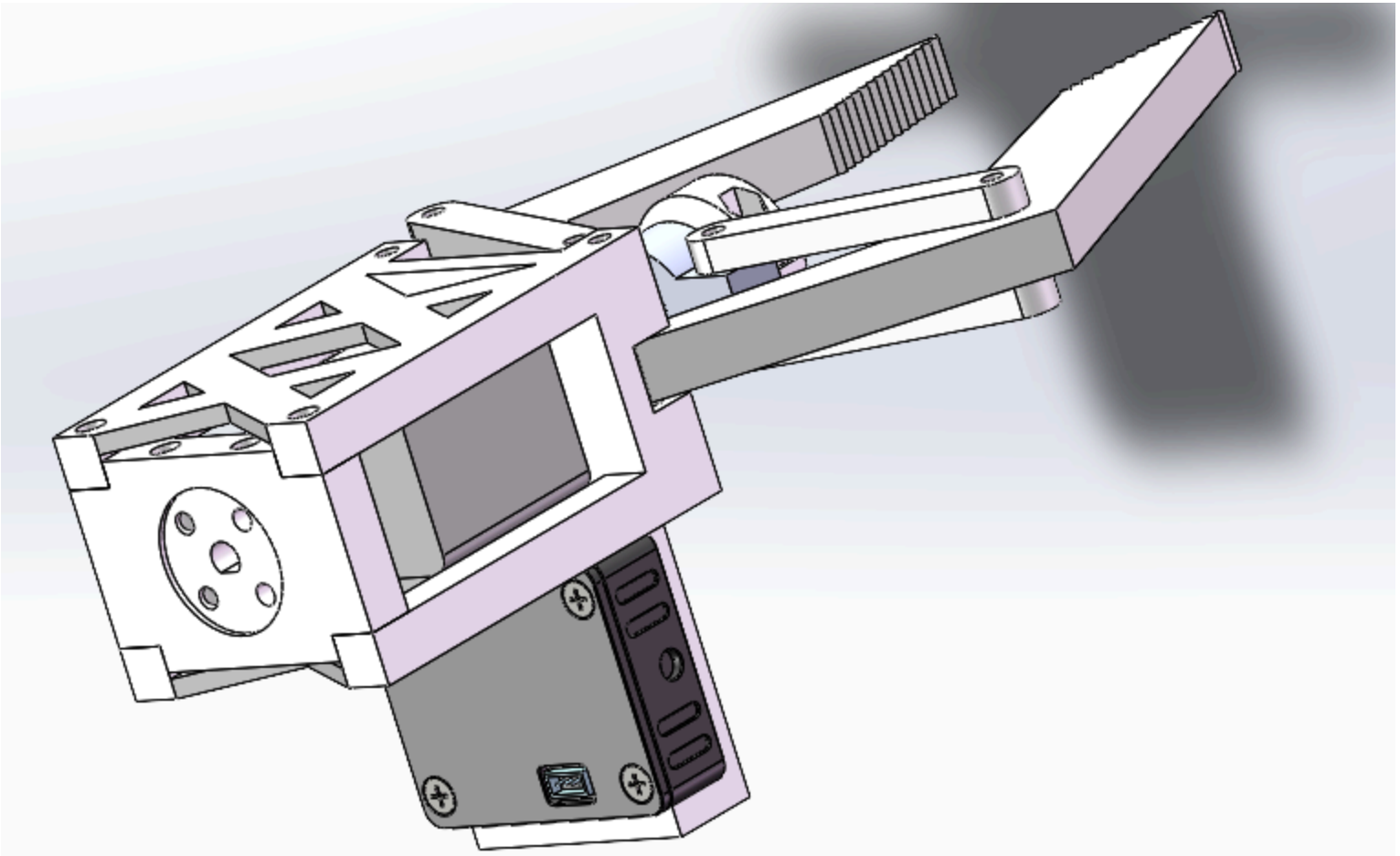
1. The initial end-effector version only accounted for camera-flange connection, omitting gripper integration. This version was fully designed and modeled by team members but abandoned after initial hand-eye calibration



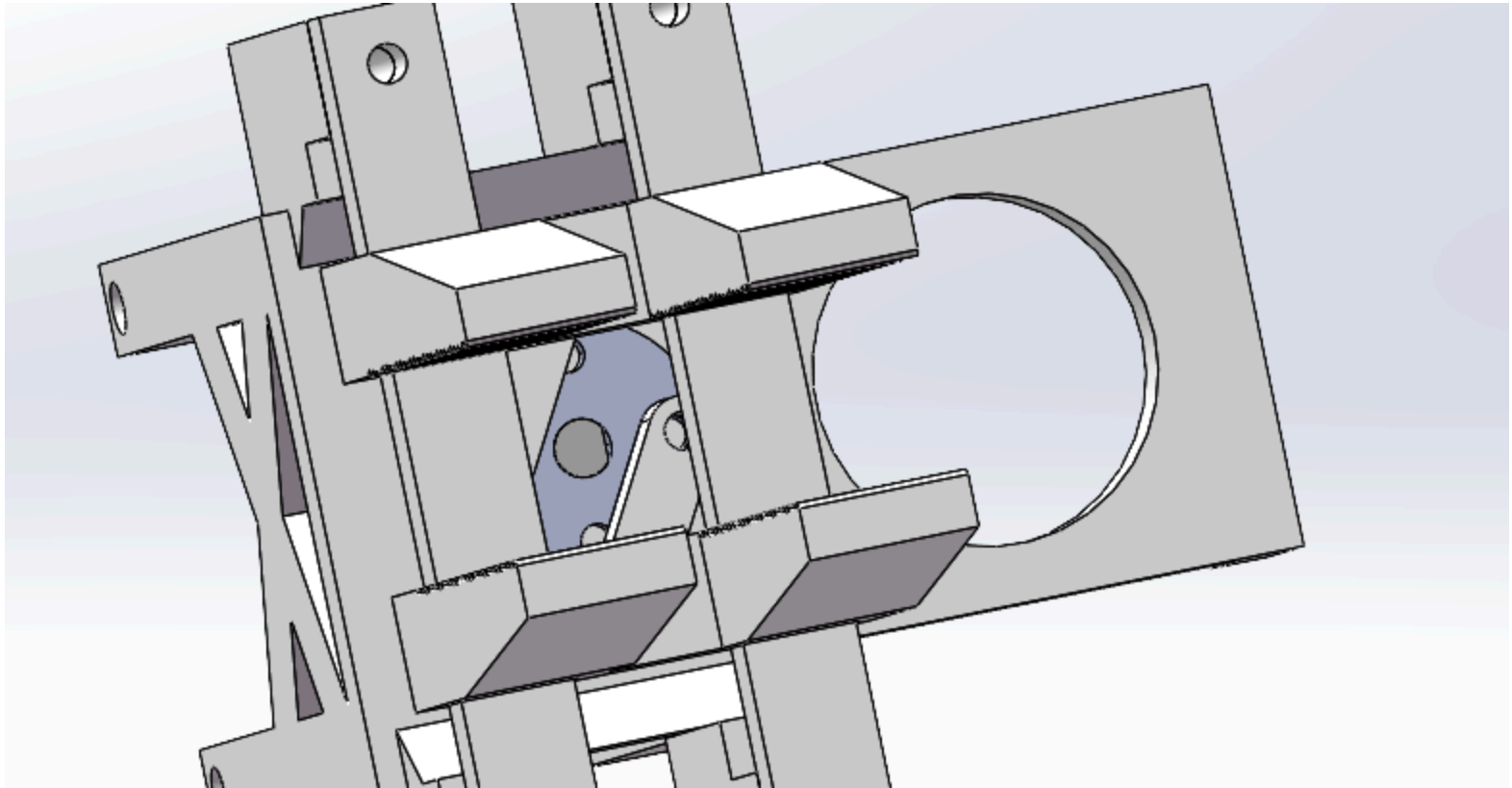
2. The second version incorporated gripper design. After technical research, we implemented a leadscrew transmission mechanism integrated with the camera mount. Design flaws included: insufficient motor terminal interface clearance preventing proper installation, and discontinued flange connection resulting in unstable end connections



3. The third version iterated on the second design, maintaining the leadscrew mechanism while optimizing flange connections and expanding motor housing space. However, the leadscrew's precision requirements exceeded 3D printing tolerances, causing excessive friction and transmission failure



4. The fourth (final) version abandoned previous architectures, adopting a simpler linkage mechanism inspired by industrial robotic grippers. While linkages offer reduced precision and payload capacity compared to leadscrews, their superior trajectory control advantages were deemed more critical. The final gripper successfully accomplished light payload grasping operations.



Possible Improvements

Final Version Defect Analysis

- There are still gaps in the 3D printed parts, resulting in insufficient precision
- The gearbox and linkage structure have not been assembled, so the gripper cannot handle heavy loads
- The motor often experiences stall protection

Extension

- The gripper motor can be treated as joint 7, connected in series with the original six motors. By updating the DH parameters and recalibrating, and rewriting the forward and inverse kinematics equations, the gripper's opening and closing can be seamlessly integrated into the kinematic model, enabling more precise grasping task control. This approach aligns better with our expectations for the gripper.

Challenges Encountered:

- Workspace constraints: The opening and closing of the gripper may change the effective workspace of the end-effector, so reachability must be checked in the inverse kinematics.
- Collision detection: During the opening and closing process, dynamic collision detection between the gripper and the environment is required.

Solutions

- Prioritize solving for the manipulator joints: Ignore the gripper's opening/closing at first and solve the inverse kinematics for the arm.
- Verify gripper reachability: Based on the inverse kinematics solution, check whether the gripper's opening/closing will cause interference or exceed limits. If infeasible, adjust the target pose or opening degree and recalculate.

Further Exploration of Gripper Designs

We hope the gripper can perform more functions in a wider range of scenarios. The current simple gripper design is clearly insufficient for this goal. Due to hardware limitations, we have only conducted further research into various gripper designs.

Adaptive Gripper

- A mechanical gripper that can automatically adjust its gripping method according to the shape, size, or material of the target object, without the need for pre-programming or external intervention. Such grippers perform excellently in uncertain environments (e.g., cluttered object sorting, grasping flexible items).
- *Types to consider*:
  Linkage-based Adaptive Gripper
  Principle: Uses multi-link mechanisms to transmit external force to all contact points, achieving adaptability.
  Reference: Robotiq 2F-85/140—parallel two-finger gripper with built-in springs and linkages, automatically conforming to the object's contour during grasping.
  Advantages: Simple structure, high reliability, no need for sensors.


ROBOTIQ

Intelligent Perception Gripper

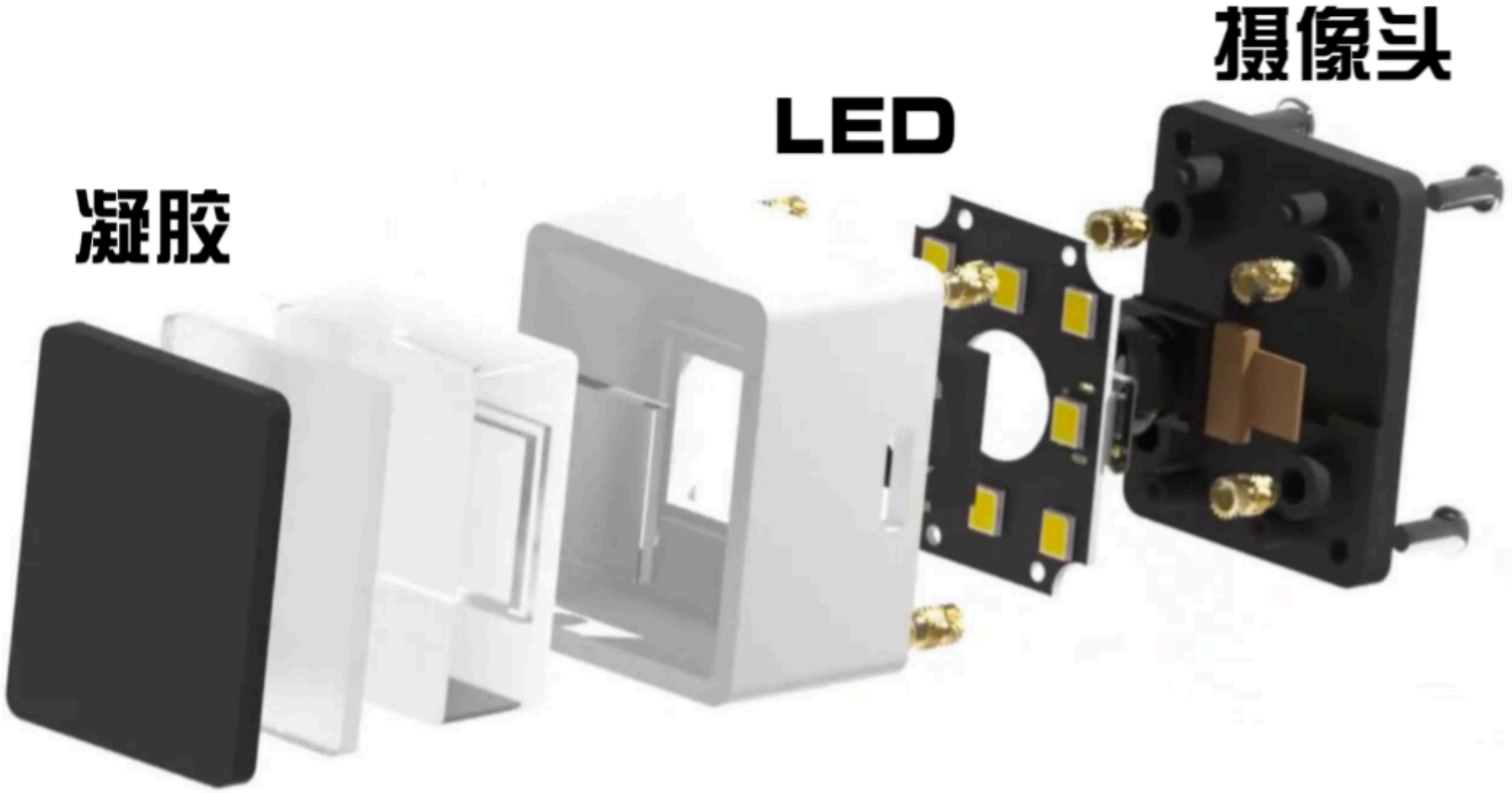
- Integrates multiple sensors and intelligent algorithms to detect the physical properties of the grasped object and the environment in real time, enabling high-precision, adaptive, and safe operation.

- Advantages: Precise force control to prevent overpressure (e.g., in precision electronics assembly) or slippage (e.g., grasping slippery objects), and can handle fragile items. Intelligent decision-making—machine learning can automatically learn optimal grasping postures, reducing manual programming. Human-robot collaboration safety—force sensors can instantly stop movement to avoid harming human operators.
- Disadvantages: Complex system—visual, force, and tactile data must be synchronized and calibrated, making debugging difficult. Lower real-time performance—complex algorithms may reduce response speed, affecting high-speed grasping. Higher power consumption and larger size—multiple sensors and real-time computation require continuous power, limiting battery-powered scenarios.

- *Available sensors*:

  Tactile Sensor

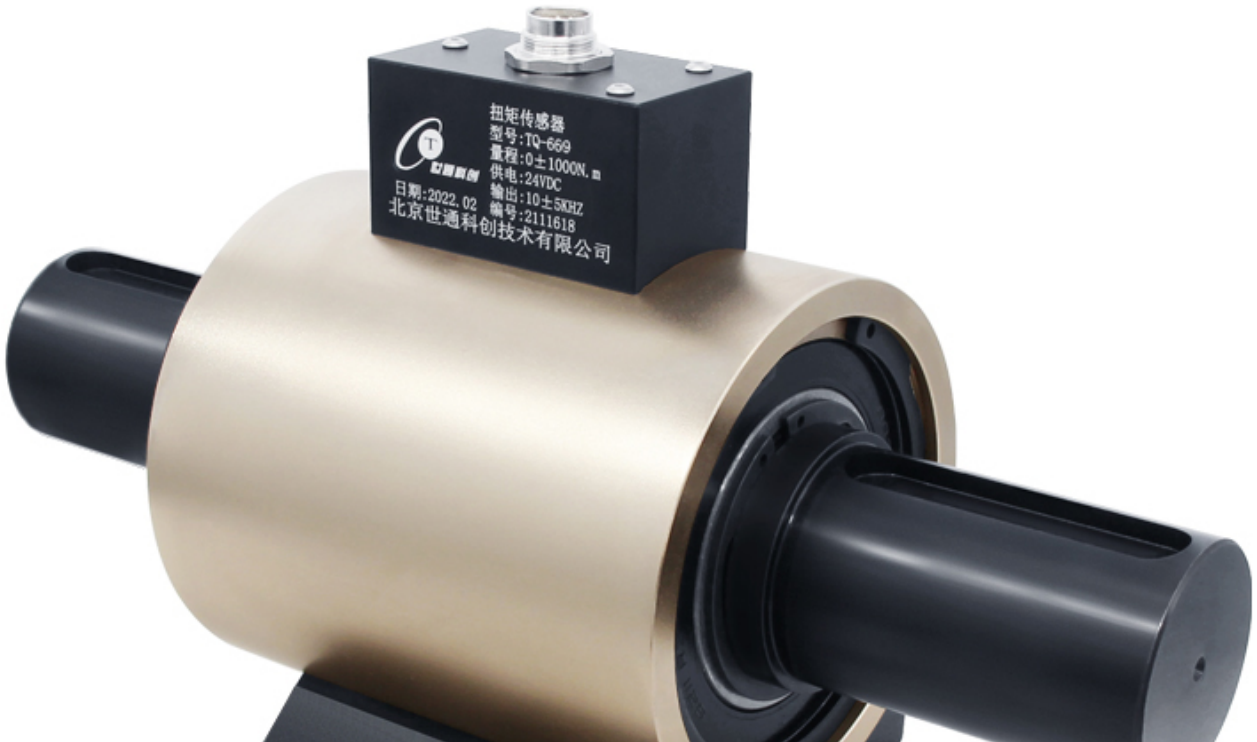
  Function: Detects contact force, pressure distribution, vibration, etc.



- *Available sensors*:

  Force/Torque Sensor

  Function: Measures gripping force and torque to prevent slippage or damage to objects.



Depth Sensor

Function: Provides information on object position, shape, and pose.

Kinematic Model Construction

- Measure the robotic arm's DH parameters
- Implement forward kinematics
- Implement inverse kinematics
- Control the robotic arm using ROS

Measuring DH Parameters

link	a (m)	alpha (rad)	d (m)	theta-offset (rad)
1	0.0	pi/2	0.0	0
2	0.17	0	0.0	0
3	0.07	-pi/2	0.0	pi/2
4	0.0	pi/2	0.11	0
5	0.0	-pi/2	0.0	-pi/2
6	0.0	0	0.09	0

Common Pitfalls (Discovered Iteratively)

ROS2 uses modified DH convention rather than standard DH

θ_1: 0 or π affects kinematic calibration

a_1: Whether zero depends on base_link position

d_6 requires adjustment after gripper installation

Forward Kinematics Implementation

Transformation Matrix

```
T=[cos(theta), -sin(theta)*cos(alpha),sin(theta)*sin(alpha), a*cos(theta);
   sin(theta),  cos(theta)*cos(alpha),cos(theta)*sin(alpha), a*sin(theta);
            0,              sin(alpha),           cos(alpha),           d;
            0,                       0,                    0,           1];
```

```
T06 = simplify(T01 * T12 * T23 * T34 * T45 * T56)
End Effector Position: T06[0:3, 3]
```

Transformation Matrix Construction Process

The transformation matrix is constructed by cascading four fundamental transformations:

1. **Rotation about z_{i-1} by θ_i:**

$$\mathrm{Rot}(z_{i-1}, \theta_i) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. **Translation along z_{i-1} by d_i:**

$$\mathrm{Trans}(z_{i-1}, d_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformation Matrix Construction Process (Continued)

3. **Translation along x_i by a_i:**

$$\mathrm{Trans}(x_i, a_i) = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. **Rotation about x_i by α_i:**

$$\mathrm{Rot}(x_i, \alpha_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformation Matrix Construction Process (Continued)

Final Transformation Matrix Cascade Form

$$^{i-1}T_i = \mathrm{Rot}(z_{i-1}, \theta_i) \cdot \mathrm{Trans}(z_{i-1}, d_i) \cdot \mathrm{Trans}(x_i, a_i) \cdot \mathrm{Rot}(x_i, \alpha_i)$$

Expanded form:

$$^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverse Kinematics Implementation

Joint Angle Calculation

- Step 1: Find wrist center

```
z_hat = Matrix([0, 0, 1]).reshape(3, 1)
Pw = simplify(Pe - d6 * R06 * z_hat)
```

- **Wrist**: Center Point: The intersection of the axes of the last three rotational joints (joints 4, 5, and 6). It decomposes the position and orientation, facilitating the inverse kinematics solution. The position remains unaffected by changes in the end joints.

Solving Joint Rotation Angles (Continued)

$$P_{wx}^2 + P_{wy}^2 + P_{wz}^2 = a_2^2 + 2a_2a_3\cos\theta_3 - 2a_2d_4\sin\theta_3 + a_3^2 + d_4^2$$

$$\cos(\theta_3 + \phi) = \frac{P_w^2 - a_2^2 - a_3^2 - d_4^2}{2a_2\sqrt{a_3^2 + d_4^2}}, \phi = \arctan\frac{d_4}{a_3}$$

$$\theta_3 = \pm\arccos\frac{P_w^2 - a_2^2 - a_3^2 - d_4^2}{2a_2\sqrt{a_3^2 + d_4^2}} - \arctan\frac{d_4}{a_3}$$

$$P_{wx}^2 + P_{wy}^2 = (a_2\cos\theta_2 + a_3\cos(\theta_2 + \theta_3) - d_4\sin(\theta_2 + \theta_3))^2$$

$$\tan(\theta_1) = \frac{p_{wy}}{p_{wx}} \Rightarrow \theta_1 = \arctan 2(p_{wy}, p_{wx})$$

- The distance from point P_w to the origin of the base only depends on θ_3. The projection distance of this distance on the XY plane only depends on θ_2 and θ_3, and the angle only depends on θ_1.

Solving Joint Rotation Angles (Continued)

Let R_3^6 be the Rotation Matrix from frame 3 to frame 6, then

Solution 1:

$$\theta_5 = \operatorname{atan2}\left(\sqrt{r_{31}^2 + r_{32}^2},\ r_{33}\right), \theta_4 \qquad = \operatorname{atan2}\left(\frac{r_{23}}{\sin\theta_5},\ \frac{r_{13}}{\sin\theta_5}\right),$$

$$\theta_6 = \operatorname{atan2}\left(\frac{r_{32}}{\sin\theta_5},\ -\frac{r_{31}}{\sin\theta_5}\right).$$

Solution 2:

$$\theta_5' = -\operatorname{atan2}\left(\sqrt{r_{31}^2 + r_{32}^2},\ r_{33}\right), \theta_4' \qquad = \operatorname{atan2}\left(\frac{r_{23}}{\sin\theta_5'},\ \frac{r_{13}}{\sin\theta_5'}\right) = \theta_4 + \pi,$$

$$\theta_6' = \operatorname{atan2}\left(\frac{r_{32}}{\sin\theta_5},\ -\frac{r_{31}}{\sin\theta_5}\right) = \theta_6 + \pi.$$

Note that there is coupling when $\theta_5 = 0$.

Controlling the Manipulator Using ROS

Quaternion

ROS2 describes the pose using the quaternion $q = (x, y, z, w)$, which encodes 3D rotation through the rotation axis and half-angle, avoiding the gimbal lock problem of Euler angles (the loss of one rotational degree of freedom).
Normalization: $x^2 + y^2 + z^2 + w^2 = 1$

Assume the rotation axis is the normalized vector $\mathbf{u} = (u_x, u_y, u_z)$ and the rotation angle is θ. Then the quaternion can be expressed as:

$$\mathbf{q} = \left(\cos\left(\frac{\theta}{2}\right),\ u_x \sin\left(\frac{\theta}{2}\right),\ u_y \sin\left(\frac{\theta}{2}\right),\ u_z \sin\left(\frac{\theta}{2}\right) \right)$$

That is:

$$w = \cos\left(\tfrac{\theta}{2}\right), \quad (x, y, z) = \mathbf{u} \cdot \sin\left(\tfrac{\theta}{2}\right)$$

Robot Kinematic Parameter Calibration and Hand-Eye Calibration

- **Objective**: Improve robot accuracy and enable precise visual perception systems
- DH Parameter Calibration: Obtain **calibrated** DH parameters
- Camera Calibration: Determine the **camera matrix** and **distortion coefficients**
- Hand-Eye Calibration: Acquire the **transformation matrix** between the camera and the robot end-effector

DH Parameter Calibration: Error Compensation Model

Error compensation model:

$$[Actual] = [C][sensed]$$

- $[sensed]$: Sensor measurement value
- $[C]$: **Error compensation matrix to be calibrated**
- $[Actual]$: Theoretical model output value

DH Parameter Calibration: Theoretical Procedure

1. **Data Collection**: Collect N different robot poses.
2. **Equation Construction**: $T_{\text{sensed}}^{(k)} = C \cdot T_{\text{Actual}}^{(k)}, \quad k \in [N]$
3. **Solve for** $[C]$: Convert the equations into a least squares optimization problem to find the optimal $[C]$:

$$\min_{C} \sum_{k=1}^{N} \left\| T_{sensed}^{(k)} - C \cdot T_{Actual}^{(k)} \right\|^2$$

4. **Parameter Decomposition**: Use geometric analysis or differential kinematics to decompose $[C]$ into **DH parameter corrections** ($\Delta a, \Delta d, \Delta\alpha, \Delta\theta$).
   Here, $[\mathrm{C}] = \begin{bmatrix} R_C & t_C \\ 0 & 1 \end{bmatrix}$
5. **Update DH Parameters**

DH Parameter Calibration: Data Collection

Vertex	theta1	theta2	theta3	theta4	theta5	theta6
1	-0.051	2.156	-0.422	0.000	0.968	-1.417
2	-0.004	2.131	-0.398	-2.866	1.431	-1.450
3	-0.696	2.055	0.000	0.000	0.628	0.000
4	-0.459	2.190	-0.628	0.000	1.240	-1.438

DH Parameter Calibration: Code Implementation

The core least squares optimization is implemented in the `iterative_calibration` function:

```
function dh_calibrated =
iterative_calibration(dh_initial, joint_configs, param_indices, target_poses)
  % ... [Initialization code] ...
  for iter = 1:max_iterations
    % 1. Build Jacobian matrix and error vector
    [total_jacobian, total_error_vector] =
    build_pose_calibration_system(dh_calibrated, joint_configs, param_indices, target_poses);
    % 2. Core least squares computation
    lambda = 1e-8;  % Regularization parameter
    JTJ = total_jacobian' * total_jacobian + lambda * eye(size(total_jacobian, 2));
    JTe = total_jacobian' * total_error_vector;
    delta_params = JTJ \ JTe;  % Solve least squares problem
    % ... [Step size adjustment and parameter update] ...
  end
end
```

DH Parameter Calibration: Result Analysis

- Calibrated DH Parameters

Link	a (m)	α (deg)	d (m)	θ-offset (deg)
1	-0.115158	89.10	-0.027282	1.92
2	0.079603	3.11	-0.616748	-0.38
3	0.035327	-91.93	0.605532	90.48
4	0.005171	97.39	0.106892	-1.26
5	0.024034	-97.59	0.010883	-90.18
6	-0.006751	-0.47	0.017135	-1.51

DH Parameter Calibration: Result Analysis

Vertex	Error Before Calibration (m)	Error After Calibration (m)
Vertex 1	0.009448	0.009981
Vertex 2	0.013711	0.013474
Vertex 3	0.004012	0.003570
Vertex 4	0.009906	0.008885

- Average error before calibration: 0.042002
- Average error after calibration: 0.033400
- Accuracy improvement: approximately `(0.042002 - 0.033400) / 0.042002 * 100% = 20.5%`

Camera Calibration: Camera Parameter Model

1. **Camera Intrinsic Matrix** $\begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

- f_x, f_y: Focal lengths (in pixels).
- s: Skew coefficient (usually 0).
- (c_x, c_y): Principal point coordinates.

2. **Distortion Coefficients**

- Real lenses introduce distortion, mainly radial and tangential. The distortion coefficients D are expressed as:
  - $D = [k_1, k_2, p_1, p_2, k_3]$
- Where k_1, k_2, k_3 are radial distortion coefficients, and p_1, p_2 are tangential distortion coefficients.

Camera Calibration: Imaging Geometry Model

Projection Equation:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Intrinsic Matrix } \boldsymbol{K}} \cdot \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix}}_{\text{Extrinsic Matrix } [\boldsymbol{R}|\boldsymbol{t}]} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

- $(\boldsymbol{u}, \boldsymbol{v})$: Pixel coordinates
- $(\boldsymbol{X_w}, \boldsymbol{Y_w}, \boldsymbol{Z_w})$: World coordinates
- $\boldsymbol{s}$: Scale factor

Camera Calibration: Lens Distortion Model

Radial Distortion:

$$\begin{cases} x_{\text{corr}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{\text{corr}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{cases}$$

Tangential Distortion:

$$\begin{cases} x_{\text{corr}} = x + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{\text{corr}} = y + p_1(r^2 + 2y^2) + 2p_2 xy \end{cases}$$

Distortion Type	Physical Cause	Effect
Radial Distortion	Lens curvature	Curved image edges
Tangential Distortion	Lens-sensor misalignment	Image skew/deformation

Camera Calibration: Optimization Principle

Parameters are solved by minimizing the reprojection error:

$$\min_{K,\text{dist},R,t} \sum_{i=1}^{N} \left\| \begin{bmatrix} u_i \\ v_i \end{bmatrix}_{\text{observed}} - \pi \left(K \cdot [R|t] \begin{bmatrix} X_{w_i} \\ Y_{w_i} \\ Z_{w_i} \\ 1 \end{bmatrix} \right) \right\|^2$$

- $\boldsymbol{\pi}$: Projection function including distortion correction
- $\boldsymbol{N}$: Number of calibration points (typically ≥ 50)

Camera Calibration: Result Analysis

After capturing 25 valid chessboard images, the calibration results are as follows:

- **Reprojection Error**: 0.16898328190736067
- **Camera Matrix**:

$$K = \begin{bmatrix} 595.70605082 & 0 & 314.03378529 \\ 0 & 595.55672367 & 222.78923338 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Distortion Coefficients**:

$$D = [-4.64259907e-01, 2.98907023e-01, 1.06860920e-03, -2.46000222e-04, -1.29073349e-01]$$

Parameter	Value
Reprojection Error	0.16898328190736067
Focal Length f_x	595.70605082
Focal Length f_y	595.55672367
Principal Point c_x	314.03378529
Principal Point c_y	222.78923338

- **Reprojection Error**: The low error of 0.16898328190736067 indicates reliable calibration results.
- **Camera Matrix**: The focal lengths f_x, f_y are close to 600 pixels, and the principal point is near the image center, as expected.
- **Distortion Coefficients**: The relatively small distortion coefficients indicate minor lens distortion.

Hand-Eye Calibration: Principle

The mathematical core of hand-eye calibration is to solve for the transformation matrix X, which represents the transformation from the camera coordinate system to the end-effector coordinate system. In the `eye-in-hand` configuration, the camera moves with the end-effector, and calibration is based on the following equation:

$$AX = XB$$

where:

- A: The transformation matrix of the robot end-effector between two poses (robot motion).
- B: The relative transformation matrix of the camera with respect to a fixed calibration target (e.g., chessboard).
- X: The fixed transformation matrix from the camera to the end-effector to be solved, including the rotation matrix R_X and translation vector T_X.

Hand-Eye Calibration: Tsai-Lenz Algorithm Principle

The Tsai-Lenz algorithm solves the hand-eye calibration problem in two steps:

1. **Rotation Part**: Use the rotation matrices of A and B to solve for R_X:

$$R_A R_X = R_X R_B$$

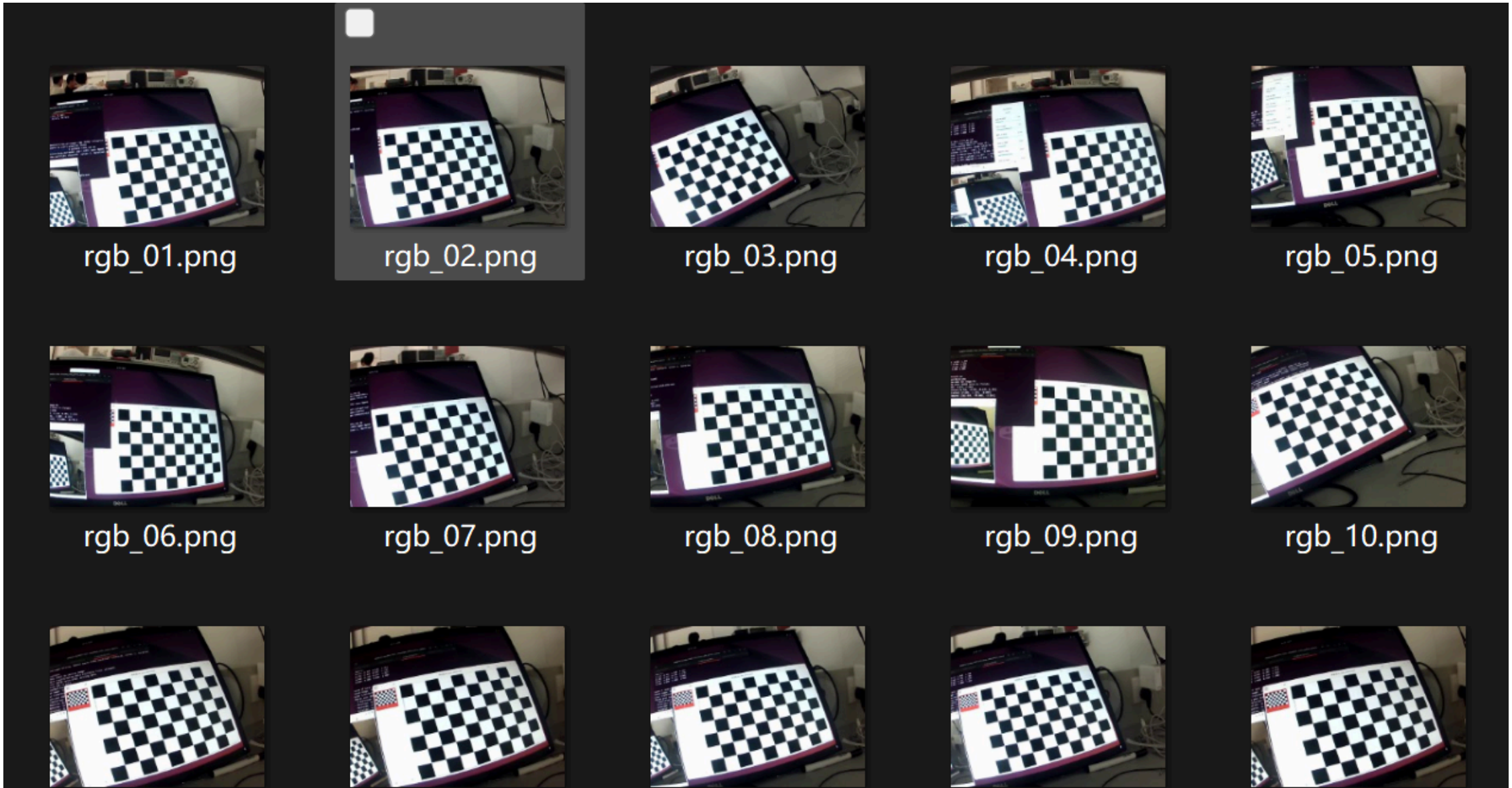
Here, R_A and R_B are the rotation components of A and B, respectively. R_X can be solved using Singular Value Decomposition (SVD) or quaternion methods.

2. **Translation Part**: After obtaining R_X, solve for the translation vector T_X:

$$T_A + R_A T_X = T_X + R_X T_B$$

Rearranging and applying the least squares method yields T_X.

Hand-Eye Calibration: Data Collection



Hand-Eye Calibration: Data Collection

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
------        2025/7/10     12:41            326 tf_01.txt
------        2025/7/10     12:42            326 tf_02.txt
------        2025/7/10     12:43            329 tf_03.txt
------        2025/7/10     12:44            329 tf_04.txt
------        2025/7/10     12:45            327 tf_05.txt
------        2025/7/10     12:45            325 tf_06.txt
------        2025/7/10     12:36            329 tf_07.txt
------        2025/7/10     12:37            326 tf_08.txt
------        2025/7/10     12:38            325 tf_09.txt
------        2025/7/10     12:24            329 tf_10.txt
------        2025/7/10     12:25            329 tf_11.txt
------        2025/7/10     12:26            329 tf_12.txt
------        2025/7/10     12:27            329 tf_13.txt
------        2025/7/10     12:27            329 tf_14.txt
```

Hand-Eye Calibration: Results

```
sist@sist-OptiPlex-7040: ~/Handeye_calib_python_opencv
sist@sist-OptiPlex-7040:~/Handeye_calib_python_opencv$ python3 hand_eye_calibrat
ion.py --eye_in_hand --camera_intrinsics camera_params.txt

Using user-provided camera intrinsics
Camera matrix:
[[595.70605   0.      314.03378]
 [  0.      595.5567  222.78923]
 [  0.        0.        1.     ]]
Distortion coefficients: [-4.6425989e-01  2.9890701e-01  1.0686092e-03 -2.460002
1e-04
 -1.2907335e-01]
Found 20 images
Successfully loaded data 01: rgb_01.png
Successfully loaded data 02: rgb_02.png
Successfully loaded data 03: rgb_03.png
Successfully loaded data 04: rgb_04.png
Successfully loaded data 05: rgb_05.png
Successfully loaded data 06: rgb_06.png
Successfully loaded data 07: rgb_07.png
Successfully loaded data 08: rgb_08.png
Successfully loaded data 09: rgb_09.png
Successfully loaded data 10: rgb_10.png
Successfully loaded data 11: rgb_11.png
Successfully loaded data 12: rgb_12.png
Successfully loaded data 13: rgb_13.png
```

Hand-Eye Calibration: Result Analysis

- Transformation Matrix

$$\begin{bmatrix} -0.56456052 & -0.74145612 & 0.36264893 & -0.40747802 \\ -0.60736853 & 0.67070021 & 0.42575193 & -0.80709697 \\ -0.55890509 & 0.02010118 & -0.82898797 & -0.6781105 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

- Rotation Matrix:

$$\begin{bmatrix} -0.56456052 & -0.74145612 & 0.36264893 \\ -0.60736853 & 0.67070021 & 0.42575193 \\ -0.55890509 & 0.02010118 & -0.82898797 \end{bmatrix}$$

- Translation Vector:

$$\begin{bmatrix} -0.40747802 & -0.80709697 & -0.6781105 \end{bmatrix}$$

Hand-Eye Calibration: Result Analysis

- The translation vector error is unacceptable.
  Reflection on error sources:
- DH parameter calibration error is too large
  ---> Use theoretical values instead
- The Tsai-Lenz algorithm is sensitive to noise. It is possible that the pose changes during image acquisition were too small, resulting in similar configurations and insufficient excitation in rotation and translation.
  ---> Re-acquire images with a wider range of pose changes

Hand-Eye Calibration: Result Analysis

- Transformation Matrix

$$\begin{bmatrix} 0.9786423 & 0.16541087 & -0.12205938 & -0.07895812 \\ -0.16500218 & 0.98620054 & 0.01351946 & 0.2536186 \\ 0.1226113 & 0.00690934 & 0.99243072 & 2.74594491 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

- Rotation Matrix:

$$\begin{bmatrix} 0.9786423 & 0.16541087 & -0.12205938 \\ -0.16500218 & 0.98620054 & 0.01351946 \\ 0.1226113 & 0.00690934 & 0.99243072 \end{bmatrix}$$

- Translation Vector:

$$\begin{bmatrix} -0.07895812 & 0.2536186 & 2.74594491 \end{bmatrix}$$

Hand-Eye Calibration: Rotation Matrix Analysis

- Row Norms:
  - 1st row norm: $\sqrt{0.9786423^2 + 0.16541087^2 + (-0.12205938)^2} \approx 1.000$
  - `...` All row norms are very close to 1 (error on the order of 10^{-4}), indicating that each row is a **unit vector**.
- Orthogonality Between Rows:
  - Dot product of 1st and 2nd rows: $0.9786423 \times (-0.16500218) + 0.16541087 \times 0.98620054 + (-0.12205938) \times 0$
  - `...` All dot products are close to zero (error on the order of 10^{-4}), indicating the matrix is **approximately orthogonal**.
- Determinant Check:

$\det(R) = 0.9786423 \times (0.98620054 \times 0.99243072 - 0.01351946 \times 0.00690934) - \cdots \approx 0.99986$

Hand-Eye Calibration: Translation Vector Analysis

- The camera is located at position $[-\mathbf{0.079}, \mathbf{0.254}, \mathbf{2.746}]^{\boldsymbol{T}}$ relative to the end-effector.
- The camera's orientation relative to the end-effector is described by Euler angles ($\mathrm{Yaw} \approx -9.56^\circ$, $\mathrm{Pitch} \approx -7.04^\circ$, $\mathrm{Roll} \approx 0.4^\circ$).

Hand-Eye Calibration: Algorithm Comparison

According to the OpenCV-Python Tutorials, there are five hand-eye calibration methods.

◆ HandEyeCalibrationMethod

enum **cv::HandEyeCalibrationMethod**

```
#include <opencv2/calib3d.hpp>
```

Enumerator	
CALIB_HAND_EYE_TSAI Python: cv.CALIB_HAND_EYE_TSAI	A New Technique for Fully Autonomous and Efficient 3D Robotics Hand/Eye Calibration **[277]**.
CALIB_HAND_EYE_PARK Python: cv.CALIB_HAND_EYE_PARK	Robot Sensor Calibration: Solving AX = XB on the Euclidean Group **[216]**.
CALIB_HAND_EYE_HORAUD Python: cv.CALIB_HAND_EYE_HORAUD	Hand-eye Calibration **[129]**.
CALIB_HAND_EYE_ANDREFF Python: cv.CALIB_HAND_EYE_ANDREFF	On-line Hand-Eye Calibration **[13]**.
CALIB_HAND_EYE_DANIILIDIS Python: cv.CALIB_HAND_EYE_DANIILIDIS	Hand-Eye Calibration Using Dual Quaternions **[67]**.

- Park transformation matrix

$$\begin{bmatrix} 0.92913649 & 0.33900283 & 0.1475888 & -0.22742445 \\ 0.20159417 & -0.79910792 & 0.56638002 & 0.41854903 \\ -0.30994381 & 0.49649131 & 0.81082132 & 3.0588169 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

- Horaud transformation matrix

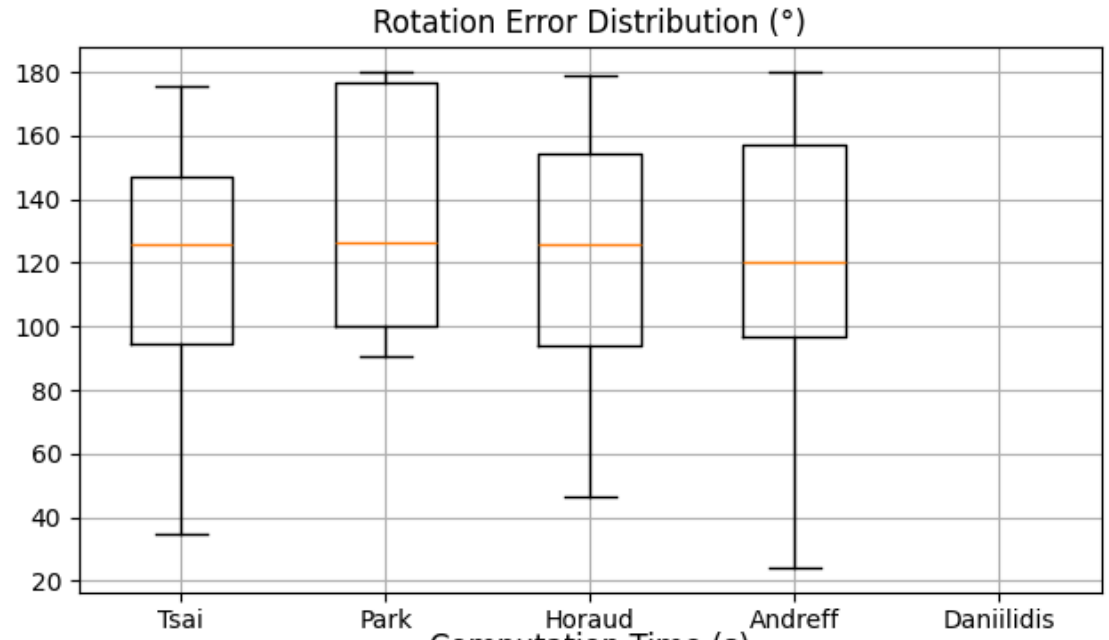
$$\begin{bmatrix} 0.92913649 & 0.33900283 & 0.1475888 & -0.22742445 \\ 0.20159417 & -0.79910792 & 0.56638002 & 0.41854903 \\ -0.30994381 & 0.49649131 & 0.81082132 & 3.0588169 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

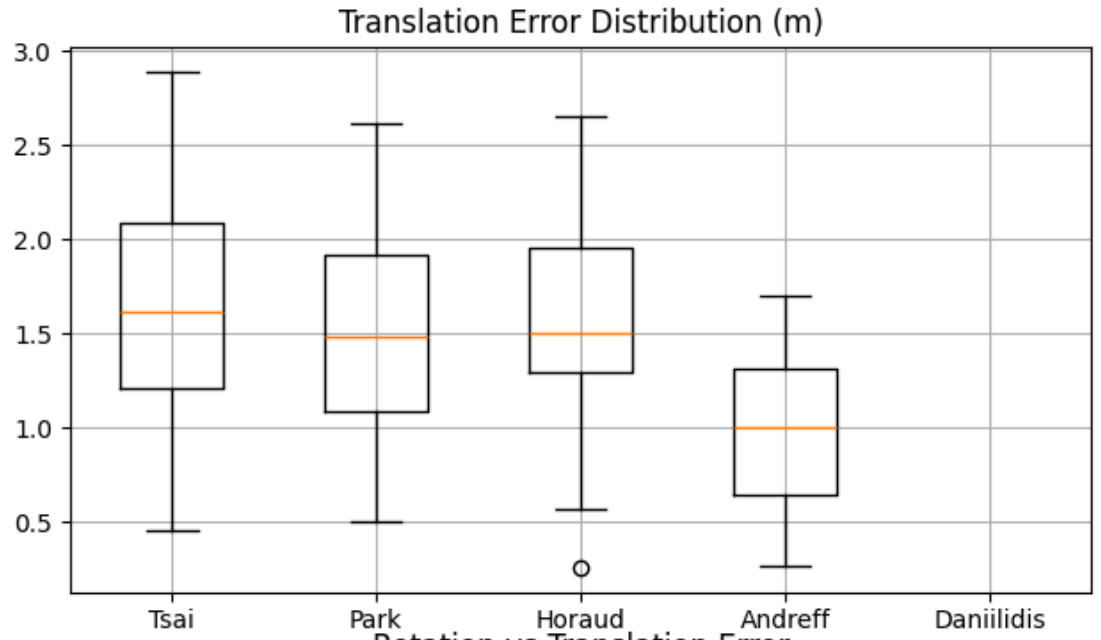
- Andreff transformation matrix

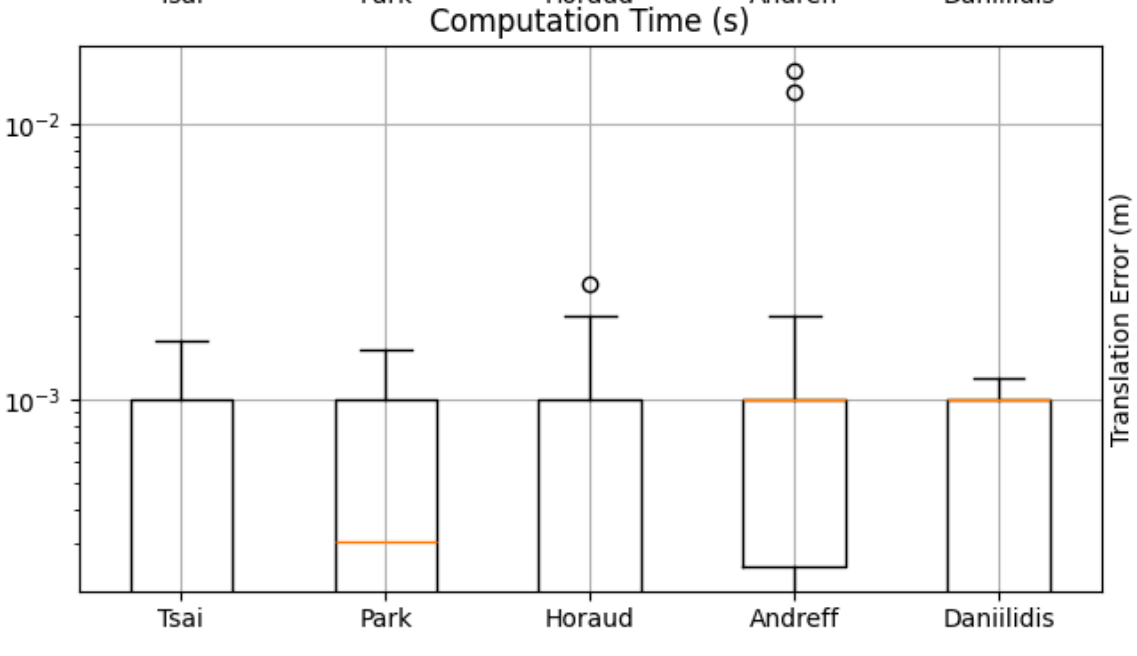
$$\begin{bmatrix} -0.38682304 & 0.92106773 & -0.04474565 & -0.13578577 \\ 0.91903938 & 0.38904711 & 0.06331643 & 0.39791137 \\ 0.07572688 & -0.01663076 & -0.9969899 & 2.75047498 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

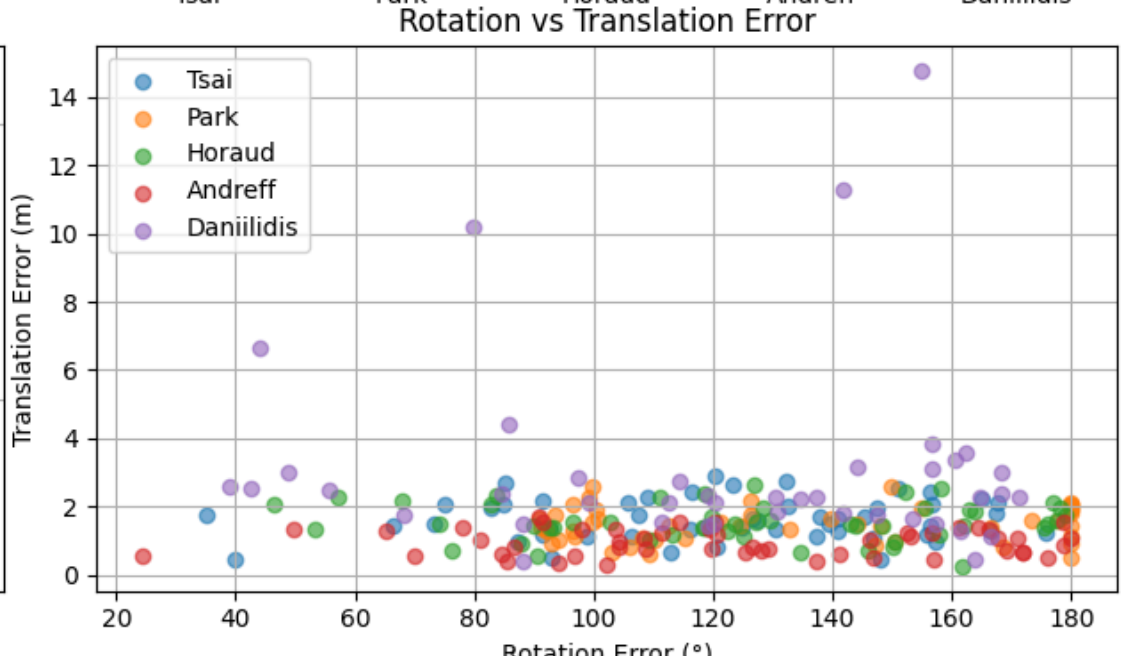
- Daniilidis transformation matrix

$$\begin{bmatrix} -0.99747307 & 0.04552641 & -0.05454197 & -0.35690244 \\ -0.04716684 & -0.99846084 & 0.02917594 & 0.20397047 \\ -0.05312975 & 0.03167478 & 0.99808514 & 0.24079183 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$


Rotation Error Distribution (°)
180
160
140
120
100
80
60
40
20
Tsai
Park
Horaud
Andreff
Daniilidis


Translation Error Distribution (m)
3.0
2.5
2.0
1.5
1.0
0.5
Tsai
Park
Horaud
Andreff
Daniilidis


Computation Time (s)
10^{-2}
10^{-3}
Tsai
Park
Horaud
Andreff
Daniilidis


Rotation vs Translation Error
Tsai
Park
Horaud
Andreff
Daniilidis
Translation Error (m)
14
12
10
8
6
4
2
0
20
40
60
80
100
120
140
160
180

Method	Algorithm Principle	Advantages	Disadvantages
Tsai-Lenz	1. Separates rotation and translation 2. Uses axis-angle for rotation 3. Least squares solution	High computational efficiency Simple implementation Moderate noise robustness	Decoupling rotation and translation may cause bias Accuracy drops under extreme poses
Park	1. Lie algebra-based optimization 2. Minimizes objective function 3. Jointly solves rotation and translation	Highest accuracy Insensitive to initial values Performs well with complex motion trajectories	High computational complexity Sensitive to small sample sizes More complex implementation

Method	Algorithm Principle	Advantages	Disadvantages
Andreff	1. Linear solver 2. Direct matrix equation solution 3. Based on linear algebra	Fastest computation Simplest implementation Low resource requirements	Lowest accuracy Sensitive to noise Requires good initial conditions
Daniilidis	1. Uses dual quaternions 2. SVD decomposition 3. Unified representation of rotation and translation	Good numerical stability Robust to singular poses Accuracy second only to Park	Most complex implementation High computational resource consumption Difficult mathematical theory

Robot Simulation

Utilize computers to intergrate planning and commanding the robot arm.

Why Are We Assigned This Part of Experiment

1. **Mitigate Real-World Risks** by testing algorithms in virtual environments, avoiding hardware damage or safety hazards during development.
2. **Validate Performance** under diverse scenarios (e.g., dynamic obstacles, singularities), ensuring robustness before deployment.
3. **Optimize Efficiency** by rapidly iterating pathfinding strategies and controller tuning without costly physical trials.
4. **Enable Scalability** for complex systems (swarms, human-robot collaboration) in a reproducible, controlled setting.
5. **Accelerate Innovation** by bridging theoretical models and real-world applicability, reducing development time and costs.

These experiments transform theoretical motion planning into reliable, deployable robotic behaviors. **So, let's start!**

Simmulation Method 1: Matlab default Toolbox

By using `Robotics System Toolbox`, we call built-in methods to plot random postures and provide an interactive GUI to visualize how the robot arm points to given points.

Robotics System Toolbox 作者: MathWorks

MathWorks

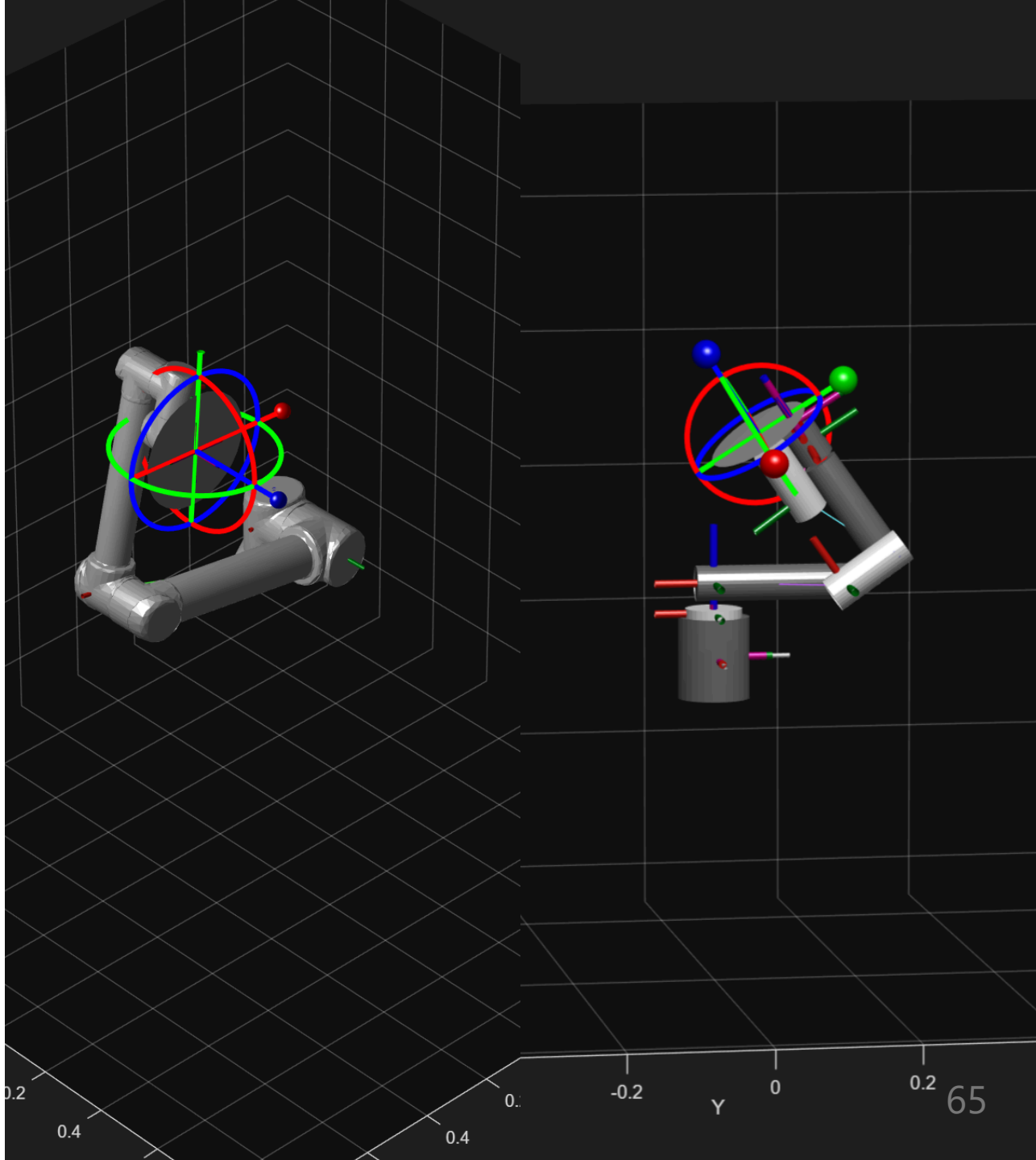
Design, simulate, test, and deploy **robotics** applications

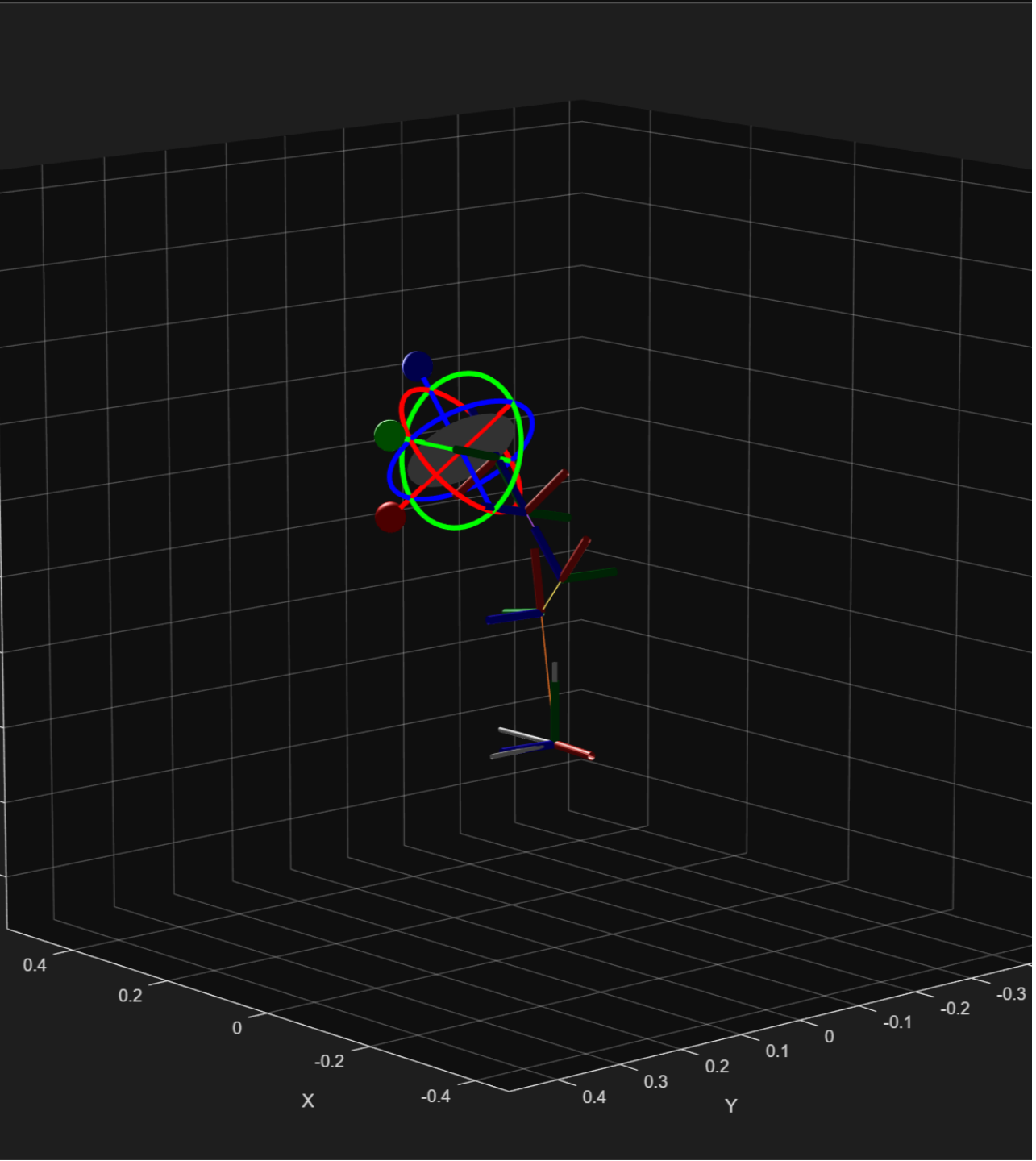
Robotics System Toolbox™ provides tools and algorithms for designing, simulating, testing, and deploying manipulator and mobile **robot** applications. For manipulators, the toolbox includes algorithms

Inverse Kinematics Designer - Design inverse kinematics solvers, configurations, and waypoints

Predefined or Pre-built urdf Robot Model

Direct and ready to use way to experience simulating, exhibits good kinematic and visual performance. Moving the target automatically triggers inverse kinematic calculation and fits the arm to the desired point with a clear and reasonable posture.





Building a Body Tree

By providing Standard DH Parameters and linking `rigidBody` s to `rigidBodyTree`, we build a self-made model for the same arm that the '.urdf' file repersents. Its kinematic characteristics is similar to previous model and supports same graphic interaction. This function is implemented through the following code:

```
gui = interactiveRigidBodyTree(
    si190_robot, MarkerScaleFactor=0.5);
```

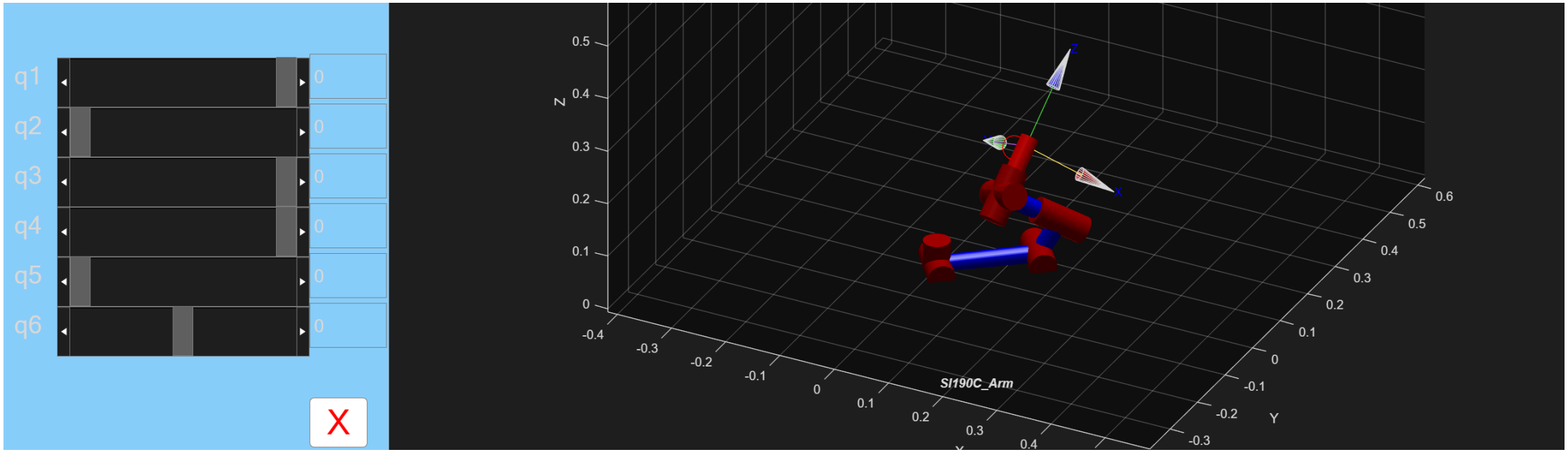
Simulation Method 2: Third Party Toolbox

The third party toolbox offers other function to describe the robot, with joint angle limits. Also has its own forward and inverse kinematics calculating as well as mathematical matrix transformation functions.

```
% Definition
L1=Link('d',d1, 'a', a1, 'alpha', alpha1, 'offset', theta1, 'qlim',[-360 0]*pi/180); % ...

%Return solution for sequential rotations about X, Y, Z axes
rpy=tr2rpy(T1, 'xyz')*180/pi;

% Kinematics
mask_vector = [1,1,1,0,0,0];
axi_val = robot.ikine(T1,'mask',mask_vector,'ilimit',500,'tol',1e-7)
```

Cross Toolbox / Method Comparison

After trying multiple extra approach to the same task, we are able to establish comparisions.

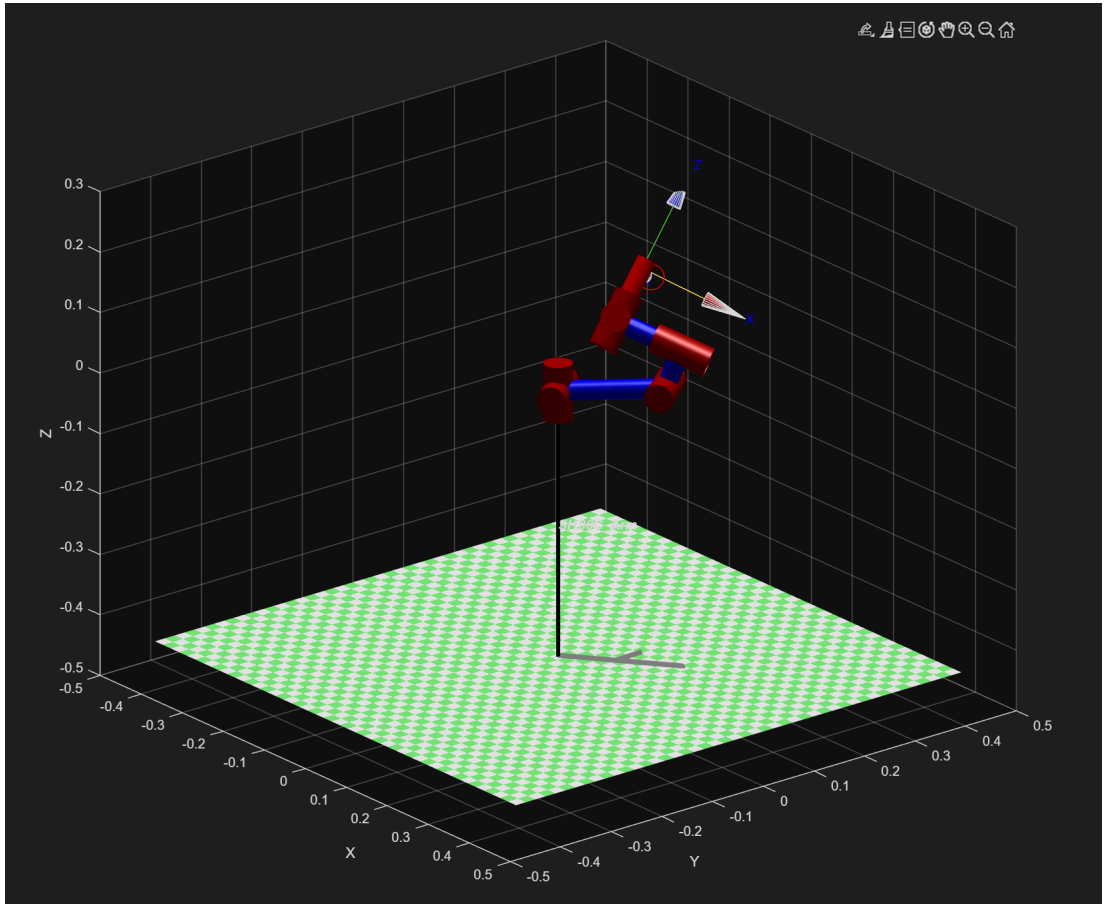


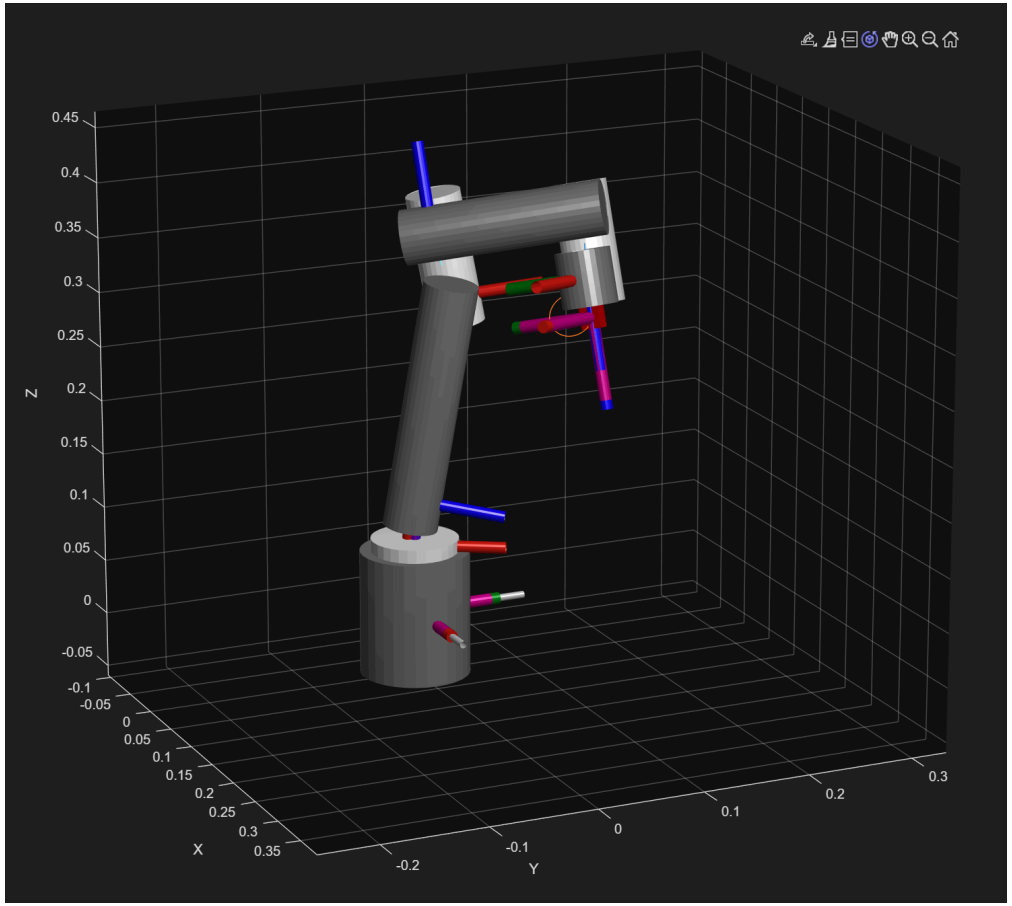
GUI

- **Default Toolbox**: Set a target posture, the robot model fits it real time.
  - **Pros**: Straightforward IK visualize. Allows building a model arbitrarily.
  - **Cons**: Robot may not fit the given target well. Manually increasing iterations sacrifices time performance. Also, the simple tree model has weaker visual performance with lines than those with 3D volume models.
- **Third Party**: Has slides to set the joint angles one by one.
  - **Pros**: Also set joint limits to align with real mechanical limits.
  - **Cons**: This GUI actually provides a FK interaction, fitting a target point requires code modification, rather than dragging the end effector directly.

Trajectory Tracking

The two toolboxes have similar performance on our simple tracking task, with different code implementation. The amount of code may differ but applies similar ideas.





System Toolbox

```
% Config the trajectory
t = (0:0.2:10)'; % Time
count = length(t);
center = [0.1-0.015 0.1+0.015 0.3-0.015];
radius = 0.02;
theta = ( t/t(end) )*2*pi;
traj = (center + radius*cos(theta)*[1/sqrt(2) 1/sqrt(2) 0] + ...
        radius*sin(theta)*[-1/sqrt(6) 1/sqrt(6) 2/sqrt(6)]);

% IK solver
q0 = homeConfiguration(si190_robot);
qs = repmat(q0, count, 1);
ik = inverseKinematics('RigidBodyTree', si190_robot);
weights = [0 0 0 1 1 1];
endEffector = 'body6';
for i = 1:count
    % Target transformation (4x4) — only position is specified
    tgtPose = trvec2tform(traj(i,:));

    % Solve IK and update configuration
    [qs(i,:), ~] = ik(endEffector, tgtPose, weights, q0);

    % Use current solution as initial guess for the next
    q0 = qs(i,:);
end
```

Robotics Toolbox for MATLAB

```
%% Trajectory Planning Sim
% Def Circle
N = (0:0.5:100)';
center = [0.1-0.015 0.1+0.015 0.2-0.015];
radius = 0.02;
theta = ( N/N(end) )*2*pi;
points = (center + ...
          radius*cos(theta)*[1/sqrt(2) 1/sqrt(2) 0] + ...
          radius*sin(theta)*[-1/sqrt(6) 1/sqrt(6) 2/sqrt(6)])';
plot3(points(1,:),points(2,:),points(3,:),'r');

Ttt = transl(points');

% Only consider position | Define mask vector
qq = robot.ikine(Ttt,'mask',[1 1 1 0 0 0]);
hold on;
robot.plot(qq,'tilesize',0.02);
```

Algorithm Optimizing

- **BFBS**: Faster convergence for well-conditioned problems; used in general-purpose IK with simple constraints; but struggles near singularities.
- **L-M**: Robust near singularities; used in high-precision tasks or ill-posed configurations; but is slower and more sensitive to orientation weights.

Our calculation is easy and does *not require much time* even equipped with a slower method. Our task also *doesn't involve orientation weight changing*, but cares about the *accuracy* under our requirements. Therefore, to avoid limit violation, **L-M** method is better than the default **BFBS** algorithm.

```
% Inverse kinematics with built model
ik = inverseKinematics('RigidBodyTree', si190_robot, 'SolverAlgorithm', 'LevenbergMarquardt');
```

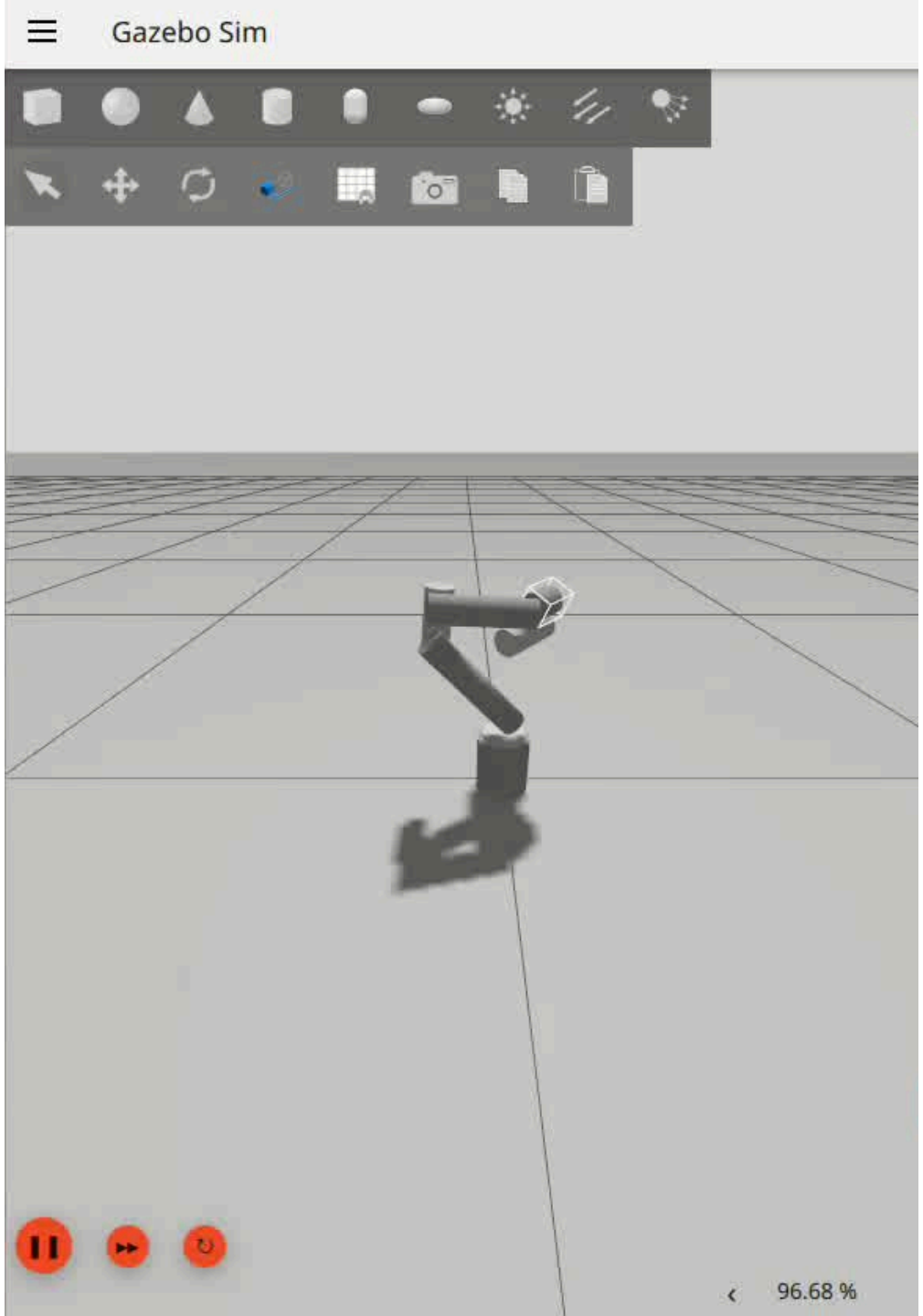
Initial Guess Optimize

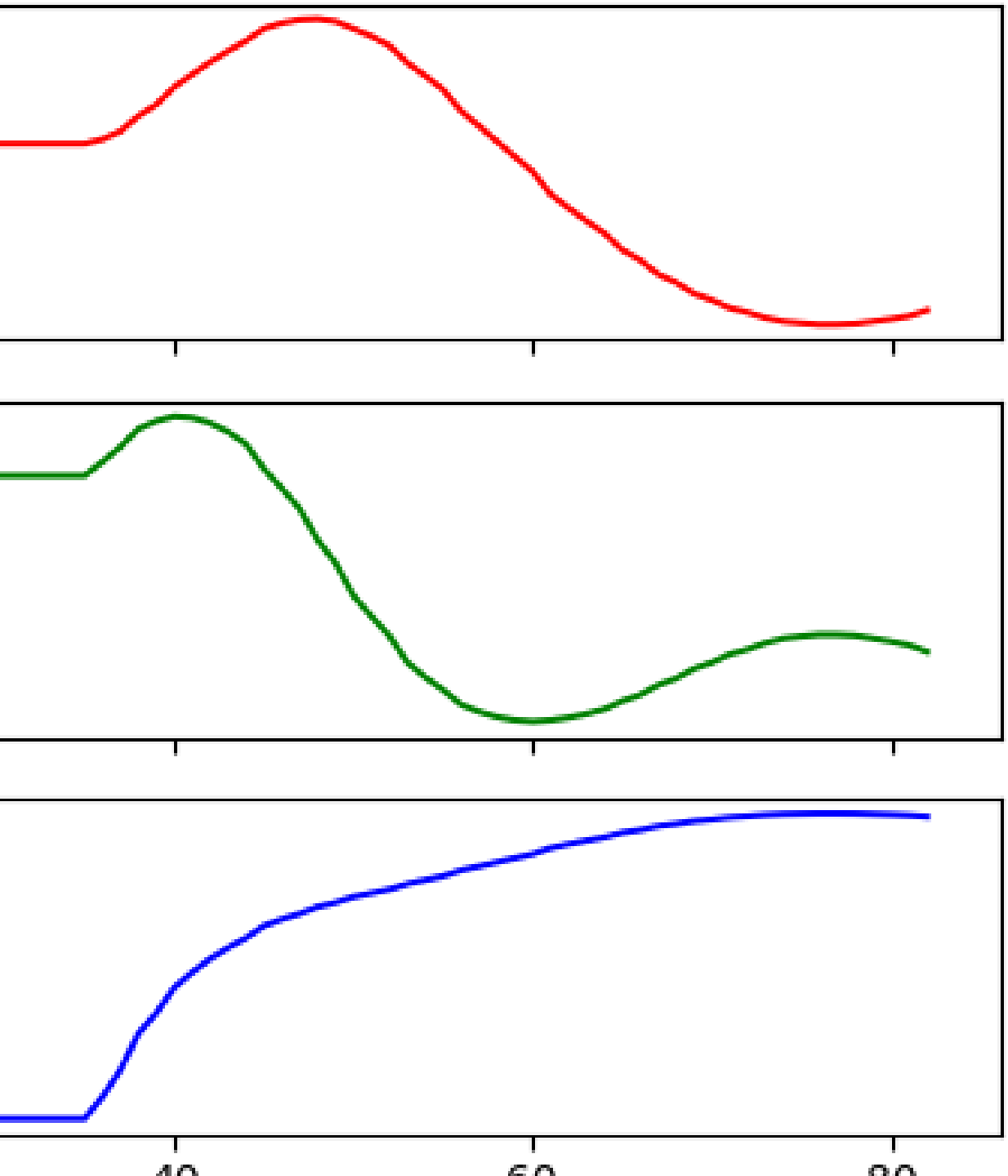
Observe and set an approximate position, rather than home position, to avoid bad suboptimal solutions.

```
% Manually set initial guess to achieve better solution (limits voilation, etc.)
q0 = struct('JointName', {'joint_0', 'joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', },
            'JointPosition', {-2.7, 0.0, -0.6, -2.2, 3.14, 0.0});
```

Simulation Method 3: Gazebo and End Effector Monitoring

- Publisher
  By completing the publisher code in the provided package setup, ros2 uses Gazebo to send joint command to the simulation environment. It also control the real bot with same command synchronously, if provided `env:=real` param.





Simulation Method 3: Gazebo and End Effector Monitoring

- Listener
  By 'listening' (i.e. monitoring) on the `'/tf'` topic and process the data mathematically (transforming the 7-variable posture to transformation matrix), we can draw the line chart of xyz coordinates using `matplotlib`.

Realted Exploring and Analyzing

Publishing and Subcribing on each topics.

```
self.publisher_ = self.create_publisher(JointTrajectory,
      '/joint_position_controller/joint_trajectory', 10)
# ———————————————————————————————————————— #
self.subscription = self.create_subscription(TFMessage,
                    '/tf', self.listener_callback, 10)
```

Mathematical translation from posture to Matrix.

```
# Compute the full 4x4 homogeneous transformation matrix
T = np.eye(4)
T[0:3, 0:3] = Rotation.from_quat([qx,qy,qz,qw]).as_matrix()
T[0:3, 3] = [tx, ty, tz]  # Set translation
t_list.append(T)
```

Plot creating and visualizing data real time.

```
# Initialize data storage for plotting
self.x_data = deque(maxlen=100) # y, z the same
self.time_data = deque(maxlen=100)
self.counter = 0

# Set up the plot
plt.ion()  # Interactive mode on
self.fig, (self.ax_x, self.ax_y, self.ax_z) = plt.subplots(3, 1, sharex=True)
self.fig.suptitle('End-Effector Position')
```

```
def update_plot(self):
    self.line_x.set_data(self.time_data, self.x_data)

    # Rescale and redraw
    for ax in [self.ax_x, self.ax_y, self.ax_z]:
        ax.relim()
        ax.autoscale_view() #  ... then flush the data
```

Further Thoughts

Environment Configuring Explored

An ROS2 package can consist multiple code files and are assigned as following:

> - `CMakeLists.txt` file that describes how to build the code within the package
> - `include/<package_name>` directory containing the public headers for the package
> - `package.xml` file containing meta information about the package
> - `src` directory containing the source code for the package

These directs the building proccess that turns the source code into real executalbles. As a result, `colon build` should follow every modification.

Gazebo vs. MATLAB Comparison

Gazebo Workflow:

URDF robot modeling → 2. Add Gazebo plugins → 3. Build ROS control nodes → 4. Physical simulation verification → 5. Deploy to real robot

MATLAB Workflow:

DH parameter modeling → 2. Design control algorithms → 3. Numerical simulation verification → 4. Generate C++ code → 5. Integrate into ROS

Gazebo vs. MATLAB Comparison

Gazebo Core Advantages:

- High-fidelity environment simulation: Can simulate complex scenarios such as lighting changes and terrain interaction
- Hardware-in-the-loop (HIL) support: Directly connect real sensors via ROS
- Distributed simulation: Supports multi-robot collaborative simulation

Gazebo vs. MATLAB Comparison

MATLAB Core Advantages:

- Algorithm development efficiency:
  - Built-in optimization/machine learning toolboxes
  - Matrix computation speed superior to C++ (for small-scale calculations)
- Visualization tools:
  - Automatically generate professional charts such as Bode plots and phase diagrams
  - 3D visualization response time faster than Gazebo

Robot Moving Smoothing

The 'publisher' is the direct Gazebo commander of the robot, which sends a position command at fixed intervals.

- **The trial**: set the command interval to 1(sec) and message `time_from_start` to 3(sec).
    - *Problems*: Next message overwrites the last one since the interval is shorter than command duration. Position increment too large causing the robot arm to lag and stop when the commanded position switches.
- **Optimize**: Make the command duration and interval the same, increase command frequency and lower the position increment per iteration.

The advantage of Gazebo

Except for the content mentioned in course slides

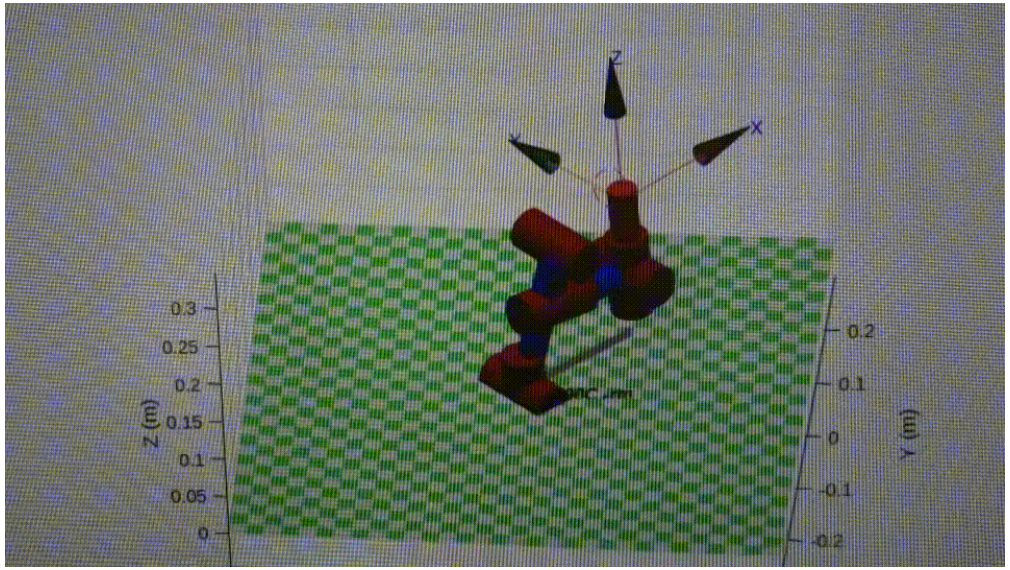
> - Gazebo is a simulation tool for creating realistic, dynamic environments and **simulating robots' physical interactions** with their surroundings. Used for testing algorithms, performing experiments, and developing robot behaviors in a controlled virtual setting.

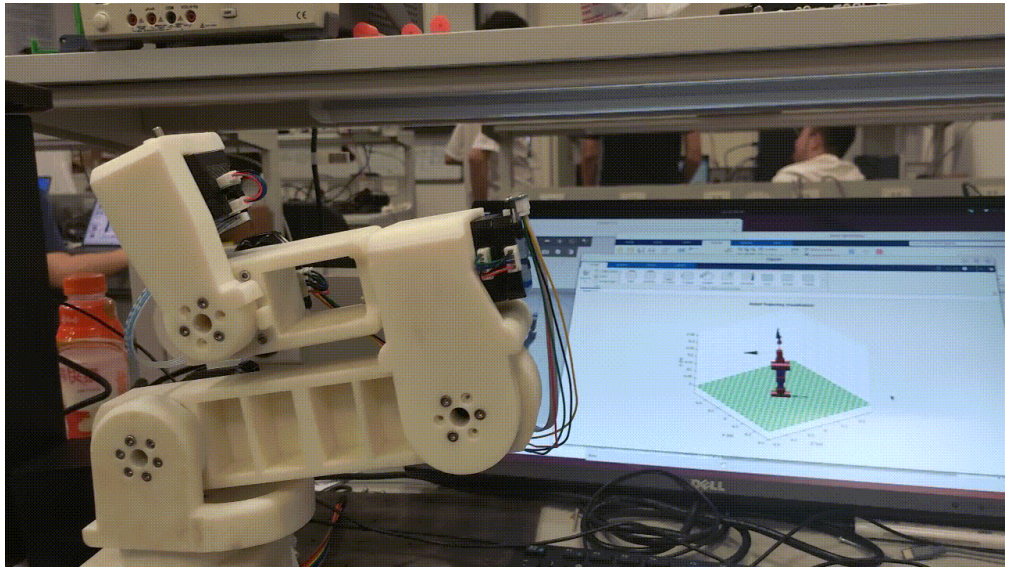
It has some more fundamental features. The repository provided acts as a platform to process and manage data, implement route planning algorithms and transfers easily to the real arm. This is exactly the importance of simulation.

By configuring the repositories, it can also work in collaboration with Matlab, fully leveraging the power of its mathematical tools and robot libraries.

Motion Planning

Simulation





Motion Planning

Inverse Solution Jump

The solution result of the ikine function at adjacent points may jump out of the current continuous solution in the joint space due to numerical errors or multiple solutions, resulting in a sudden change in the joint angle.

Key Improvement Plan: For each solution calculation, use the previous point's solution, `last_guess`, as the initial value.

```
last_guess = initial_guess;
for i = 1:point_num
    q(i,:) = robot.ikine(Ttt(:,:,i), 'mask', mask_vector, 'q0', last_guess);
    last_guess = q(1, :);
end
```

- Increase the `point_num` to make the points more densely distributed, and the inverse solution will be more continuous.

mask_vector

`mask_vector` determines whether each joint needs to be solved by IK. A value of 1 indicates that the calculation is required, while a value of 0 indicates that the calculation is not needed. Different mask values set will result in
According to the experimental results, the mask values for the last three joints can be set to either 0 or 1. When `mask_vector` = `[1,1,1,0,0,0]`, the solver may still make slight adjustments to joint 4-5 to improve numerical stability.

These are the different initial solutions calculated in the two cases.

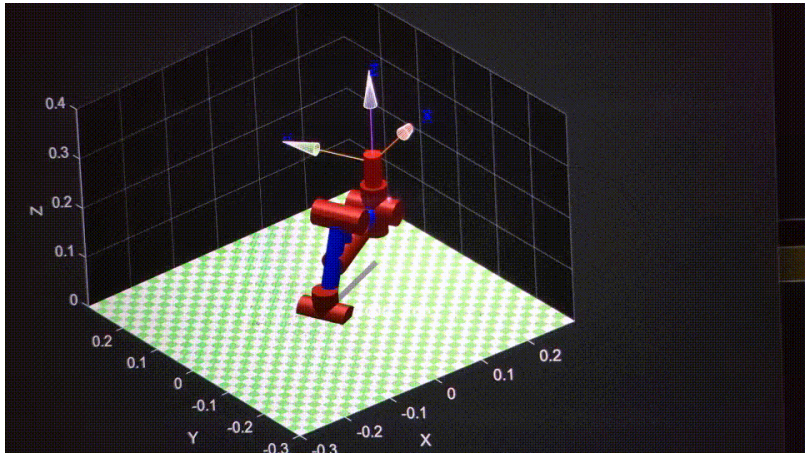
Parameter	link_1	link_2	link_3	link_4	link_5	link_6
`mask_vector = [1,1,1,0,0,0]`	0.9018	1.1628	2.0582	-0.0000	3.0621	-0.9018
`mask_vector = [1,1,1,1,1,1]`	0.8737	1.1864	2.0233	-0.0575	3.0255	0

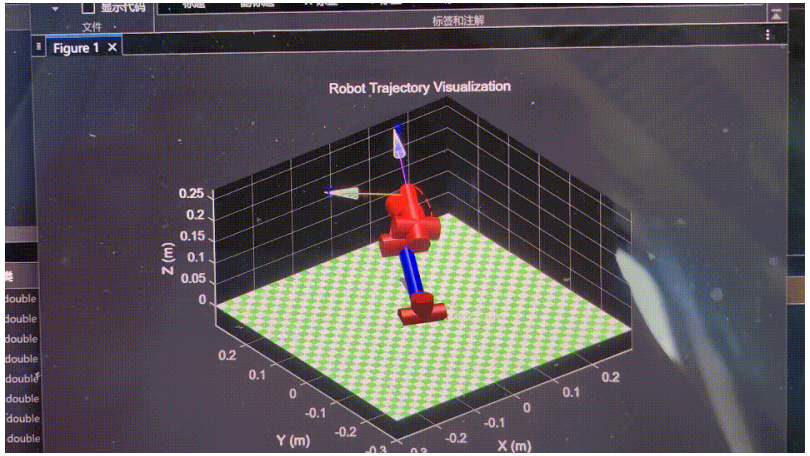
A*

```
% Mark obstacles in the map
radius
obstacles()
for i = 1:size(obstacles, 1)
    dist = sqrt((X - obstacles(1)+0.01).^2 + (Y - obstacles(2)-0.04).^2);
    map(dist <= (robot_radius + radius)) = 1; % Add safety margin
end

% Convert to binary occupancy map
 binary_map = binaryOccupancyMap(map, 1/resolution);
binary_map.LocalOriginInWorld = [x_min, y_min];

% 6. Force start and goal positions to be free
start_grid = world2grid(binary_map, start_pos(1:2));
goal_grid = world2grid(binary_map, goal_pos(1:2));
setOccupancy(binary_map, [start_grid; goal_grid], 0); % Set to free
```

hindrances disposal

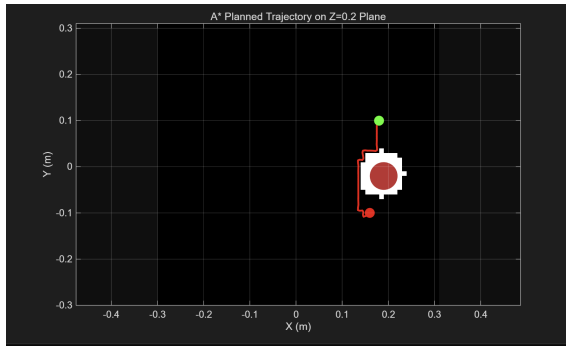
Visualization

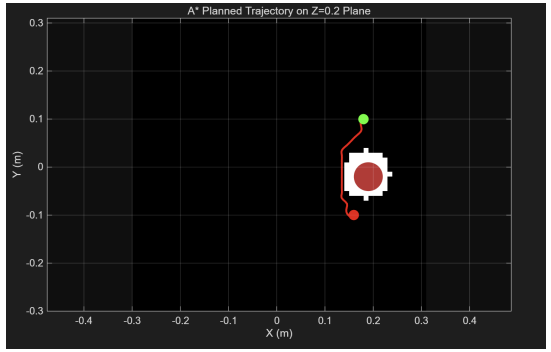


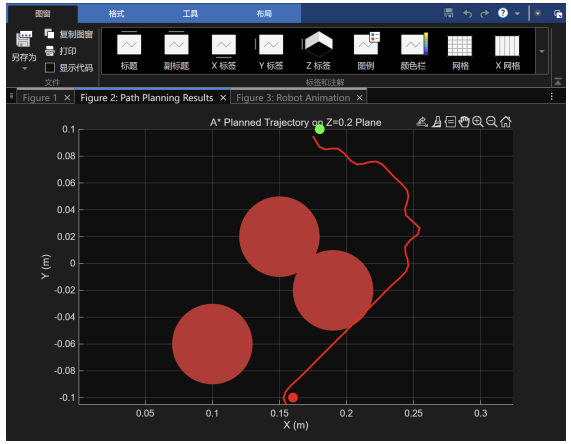


A* Optimization

1. Improve the heuristic function

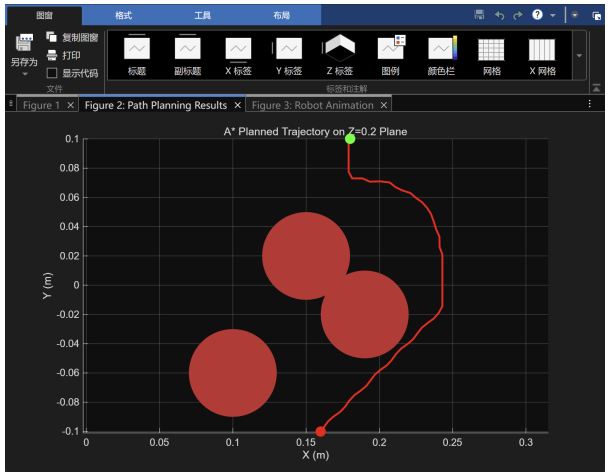






A* Optimization

2. Increase the resolution



3. Improve the weight ratio

WG/WH < 1

WG/WH = 1

WG/WH > 1