

# MArm Robotic Arm Visual Grasping System

[abstract] Project Overview

MArm (Mobile Arm / Manipulator) is an open-source desktop-level six-axis robotic arm visual grasping system based on ROS 2 Humble. This article covers hardware selection, environment setup, calibration procedures, core algorithms, and a complete operational walkthrough, giving you a comprehensive understanding of the "Perception → Planning → Execution" closed loop for robotic arm visual grasping.

## Table of Contents

- [[#1. Design Philosophy & Application Scenarios]]
- [[#2. Hardware Selection & Environment Setup]]
- [[#3. Preparation: Calibration Procedures]]
- [[#4. Core Algorithm Analysis]]
- [[#5. Operation Guide: Launch Procedures]]
- [[#6. Workspace Directory Structure]]

## 1. Design Philosophy & Application Scenarios

### 1.1 Design Philosophy

The MArm project aims to build an **open-source, modular, and easily extensible** desktop-level six-axis robotic arm control system. The system integrates computer vision (OpenCV), motion planning (MoveIt 2), and low-level hardware control to form a complete **Perception-Planning-Execution** closed loop.

In terms of design, the project emphasizes **software-hardware decoupling**, enabling efficient collaboration between modules through standard ROS 2 interfaces (such as `JointState` and the custom `SetArmAction`).

### 1.2 Application Scenarios

| Scenario                          | Description                                                                                                      |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------|
| Education & Research              | An ideal platform for learning ROS 2, robot kinematics, computer vision, and hand-eye calibration                |
| Automated Sorting                 | Combined with visual recognition, enables automatic grasping and sorting of objects by specific colors or shapes |
| Flexible Manufacturing Validation | Validates complex grasping algorithms and multi-sensor fusion solutions                                          |

## 2. Hardware Selection & Environment Setup

### 2.1 Hardware Components

Figure 1: MArm Robotic Arm + QS6490 Development Board + USB Camera

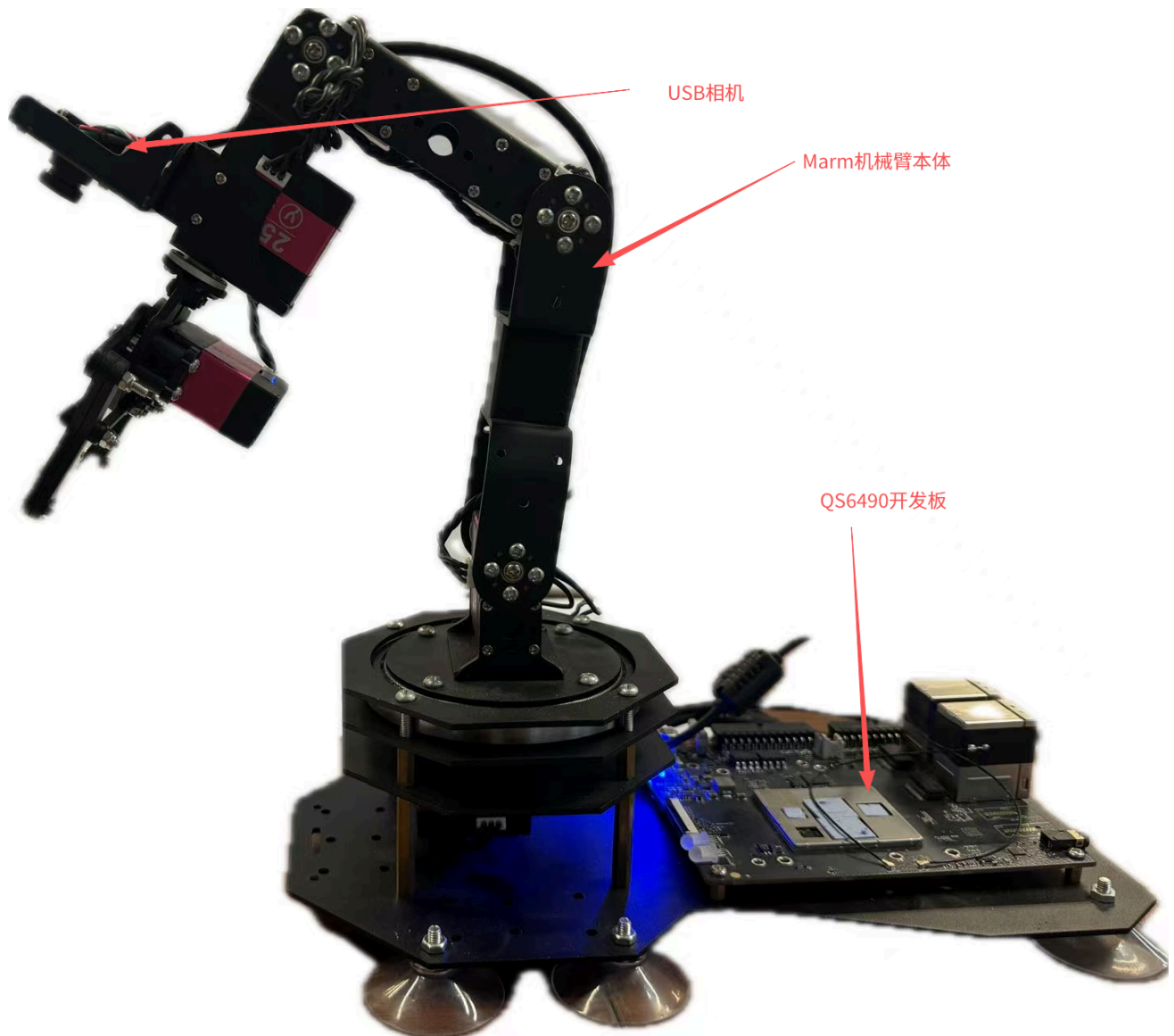
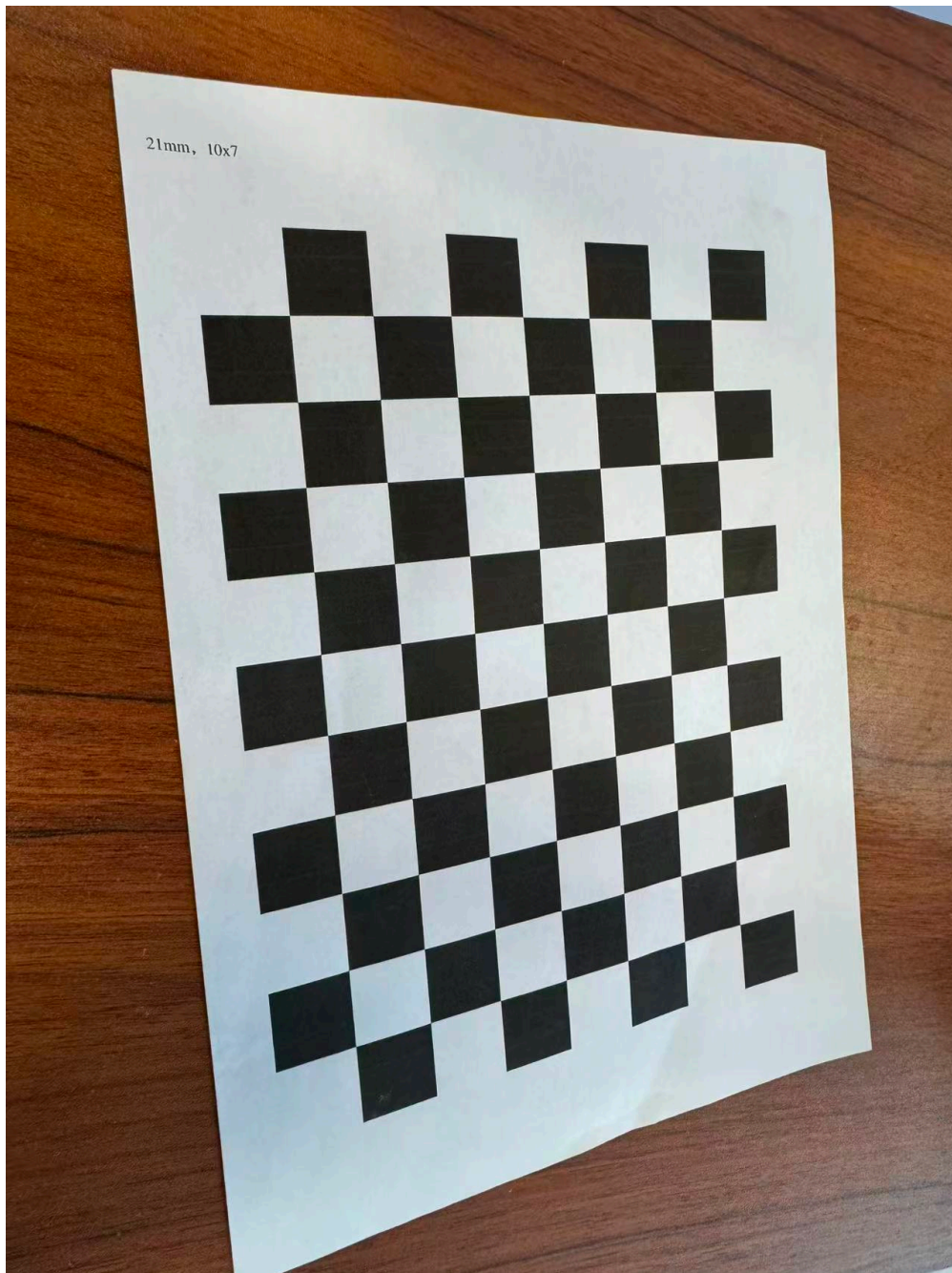


Figure 2: Camera Calibration Board (ArUco QR Code)



| Component         | Model / Specification                                                      | Purpose                                     |
|-------------------|----------------------------------------------------------------------------|---------------------------------------------|
| Robotic Arm Body  | A1 six-DOF desktop robotic arm (3D-printed shell + metal structural parts) | Motion execution                            |
| Drive Actuators   | 6 bus / PWM servos                                                         | Control each joint and end-effector gripper |
| Core Controller   | QS6490 (running ROS 2), USB-to-serial module                               | Host communication and control              |
| Vision Sensor     | RGB monocular USB camera                                                   | Object detection and pose estimation        |
| Calibration Board | ArUco QR code / checkerboard calibration board                             | Camera calibration and hand-eye calibration |

## 2.2 Software Environment

| Item                      | Version / Tool     |
|---------------------------|--------------------|
| Operating System          | Ubuntu 22.04 LTS   |
| Core Framework            | ROS 2 Humble       |
| Motion Planning           | Movelt 2           |
| Vision Processing         | OpenCV + cv_bridge |
| Coordinate Transformation | tf2_ros            |
| Hand-Eye Calibration      | easy_handeye2      |

## 2.3 Environment Installation

```
1  # 1. Install ROS 2 environment dependencies
2  sudo apt install ros-humble-desktop-full
3
4  # 2. Install MoveIt 2 dependencies
5  sudo apt install ros-humble-moveit-*
6
7  # 3. Install camera calibration program
8  sudo apt install ros-humble-camera-calibration
9
10 # 4. Install serial port driver
11 pip3 install pyserial
12
13 # 5. Build all packages
14 cd ~/marm_ws && colcon build
```

# 3. Preparation: Calibration Procedures

[!tip] Calibration Order

It is recommended to complete calibration in the following order: **Arm calibration** → **Camera intrinsic calibration** → **Hand-eye calibration**. Each step affects the accuracy of subsequent steps.

## 3.1 Robotic Arm Zero-Position Calibration

Due to assembly tolerances, the robotic arm may have angular errors at joints. Zero-position calibration is required to avoid grasping deviations caused by inconsistencies between actual and commanded poses.

### Steps:

1. Open a terminal and launch the arm SDK and Movelt control program respectively:

```
1  ros2 run arm_controller arm_controller
2  ros2 launch A1_moveit_config demo.launch.py
```

2. Observe the initial posture of the arm after power-on. Under normal conditions, it should be perfectly vertical:



3. If the arm has angular deviation, open the configuration file and modify the servo offset parameters:



4. Modify the offset values for the corresponding joints in `servo_offsets` (adjust incrementally), then save and recompile:

```

aidlux@aidlux:~$ cd marm_ws/
aidlux@aidlux:~/marm_ws$ colcon build --packages-select arm_controller
WARNING: Package name "A1_description" does not follow the naming conventions. It should start with a lower case letter
and only contain lower case letters, digits, underscores, and dashes.
WARNING: Package name "A1_moveit_config" does not follow the naming conventions. It should start with a lower case lett
er and only contain lower case letters, digits, underscores, and dashes.
Starting >>> arm_controller
Finished <<< arm_controller [2.40s]

Summary: 1 package finished [2.64s]

```

```

1 | servo_offsets:
2 |   servo_0: 0.0 # joint1
3 |   servo_1: 1.0 # joint2
4 |   servo_2: 0.0 # joint3
5 |   servo_3: 0.0 # joint4
6 |   servo_4: 0.0 # joint5
7 |   servo_5: 0.0 # joint6

```

5. Recompile and launch `arm_controller`, then check whether the arm is vertical:

```

1 | cd ~/marm_ws && colcon build
2 | ros2 run arm_controller arm_controller

```

## 3.2 Camera Intrinsic Calibration

[!warning] Important Notice

Intrinsic parameters vary significantly across different USB cameras. **Recalibration is required every time the camera is replaced**, otherwise 3D localization accuracy will be severely degraded.

**Steps:**

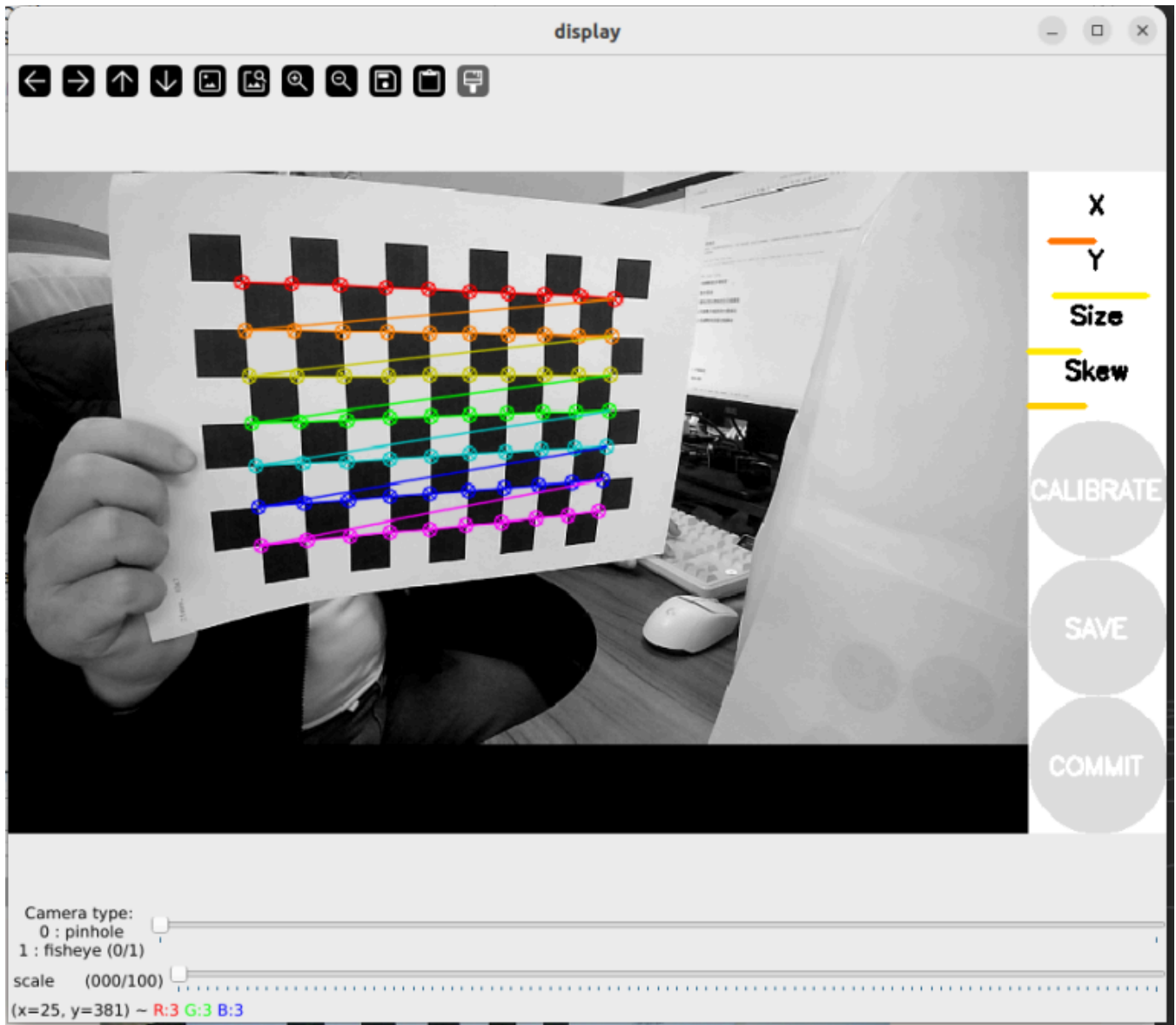
1. Launch the camera driver and calibration program:

```

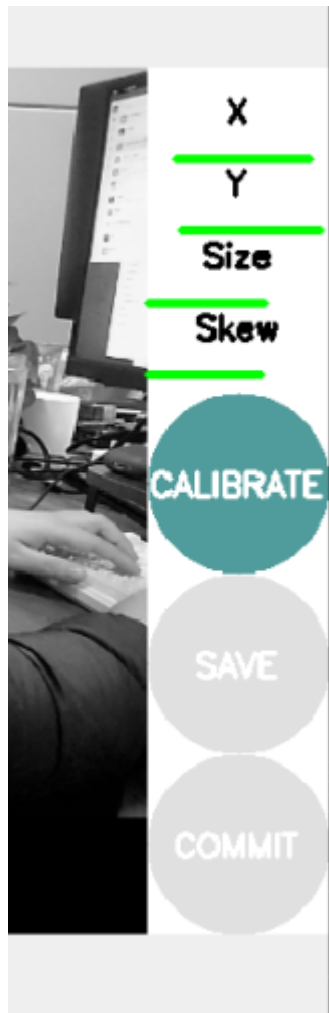
1 | ros2 launch capture_image capture_image.launch.py
2 | ros2 run camera_calibration cameracalibrator \
3 |   --size 10x7 --square 0.021 \
4 |   image:=/camera/image_raw camera:=/camera \
5 |   --no-service-check

```

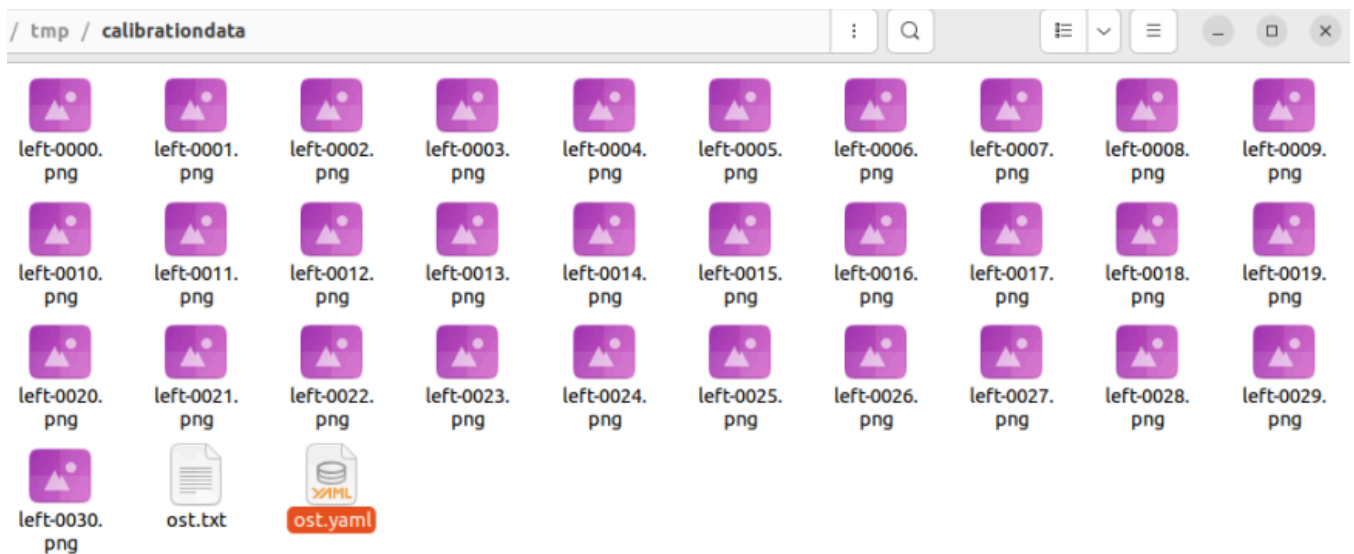
2. A calibration window will pop up after launch:



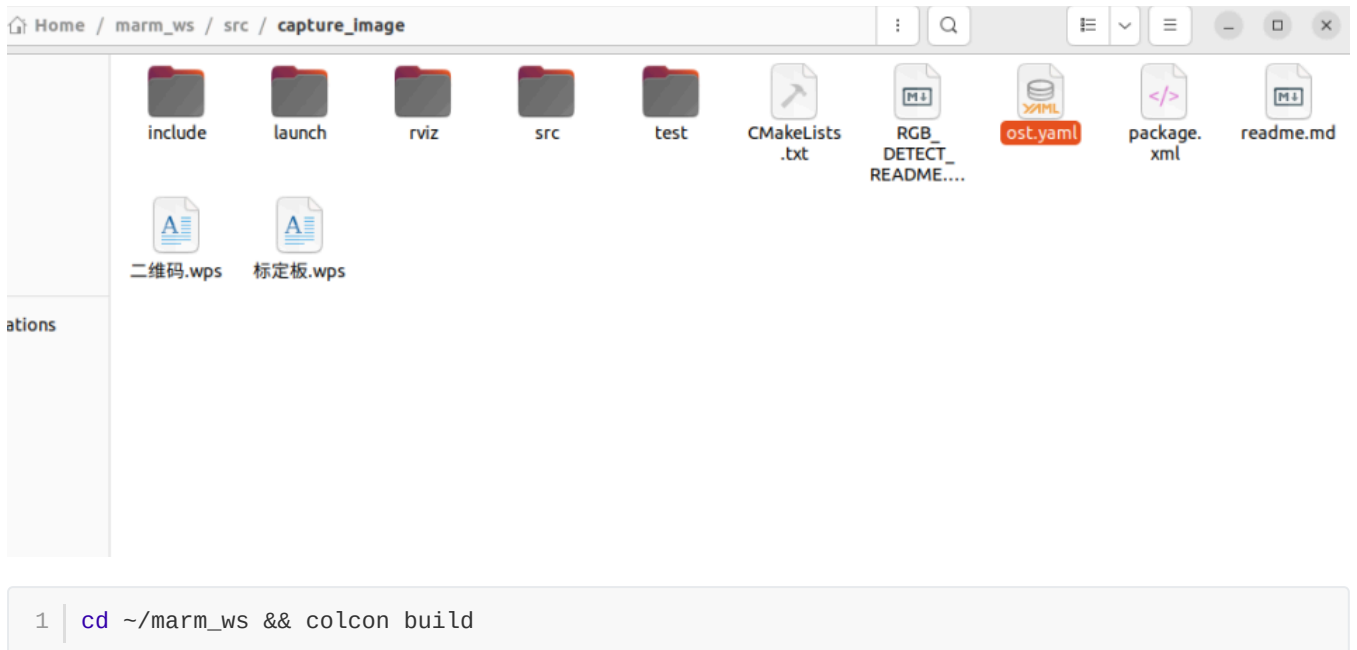
3. Move the calibration board in front of the camera at various angles and orientations until all indicators (X, Y, Size, Skew) in the right panel turn green:



- Click the **Calibrate** button, wait for the calibration computation to complete, then click **Save** to save the results. Calibration data is saved at `/tmp/calibrationdata.tar.gz`:



- Copy the `ost.yaml` file from the archive to the `capture_image` package directory, then recompile:



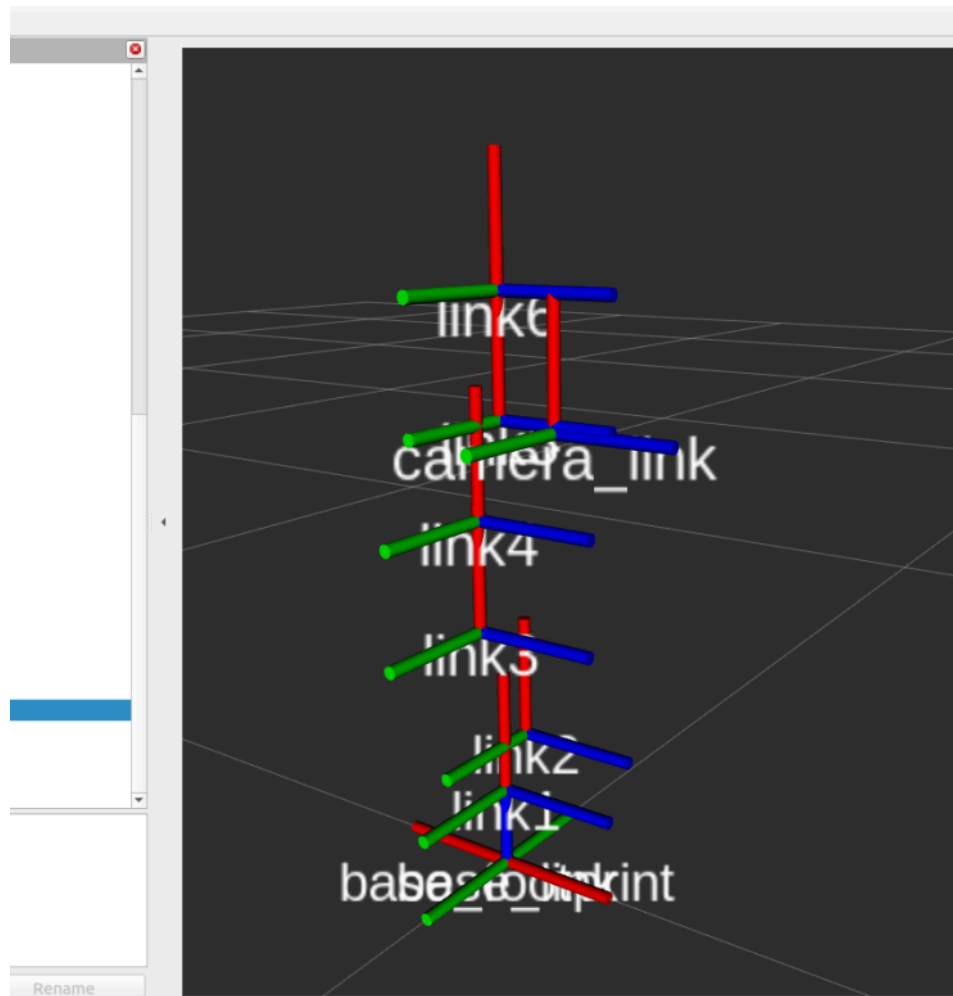
### 3.3 Hand-Eye Calibration (Eye-in-Hand)

Hand-eye calibration determines the transformation relationship between the camera coordinate frame and the robotic arm's end-effector coordinate frame. It is the key to visual grasping accuracy.

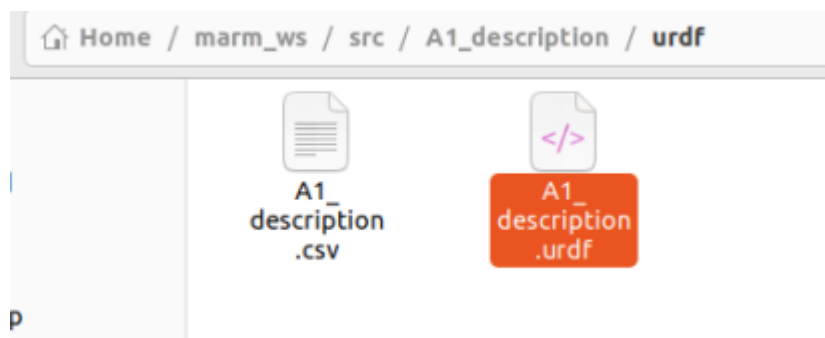
#### Coordinate Frame Relationships:

As shown in the figure below, the camera (`camera_link`) is mounted on the robotic arm's `link5`. The extrinsic transformation matrix between them needs to be calibrated:





The TF tree shows that `camera_link` is fixed under the `link5` coordinate frame:



### Extrinsic Configuration in URDF:

At the end of the robotic arm's URDF file, the extrinsic transformation parameters from `camera_link` to `link5` are defined. The `<link>` element defines URDF display properties, while the `<joint>` element defines the connection relationship and pose offset between the two coordinate frames:

```

19 <link
20   name="camera_link">
21   <inertial>
22     <origin
23       xyz="0 0 0"
24       rpy="0 0 0" />
25     <mass
26       value="0.02" />
27     <inertia
28       ixx="1e-6"
29       ixy="0"
30       ixz="0"
31       iyy="1e-6"
32       iyz="0"
33       izz="1e-6" />
34   </inertial>
35   <visual>
36     <origin
37       xyz="0 0 0"
38       rpy="0 0 0" />
39     <geometry>
40       <box size="0.01 0.03 0.03"/>
41     </geometry>
42     <material name="">
43       <color rgba="0.2 0.2 0.2 1"/>
44     </material>
45   </visual>
46   <collision>
47     <origin
48       xyz="0 0 0"
49       rpy="0 0 0" />
50     <geometry>
51       <box size="0.01 0.03 0.03"/>
52     </geometry>
53   </collision>
54 </link>
55 <joint
56   name="camera_joint"
57   type="fixed">
58   <origin
59     xyz="-0.005 0 0.05"
60     rpy="0 0 0" />
61   <parent
62     link="links" />
63   <child
64     link="camera link" />
65 </joint>
66 </robot>
67

```

XML ▾ Tab Width: 8 ▾ Ln 455, Col 1 ▾ ||

## 4. Core Algorithm Analysis

### 4.1 Color Recognition & Object 3D Localization

Color recognition and 3D localization based on HSV color space are implemented in `rgb_detect_pose_node.cpp`. The overall processing is divided into three stages:

- 1 | Color Extraction → 3D Coordinate Estimation → Coordinate Frame Transformation & Publishing

#### Stage 1: Color Extraction & Target Localization

The program supports three colors: Red (R), Green (G), and Blue (B). It first converts the image from BGR to HSV color space, then extracts a mask using preset thresholds, applies morphological operations for denoising, and finally finds contours and filters for circular targets:

```

1 // 1. Extract target color region using HSV thresholds
2 cv::inRange(hsv_img,

```

```

3   cv::Scalar(threshold.h_min, threshold.s_min, threshold.v_min),
4   cv::Scalar(threshold.h_max, threshold.s_max, threshold.v_max), mask);
5
6   // 2. Morphological denoising: opening to remove small noise, closing to fill holes
7   cv::Mat kernel = cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(5, 5));
8   cv::morphologyEx(mask, mask, cv::MORPH_OPEN, kernel);
9   cv::morphologyEx(mask, mask, cv::MORPH_CLOSE, kernel);
10
11  // 3. Find contours and fit with minimum enclosing circle
12  cv::findContours(mask, contours, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
13  for (const auto& contour : contours) {
14      cv::minEnclosingCircle(contour, center, radius);
15      // Circularity check: circularity > 0.7 to confirm circular target
16      double circularity = 4 * CV_PI * area / (perimeter * perimeter);
17  }

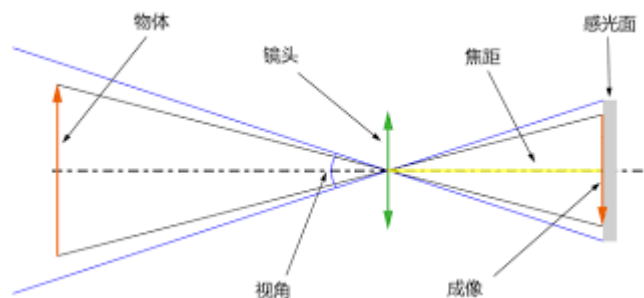
```

[!info] GUI Parameter Tuning

The program provides a GUI slider interface for real-time adjustment of HSV threshold ranges for each color. Adjusted values are automatically saved to a YAML file, enabling quick adaptation to different lighting conditions.

## Stage 2: Estimating 3D Coordinates via Pinhole Camera Model

A monocular camera follows the pinhole camera model — the farther an object is, the smaller its image. The program uses the known real diameter of the object and its pixel diameter in the image to estimate depth:



Core formula:

$$Z = \frac{f_x \times D_{real}}{D_{pixel}}$$

Where  $f_x$  is the camera focal length,  $D_{real}$  is the object's real diameter (preset to 32mm), and  $D_{pixel}$  is the pixel diameter in the image.

The implementation is in the `calculate3DPosition` function:

```

1   cv::Point3d calculate3DPosition(const cv::Point2f& center, double diameter_pixels) {
2       double fx = camera_matrix_.at<double>(0, 0); // Camera focal length X
3       double fy = camera_matrix_.at<double>(1, 1); // Camera focal length Y
4       double cx = camera_matrix_.at<double>(0, 2); // Principal point X
5       double cy = camera_matrix_.at<double>(1, 2); // Principal point Y
6
7       // Core: real diameter × focal length ÷ pixel diameter = depth Z
8       double Z = (fx * circle_real_diameter_) / diameter_pixels;
9
10      // With depth Z known, compute X and Y from pixel offsets

```

```

11     double X = (center.x - cx) * Z / fx;
12     double Y = (center.y - cy) * Z / fy;
13
14     return cv::Point3d(X, Y, Z); // Unit: millimeters
15 }

```

[!tip] Intuitive Understanding

**Larger pixel diameter** → **closer object** → **smaller Z value**; the farther the pixel center is from the image center → the larger the X/Y offset.

## Stage 3: Coordinate Frame Transformation & Publishing

After obtaining the 3D coordinates in the camera frame, the program first publishes them as a TF transform (`camera_frame` → `object`), then uses `tf_buffer->lookupTransform` to listen for the `base_footprint` → `object` transform, converting to the robotic arm's coordinate frame. After workspace filtering, the result is published:

```

1  // Publish TF in camera frame (mm → m)
2  transform.transform.translation.x = position.x / 1000.0;
3  transform.transform.translation.y = position.y / 1000.0;
4  transform.transform.translation.z = position.z / 1000.0;
5  tf_broadcaster->sendTransform(transform);
6
7  // Listen for base_footprint → object to get position in arm frame
8  auto tf = tf_buffer->lookupTransform("base_footprint", "object", ...);
9
10 // Workspace filtering: only publish within valid grasping range
11 if ((x_m > 0.1 && x_m < 0.4) && (y_m > -0.2 && y_m < 0.2) && ...) {
12     pose_msg.header.frame_id = colorTypeToFrameId(color_type); // r→"1", g→"2", b→"3"
13     object_pose_pub->publish(pose_msg); // Publish to object_pose topic
14 }

```

[!note] Color Encoding

`frame_id` encodes the color type numerically (1=Red, 2=Green, 3=Blue). Downstream grasping nodes use this field to distinguish objects of different colors, enabling targeted grasping in multi-object scenarios.

## 4.2 Multi-Angle Adaptive Grasping Planning

Due to the limited workspace of the robotic arm, when the end-effector gripper approaches the same target at different pitch angles, some angles may fail to solve inverse kinematics (IK) due to joint limits or self-collision. The `find_optimal_pitch` function automatically iterates through candidate pitch angles, finds the first plannable pose, and immediately executes it.

### Angle Calculation Logic:

The yaw angle is determined by the object's X and Y coordinates in the `base_footprint` frame, ensuring the gripper always faces the target:

$$\text{yaw} = \text{atan2}(y_{obj}, x_{obj})$$

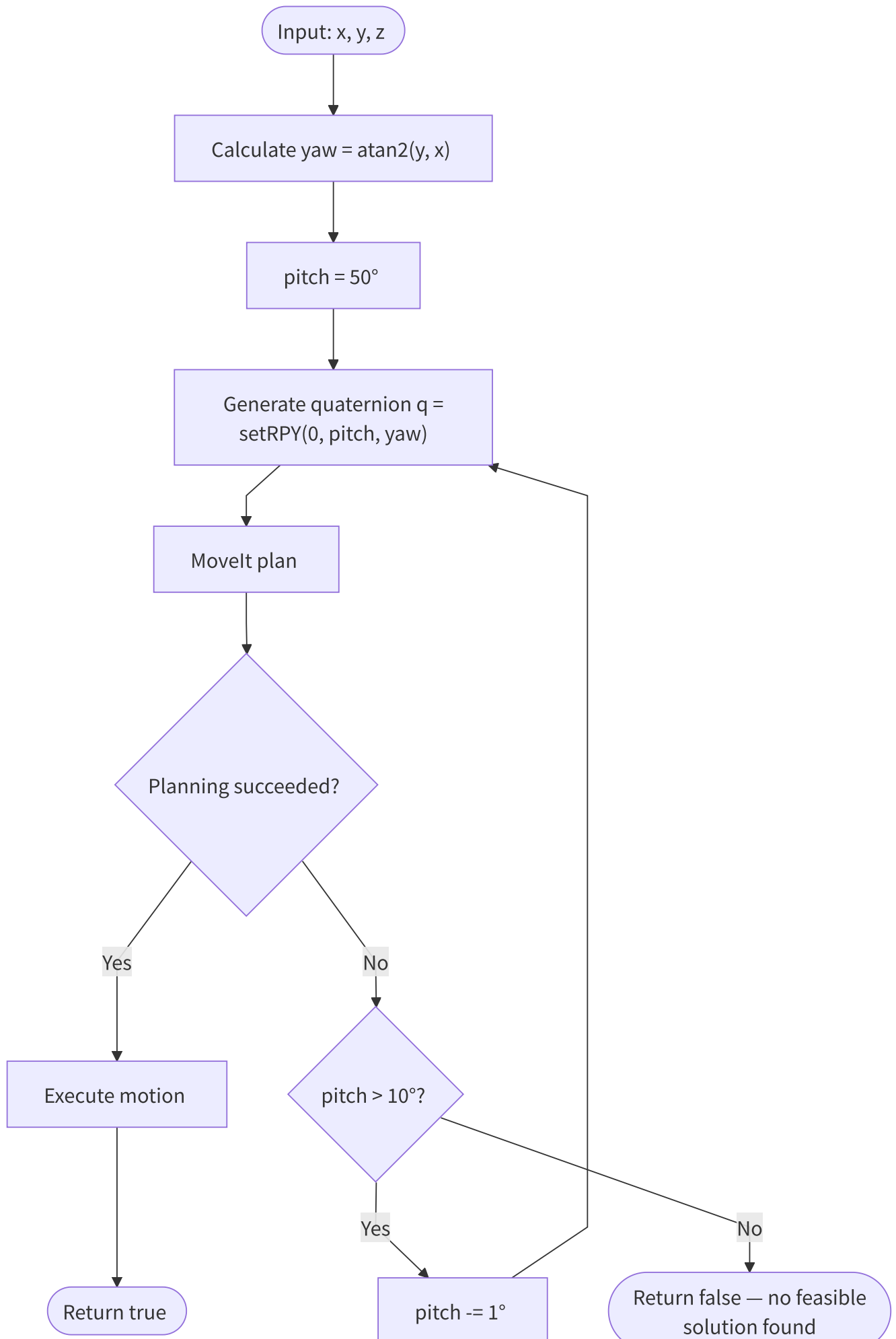
Roll is fixed at 0, and Pitch is iterated from 50° to 10°. `tf2::Quaternion::setRPY` converts Euler angles to quaternions:

$\mathbf{q} = \text{setRPY}(\text{roll} = 0, \text{pitch} \in [50^\circ, 10^\circ], \text{yaw})$

#### Core Code:

```
1 // Calculate yaw angle toward the target
2 double yaw = atan2(obj_y, obj_x);
3
4 // Iterate pitch from 50° → 10°
5 for (int angle = 50; angle >= 10; angle -= 1) {
6     tf2::Quaternion q;
7     q.setRPY(0.0, angle * M_PI / 180.0, yaw);
8     optimal_pose.orientation.x = q.x();
9     // ...
10
11     // Call MoveIt planner
12     auto result = move_group_interface_>plan(plan);
13     if (result == moveit::core::MoveItErrorCode::SUCCESS) {
14         move_group_interface_>execute(plan); // Planning succeeded, execute
15         immediately
16         return true;
17     }
18 }
```

#### Algorithm Flowchart:



## 4.3 State Machine-Based Grasping Workflow

The entire grasping workflow is implemented as a **Finite State Machine (FSM)**, driven periodically in the `timer_callback` timer. The grasping target coordinates are provided by the `object_pose` topic published by the vision node, with offsets from the configuration file applied for fine-tuned compensation.

### Parameter Configuration File

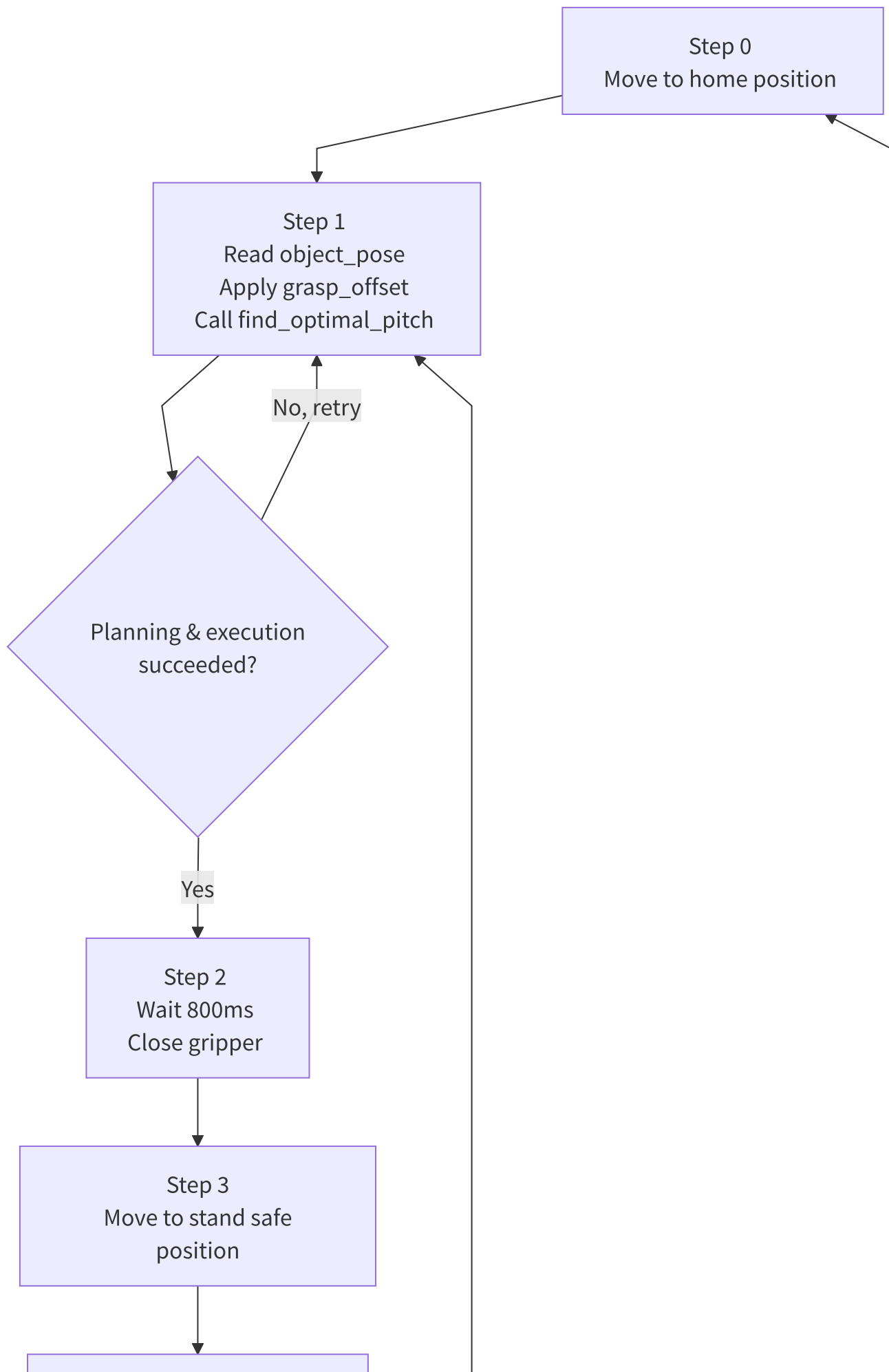
All tunable grasping parameters are centralized in `load_kinematics.yaml`. Changes take effect after recompiling, without modifying any code:

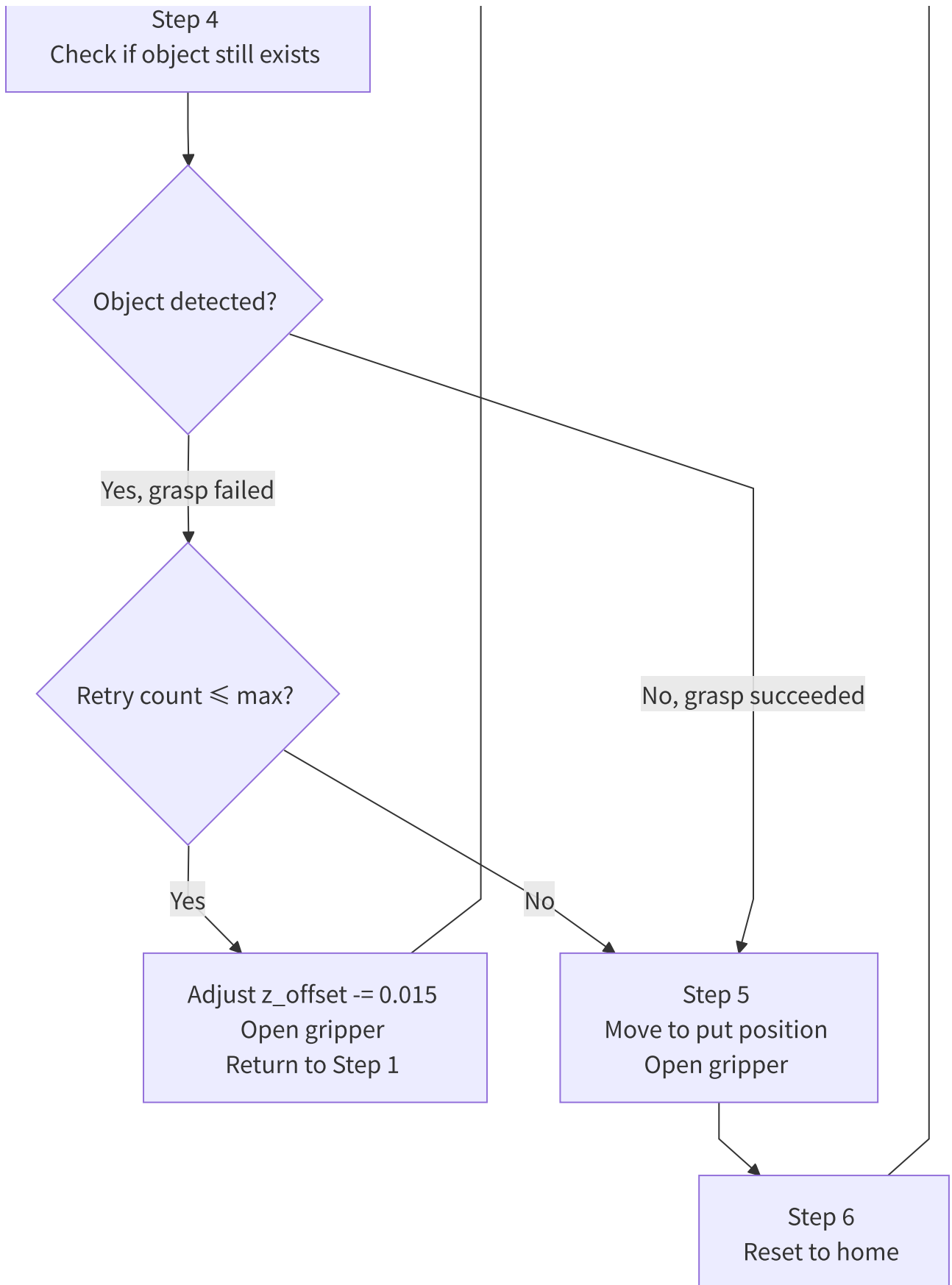
```
1  # Grasping position fine-tuning parameters (unit: meters)
2  # Used to compensate for hand-eye calibration errors or object detection deviations
3  grasp_offset:
4    x: 0.01      # X-axis offset (positive = forward)
5    y: 0.0       # Y-axis offset (positive = left)
6    z: -0.02     # Z-axis offset (positive = up)
7
8  # Grasping retry parameters
9  grasp_retry:
10   enabled: true      # Whether to enable grasping verification and retry
11   max_retries: 3     # Maximum number of retries
12   x_adjust_step: 0.01 # X-axis adjustment per retry
13   detection_timeout: 2.0 # Object detection timeout (seconds)
```

The corresponding loading logic in the program is as follows. Default values are used automatically when the YAML file is missing, ensuring the program does not crash due to configuration issues:

```
1  // Load grasping offset parameters from YAML
2  auto grasp_offset_config = config["grasp_offset"];
3  if (grasp_offset_config) {
4    grasp_offset_x_ = grasp_offset_config["x"].as<double>();
5    grasp_offset_y_ = grasp_offset_config["y"].as<double>();
6    grasp_offset_z_ = grasp_offset_config["z"].as<double>();
7  }
8
9  // Load retry parameters from YAML
10 auto grasp_retry_config = config["grasp_retry"];
11 if (grasp_retry_config) {
12   grasp_retry_enabled_ = grasp_retry_config["enabled"].as<bool>();
13   grasp_max_retries_ = grasp_retry_config["max_retries"].as<int>();
14   grasp_x_adjust_step_ = grasp_retry_config["x_adjust_step"].as<double>();
15   grasp_detection_timeout_ = grasp_retry_config["detection_timeout"].as<double>();
16 }
```

**State Machine Workflow**





**Step 1: Target Point Calculation**

```

1 // Apply hand-eye calibration compensation offsets
2 double obj_x = cached_pose.pose.position.x + grasp_offset_x_ + current_grasp_x_offset_;
3 double obj_y = cached_pose.pose.position.y + grasp_offset_y_;
4 double obj_z = cached_pose.pose.position.z + grasp_offset_z_ + current_grasp_z_offset_;
5
6 // Call multi-angle adaptive planning
7 if (find_optimal_pitch(obj_x, obj_y, obj_z, roll, yaw, target_pose)) {
8     action_trig++; // On success, proceed to Step 2
9 }

```

#### Step 4: Grasp Verification & Retry

```

1 if (is_object_still_detected(grasp_detection_timeout_, grasped_frame_id_)) {
2     // Object still detected → grasp failed, adjust Z-axis offset and retry
3     current_grasp_z_offset_ += -0.015;
4     control_gripper(true); // Open gripper
5     action_trig = 1; // Return to Step 1 for re-grasping
6 } else {
7     // Object not detected → grasp succeeded
8     action_trig++;
9 }

```

[!tip] Parameter Tuning Guide

| Symptom                                    | Parameter to Adjust                        | Direction                                                |
|--------------------------------------------|--------------------------------------------|----------------------------------------------------------|
| Grasping <b>too far forward / backward</b> | <code>grasp_offset.x</code>                | Positive = forward, negative = backward                  |
| Grasping <b>too high / too low</b>         | <code>grasp_offset.z</code>                | Positive = up, negative = down (default -0.02m)          |
| Grasping <b>too far left / right</b>       | <code>grasp_offset.y</code>                | Positive = left, negative = right                        |
| Poor retry performance                     | <code>grasp_retry.x_adjust_step</code>     | Decrease step size, or increase <code>max_retries</code> |
| False positive grasp detection             | <code>grasp_retry.detection_timeout</code> | Increase timeout duration (default 2.0s)                 |

## 5. Operation Guide: Launch Procedures

[!warning] Prerequisites

Before launching in each terminal, make sure to run `source install/setup.bash` to load the ROS 2 workspace environment.



**Description:** Launches the USB camera node and publishes the image data stream.

*Note: If there are no errors or `open camera failed` messages, the program is running successfully. If the camera fails to open, it may be due to a loose USB connection or insufficient permissions — reconnect and restart the program.*

```
aplux@aplux $ ros2 launch capture_image capture_image.launch.py
[INFO] [launch]: All log files can be found below /home/ubuntu20/.ros/log/2026-02-24-15-54-06-813420-ubuntu22-yelu
me-119246
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [capture_image_node-1]: process started with pid [119247]
[capture_image_node-1] image_raw_topic_:/camera/image_raw
[capture_image_node-1] camera_info_topic_:/camera/color/camera_info
[capture_image_node-1] image_rect_topic_:/camera/image_rect
[capture_image_node-1] [INFO] [1771919646.891776754] [capture_image]: Loading calibration from: /home/ubuntu20/01-
program/05-mbot-marm/marm-aplux/marm_aplux_ws/install/capture_image/share/capture_image/ost.yaml
[capture_image_node-1] [INFO] [1771919646.892006879] [capture_image]: Successfully loaded calibration from: /home/
ubuntu20/01-program/05-mbot-marm/marm-aplux/marm_aplux_ws/install/capture_image/share/capture_image/ost.yaml
[capture_image_node-1] [ WARN:0] global ./modules/videoio/src/cap_gstreamer.cpp (2075) handleMessage OpenCV | GStr
eamer warning: Embedded video playback halted; module source reported: Could not read from resource.
[capture_image_node-1] [ WARN:0] global ./modules/videoio/src/cap_gstreamer.cpp (1053) open OpenCV | GStreamer war
ning: unable to start pipeline
[capture_image_node-1] [ WARN:0] global ./modules/videoio/src/cap_gstreamer.cpp (616) isPipelinePlaying OpenCV | G
Streamer warning: GStreamer: pipeline have not been created
1 ros2 run capture_image rgb_detect_pose_node
```

## 5.3 Launch Vision Recognition Node

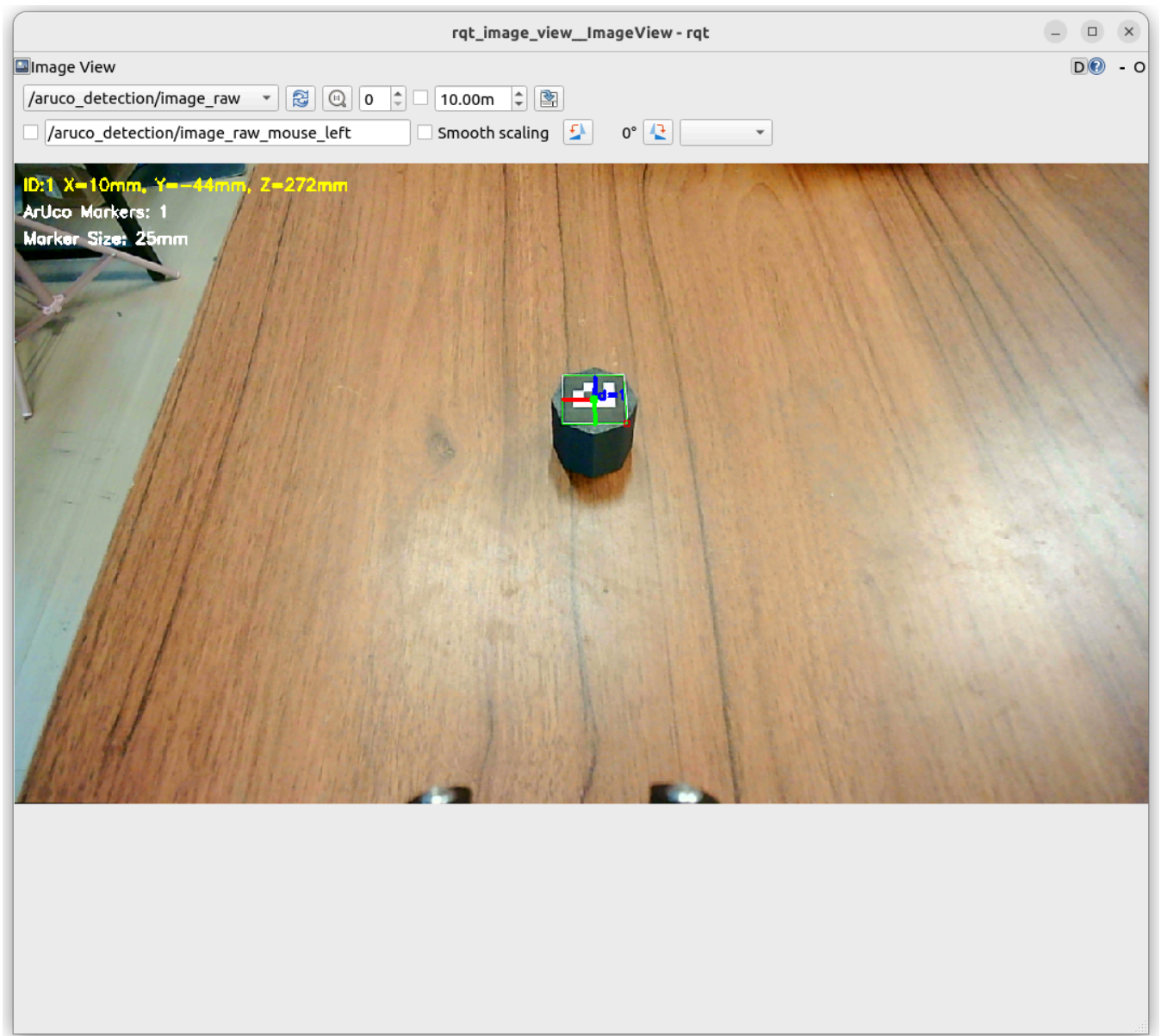
Depending on the task requirements, choose to launch either **ArUco QR code recognition** or **Color recognition** (they cannot run simultaneously).

### Option A: ArUco QR Code Recognition

```
1 | ros2 run capture_image aruco_object_pose_node
```

**Description:** Recognizes ArUco markers and publishes the object's pose in the camera coordinate frame (TF). You can view the recognition results using `rqt_image_view`:

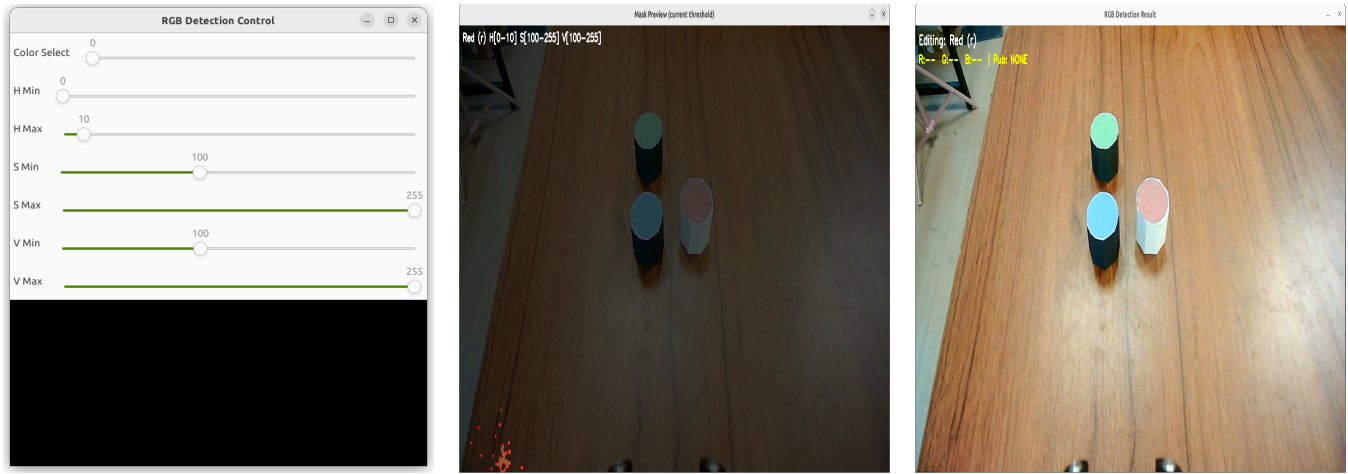
```
1 | ros2 run rqt_image_view rqt_image_view
```



## Option B: RGB Color Recognition

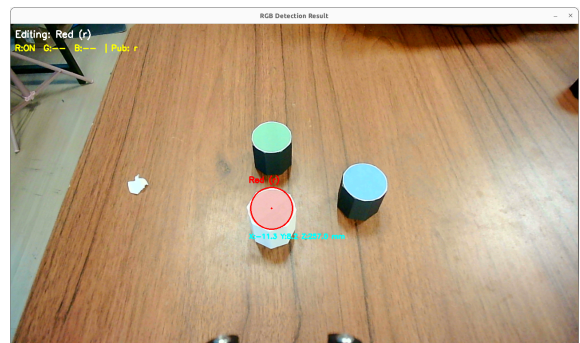
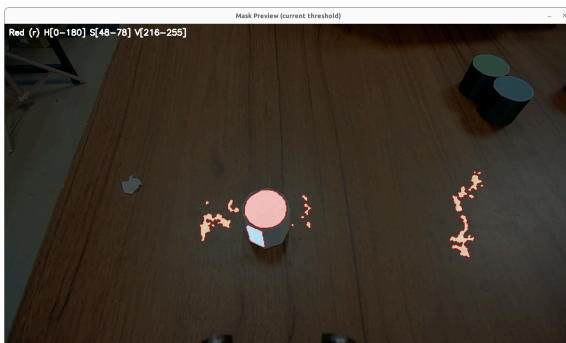
```
1 | ros2 run capture_image rgb_detect_pose_node
```

**Description:** Detects circular objects in three colors (red, green, blue), calculates their 3D positions, and publishes TF. After launch, three windows will appear: color threshold adjustment window, threshold visualization window (Mask Preview), and recognition result window (RGB Detection).

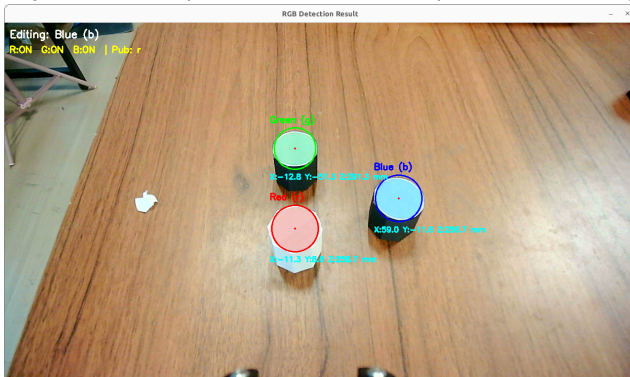


### Color Threshold Adjustment Steps:

1. **Select Color Channel:** In the adjustment window, the `Color Select` slider corresponds to three colors (0=Red, 1=Green, 2=Blue).
2. **Adjust Red Threshold (0):** Set `Color Select` to 0, observe the `Mask Preview` window, and adjust the HSV min-max values until the bright area retains only a single clear circle. The `RGB Detection` window will then display the recognition box and position information.



3. **Adjust Green/Blue Thresholds:** Set `Color Select` to 1 (Green) or 2 (Blue), and repeat the HSV adjustment steps above until a complete circular region appears.



## 5.4 Launch MoveIt2 Motion Planning

```
1 | ros2 launch A1_moveit_config demo.launch.py
```

**Description:** Launches the MoveIt2 core framework, loads the robot URDF model and semantic configuration (SRDF), and opens the RViz visualization interface.

## 5.5 Launch MoveIt Control Server

```
1 | ros2 run moveit_controller_server moveit_controller_server_node
```

**Description:** Provides a custom service interface for MoveIt action control, receives motion commands from upper-level business logic, and calls the MoveIt planner to execute trajectories.

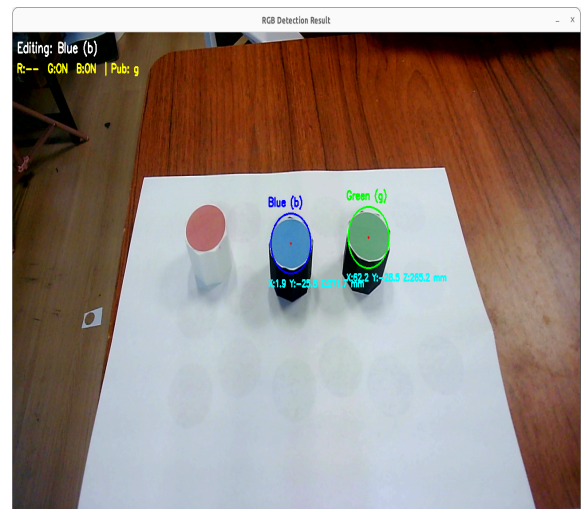
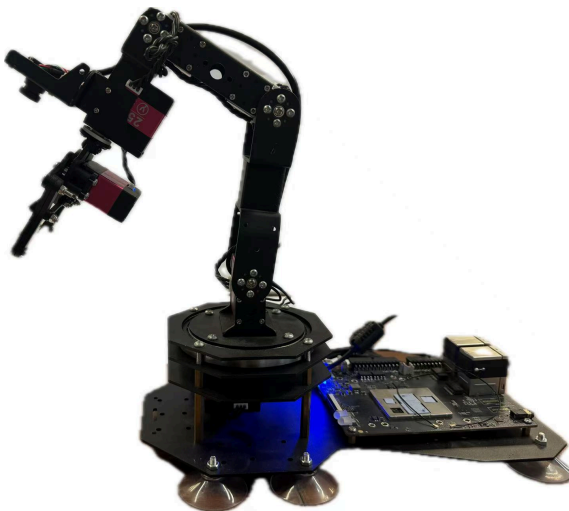
## 5.6 Launch Automated Process Control Node

```
1 | ros2 run moveit_controller_server marm_demo
```

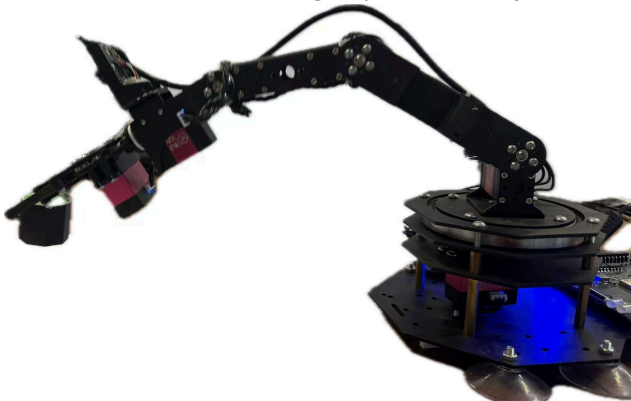
**Description:** Executes the predefined automated grasping workflow, implementing the complete task control logic.

### Grasping State Machine Workflow:

1. **Initial Detection:** The robot moves to the `stand` observation position and checks whether target objects (ArUco or specific colors) are present in the field of view.



2. **Grasping & Re-verification:** After detecting an object, the arm moves above the target to grasp it, then lifts to the `stand` position for secondary recognition. If the object is still detected in the field of view, it is determined as a "missed grasp," and the system will automatically adjust the pose and retry.



3. **Placement & Reset:** After a successful grasp, the object is moved and placed at the preset `put` position. The arm then returns to the `home` position and moves to the `stand` position to prepare for

the next grasping cycle.



#### [!warning] Notes

- **ArUco Grasping:** Currently only supports recognition and grasping of a single block; multi-target simultaneous processing is not supported.
- **Color Grasping:** Supports grasping objects of different colors in a specific order. The default grasping order is: **Red** → **Green** → **Blue**.

## 6. Workspace Directory Structure

```
1 marm_aplux_ws/src/
2 |— A1_description/           # Robotic arm URDF model description package (models,
   mesh files, RViz config)
3 |— A1_moveit_config/         # MoveIt2 configuration package (SRDF, kinematics,
   controller config, Launch files)
4 |— arm_controller/           # Arm low-level control package (serial communication,
   servo control, gripper control)
5 |— capture_image/            # Vision recognition package (camera driver, ArUco
   recognition, RGB color recognition)
6 |— custom_interface/         # Custom message and service interface package (grasping
   actions, pose services)
7 |— easy_handeye2/            # Hand-eye calibration tool package (supports Eye-in-
   Hand / Eye-on-Base)
8 |— easy_handeye2_msgs/       # Hand-eye calibration custom message package
9 |— moveit_controller_server/ # Motion control and business logic package (MoveIt
   service wrapper, state machine demo)
```

| Package                               | Core Nodes                                                              | Description                                           |
|---------------------------------------|-------------------------------------------------------------------------|-------------------------------------------------------|
| <code>arm_controller</code>           | <code>arm_controller</code>                                             | Joint angles → servo serial commands                  |
| <code>capture_image</code>            | <code>aruco_object_pose_node</code> / <code>rgb_detect_pose_node</code> | Visual recognition and pose publishing                |
| <code>A1_moveit_config</code>         | MoveIt2 framework (with RViz)                                           | Motion planning and visualization                     |
| <code>moveit_controller_server</code> | <code>moveit_controller_server_node</code> / <code>marm_demo</code>     | Motion control service + state machine business logic |
| <code>easy_handeye2</code>            | Calibration GUI                                                         | Hand-eye extrinsic data collection and solving        |