Table of Contents

Product Information

Introduction 1.1

1 Product Introduction 1.2

2 Product Specifications 1.3

Basic Setup

3 User Notes 2.1

4 First Installation 2.2

Functions and Applications

5 Software Development Guide 3.1

- 5.1 Python 3.1.1
  - 1 API Introduction 3.1.1.1
  - 2 Python Examples 3.1.1.2
  - 3 Drag Teaching 3.1.1.3
- 5.2 ROS1 3.1.2
  - 1 Environment Setup 3.1.2.1
  - 2 ROS Basics 3.1.2.2
  - 3 Using Rviz 3.1.2.3
  - 4 Basic Function Examples 3.1.2.4
- 5.3 ROS2 3.1.3
  - 1 Environment Setup 3.1.3.1
  - 2 ROS2 Basics 3.1.3.2
  - 3 Using Rviz2 3.1.3.3
  - 4 Basic Function Examples 3.1.3.4

6 Robot Usage Scenarios 3.2

- 6.1 QR Code Recognition and Grasping 3.2.1
- 6.2 Spatial Movement Case 3.2.2
- 6.3 Multi-Point Movement and Grasping Case 3.2.3
- 6.4 Keyboard typing Case 3.2.4

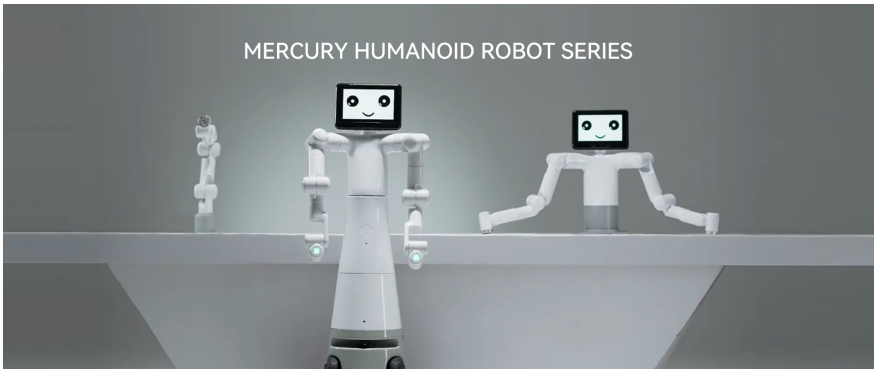
Support and Services

7 About Us 4.1

Acknowledgments

8 Acknowledgments 5.1

Mercury_X1_Gitbook



Welcome to the world of Elephant Robotics' Mercury series, where innovation meets practicality, offering an unprecedented universal hardware platform for embodied intelligence and humanoid robot research. We proudly present three unique product models, each with its exceptional features to meet your diverse needs in robotics research and education.

Mercury X1 Wheeled Humanoid Robot: With 19 degrees of freedom, it combines a high-performance mobile base with advanced dual-arm coordination and perception capabilities, powered by the NVIDIA Jetson edge computing core and 3D machine vision, providing robust computing power and hardware foundation for robotics and embodied intelligence research and development.

Mercury X1 Wheeled Humanoid Robot

Mercury B1 Semi-Humanoid Robot: Featuring 17 degrees of freedom and dual seven-axis robotic arms, equipped with the NVIDIA Jetson edge computing core, 3D machine vision, and voice interaction capabilities, it is the first choice for human-robot interaction research and dual-arm coordination control research.

Mercury B1 Semi-Humanoid Robot

Mercury A1 Seven-Axis Robotic Arm: Ideal for educators, this 7-degree-of-freedom lightweight harmonic joint robotic arm has a payload of 1kg. Its touchable quick control interface and application software greatly simplify application deployment without complex programming.

Mercury A1 Seven-Axis Robotic Arm

The Mercury series is not only a culmination of robotic technology but also a partner in promoting scientific exploration and educational innovation.

Core Documentation

This document includes comprehensive information from product introduction, detailed technical specifications to user notes and first installation guidance. We will delve into the basic functions of the Mercury X1 Wheeled Humanoid Robot, provide software development guides, and showcase successful application cases to help you effectively integrate the Mercury X1 Wheeled Humanoid Robot into various applications. Additionally, we offer rich support and service information to ensure you get the necessary help when facing any technical challenges.

Document Instructions

Depending on your needs and the professional level of Mercury X1 Wheeled Humanoid Robot application development, you can choose to follow this sequence from start to finish or use it as a standalone reference. You can navigate to any section using the sidebar on the left. The document is divided into the following five main sections:

Product Introduction

The product information section provides a basic overview of the robotic arm, including design intent, main functions, and application scenarios, helping you quickly understand the basic characteristics and usage environment of the product. Additionally, this section will detail application examples and supported extended development, providing necessary development guides and resources. At the end of the section, relevant purchase links and channels will be provided for your convenience.

Basic Setup

This chapter is an important part that every user must read carefully. It covers key notes on product use, transportation, storage, and maintenance, aiming to ensure user safety and efficiency when operating the product. Additionally, this chapter details the responsibilities for product failures or damages caused by not following these guidelines.

Functions and Applications

The functions and applications section details the basic functions of the robotic arm and software usage methods, including system usage instructions and firmware functions. The software development guide provides guidance based on different development environments, such as Python and ROS, supporting technical developers in application expansion. By showcasing successful application cases and providing supporting resources, it offers practical references and necessary support materials for a deeper understanding and use of the product.

Support and Services

The support and services section provides comprehensive troubleshooting guides and post-purchase service information, such as warranty and service terms, helping you quickly resolve issues and ensuring you understand your rights and obligations after purchase. Additionally, the 'About Us' section enhances user understanding of the design and manufacturer of Mercury series products, aiming to build trust and brand loyalty.

Acknowledgments

We greatly appreciate your time reading the Mercury X1 Wheeled Humanoid Robot user manual. We hope this document helps you better understand and effectively use this robot, inspiring your creativity. If you have any questions or need further assistance, please feel free to contact our customer support team. We look forward to seeing you complete innovative projects with the Mercury X1 Wheeled Humanoid Robot and welcome you to join our rapidly growing developer community.

Table of Contents

Product Information

- 1 Product Introduction
- 2 Product Specifications

Basic Setup

- 3 User Notes
- 4 First Installation

Functions and Applications

- 5 Software Development Guide
  - 5.1 Python
    - 1 API Introduction
    - 2 Python Examples
    - 3 Drag Teaching
  - 5.2 ROS1
    - 1 Environment Setup
    - 2 ROS Basics
    - 3 Using Rviz
    - 4 Basic Function Examples
  - 5.3 ROS2
    - 1 Environment Setup
    - 2 ROS2 Basics
    - 3 Using Rviz2
    - 4 Basic Function Examples
- 6 Robot Usage Scenarios
  - 6.1 QR Code Recognition and Grasping
  - 6.2 Spatial Movement Case
  - 6.3 Multi-Point Movement and Grasping Case

Support and Services

- 7 About Us

Acknowledgments

- 8 Acknowledgments

1 Product Overview

Product Introduction

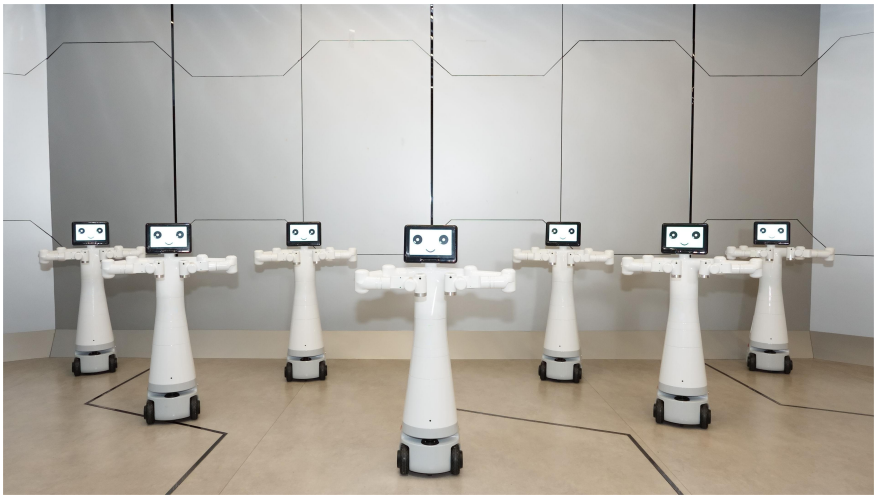
The Mercury X1 is a wheeled dual-arm humanoid robot designed for embodied intelligence applications. It features 19 degrees of freedom and is composed of the Mercury B1 dual-arm robot and a high-performance mobile base. The entire machine is equipped with the newly developed Liyuan series harmonic modules. The new modular design includes electromagnetic brake modules and hollow wiring. It has characteristics such as high precision, high inertia, and low weight. It is also equipped with the NVIDIA Jetson Xavier main control, working in coordination with four independent sub-controls. The mobile base is equipped with high-performance LiDAR, ultrasonic sensors, and 2D vision for rich perception; it uses direct drive motors with a maximum running speed of 1.2m/s; a maximum climbing height of 2CM; and a maximum climbing angle of 15 degrees. The entire machine has a maximum battery life of up to 8 hours, meeting the needs of personal and commercial applications.

Feature	Description
Name	Mercury X1
Type	Wheeled dual-arm humanoid robot
Degrees of Freedom	19 degrees of freedom
Composition	Mercury B1 dual-arm robot + high-performance mobile base
Battery Life	Up to 8 hours
Applications	Research, education, service, entertainment, remote operation

Design Philosophy

The design philosophy of the Mercury X1 wheeled humanoid robot stems from a deep understanding of the needs in the current fields of technology education and research. The rapid development of high-tech fields such as machine learning and artificial intelligence requires experimental equipment to have not only general functionality but also flexibility and precision. The design aims to meet these complex needs, providing a solution that is both flexible and precise and can adapt to various application scenarios.

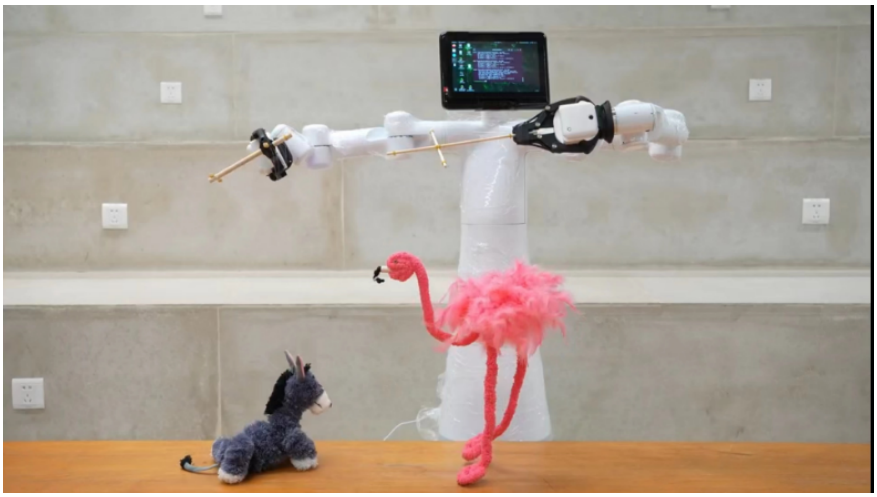
Component	Function
Harmonic Module	Provides high precision, high inertia, low weight
Hollow Wiring	Facilitates maintenance and upgrades
NVIDIA Jetson Xavier	Main control working in coordination with sub-controls
LiDAR, Ultrasonic Sensors, 2D Vision	Rich perception capabilities
Direct Drive Motor	High-performance drive, maximum speed 1.2m/s



Design Goals

Design Goal	Description	Application Scenarios and Features
General Multi-Functional Platform	The Mercury X1 wheeled humanoid robot is suitable for various application scenarios such as education, research, and commercial display.	Its multiple degrees of freedom and mobile navigation support complex motion control in various work environments, such as precise positioning and path planning.
Support for Research and Education	The Mercury X1 wheeled humanoid robot is particularly suitable for machine learning and artificial intelligence research, capable of performing high-precision experiments and technical demonstrations.	Supports end-to-end data training and reproduction, as well as basic visual tasks, making it an ideal device in laboratories.
Programmability and Expandability	The highly programmable nature of the Mercury X1 wheeled humanoid robot allows users to customize and program according to emerging technologies to meet future technological needs.	Through user-customized programming, the device can achieve optimized operations and experimental results, meeting the ever-changing needs of research and development.

Design Goal	Description	Application Scenarios and Features
Technological Innovation and Knowledge Dissemination	In commercial displays, the Mercury X1 wheeled humanoid robot can serve as a platform to showcase the latest technological achievements, aiming to enhance public understanding and interest in technology and promote the commercialization of technological innovations.	By showcasing and demonstrating the latest technological achievements, it increases public engagement, promotes the dissemination of technological knowledge, and enhances market acceptance of technological products.



Product Features

Feature	Advantage
High-Performance Mobile Base	Strong climbing ability, flexible movement
Long Battery Life	Meets long working hours
Multi-Perception Capability	Strong environmental adaptability
Modular Design	Easy maintenance and upgrades

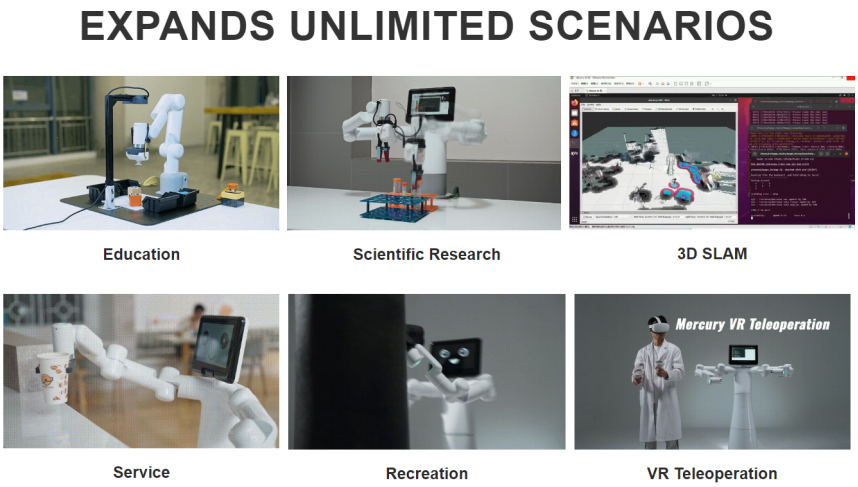
Product Value

Value	Impact
Education and Research	Promotes education and research in the field of intelligent robots
Commercial Service	Enhances interactivity and efficiency in commercial activities
Personal Entertainment	Enriches personal entertainment experience, stimulates creativity

Industry Contribution

Contribution	Description
Research Tool	Provides an experimental platform for humanoid robot research
Educational Resource	Serves as a teaching tool, cultivating talents in robotics technology
Service Innovation	Promotes the intelligent transformation of the service industry

Product Applications



Application Scenario	Function
Research	Dual-arm coordinated control, motion planning
Education	Image recognition, robot control teaching
Service	Exhibition service, company display
Entertainment	Humanoid motion simulation
Remote Operation	VR/Aloha remote control

3. Supported Extended Development

The Mercury X1 wheeled humanoid robot is extremely valuable in the fields of education and research, especially in the widely used development environments of Python and ROS (Robot Operating System). These environments provide strong support, making the Mercury series products widely applicable in machine learning, artificial intelligence research, complex motion control, and visual processing tasks.

Python	- Provides standardized robot driver libraries, available for download and installation via PyPI - Supports querying joint angle data, Cartesian coordinate data, etc. - These features make Mercury suitable for basic education and research scenarios, especially in data collection and preliminary robot control
ROS	- Supports both ROS1 and ROS2 versions, providing RVIZ simulation environment support. - Allows users to display the state information of the robotic arm in real-time, making Mercury suitable for ROS beginners and educational purposes.
Hardware Interface	- Includes IO, USB, etc., facilitating the connection of various sensors and actuators.
Software Libraries	- Provides a rich set of open-source libraries and APIs to simplify the development process.

Next Chapter →

Robot Parameter Description

> In Chapter 1, we explored the product's selling points and design philosophy, providing you with a panoramic view of the product's high-level understanding. Now, let's move on to Chapter 2 - Robot Parameter Description. This chapter is crucial for you to understand the technical details of the product. Understanding these technical parameters in detail will not only help you fully appreciate the advanced nature and practicality of our product but also ensure that you can effectively utilize these technologies to meet your specific needs.

1 Machine Specifications

Product Model	Mercury X1
Product Height	1.18m
Working Voltage	24V
Degree of Freedom	19DOF
Battery life	8 hours
Maximum Payload	1KG
Net weight	55KG
Repeatability	± 0.05mm
Drive Motor	High performance direct drive motors
Maximum Operating Speed	1.2m/s
Maximum Climbing Angle	15°
Storage Space	15L
Main Controller GPU	6-Core Arm v8.2 64-bit CPU，384-Core Volta™ GPU
Computing Power	21 TOPS
Main Controller CPU	4-Core ARM Cortex-A57 CPU
Mobile Base Controller	128 NVIDIA CUDA® cores
Screen	9-inch Touching Screen
Mobile Sensors	LIDAR, Ultrasonic Radar, 2D Vision
3D Camera	Orbbec Deeyea
Microphone	Line array 4 microphones，5m 180° pickup
IO	24V 6 Input, 6 Output
Communication	CAN Bus/WIFI/Network port/Bluetooth/USB serial port

2 Basic Software Function Support

Function/Development Environment	Usage
Free Movement	Supported
Joint Movement	Supported
Cartesian Movement	Supported
Trajectory Recording	Supported
Wireless Control	Supported
Emergency Stop	Supported
Windows	Supported
Linux	Supported
MAC	Supported
ROS 1	Supported
Python	Supported
C++	Supported
C#	Supported
JavaScript	Supported
myblockly	Supported
Arduino	Supported
mystudio	Supported
Serial Control Protocol	Supported
TCP/IP	Supported
MODBUS	Supported

3 Core Control Parameters

Main Controller Specifications

Indicator	Parameter
Main Control	Jetson Xavier
Main Control Model	Jetson Xavier NX
CPU	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6MB L2 + 4MB L3
GPU	384-core NVIDIA Volta™ GPU with 48 Tensor cores
AI Performance	21 TOPS
Storage	16 GB eMMC 5.1
CSI Camera	2 CSI cameras
Network	10/100/1000 BASE-T Ethernet
USB Interface	1 USB 3.2 2.0 (10 Gbps) 2 USB 2.0 interfaces
Other I/O	2 UART serial ports

Left & Right Arm Sub Controller Specifications

Indicator	Parameter
Sub Control	Left & Right Arm Sub Control
Sub Control Model	ESP32
Core Parameters	240MHz dual core. 600 DMIPS, 520KB SRAM. Wi-Fi, dual mode Bluetooth
Sub-Control Flash	4MB
LED Display	5X5 RGB

4 Structural Dimensions

> This chapter uses millimeters as the unit of distance and degrees as the unit of angle.

Product Dimensions and Workspace

When selecting the installation location for the robot, the cylindrical space directly above and below the robot must be considered. Avoid moving tools into the cylindrical space as much as possible. This is because it can cause the joints to rotate too quickly when the tool moves slowly, resulting in low work efficiency and difficult risk assessment.

Dual Arm End Flange Dimensions

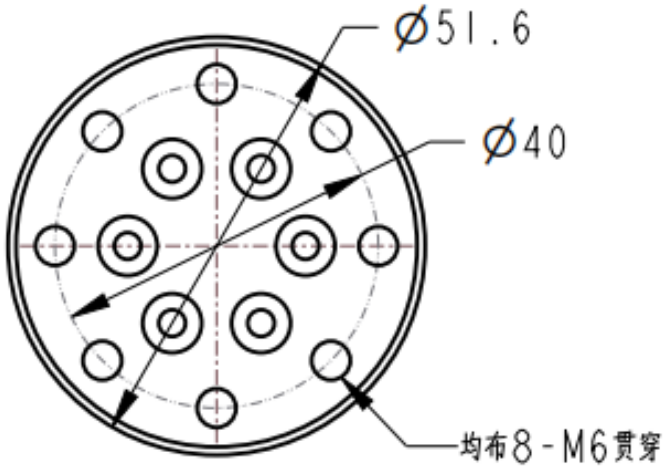


Figure 2.4.1 End Dimensions

DH value

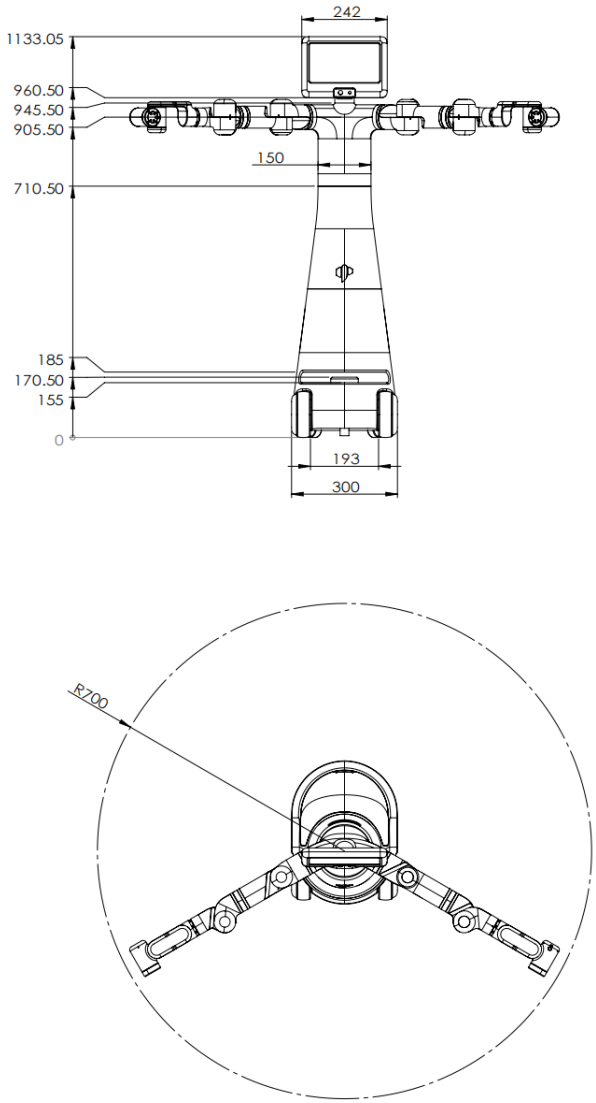


Figure2.4.2 DH value

Arm section

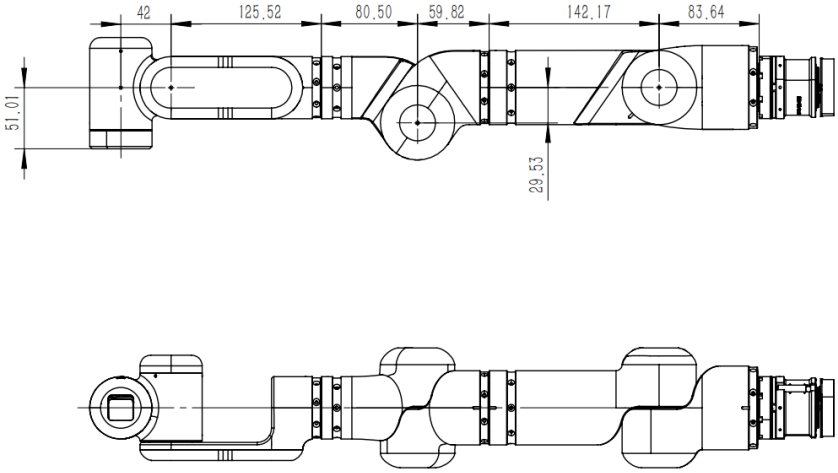


Figure2.4.3 Arm section

5 Electrical Characteristics

Chassis Interface Overview

Figure 1 Chassis Front View

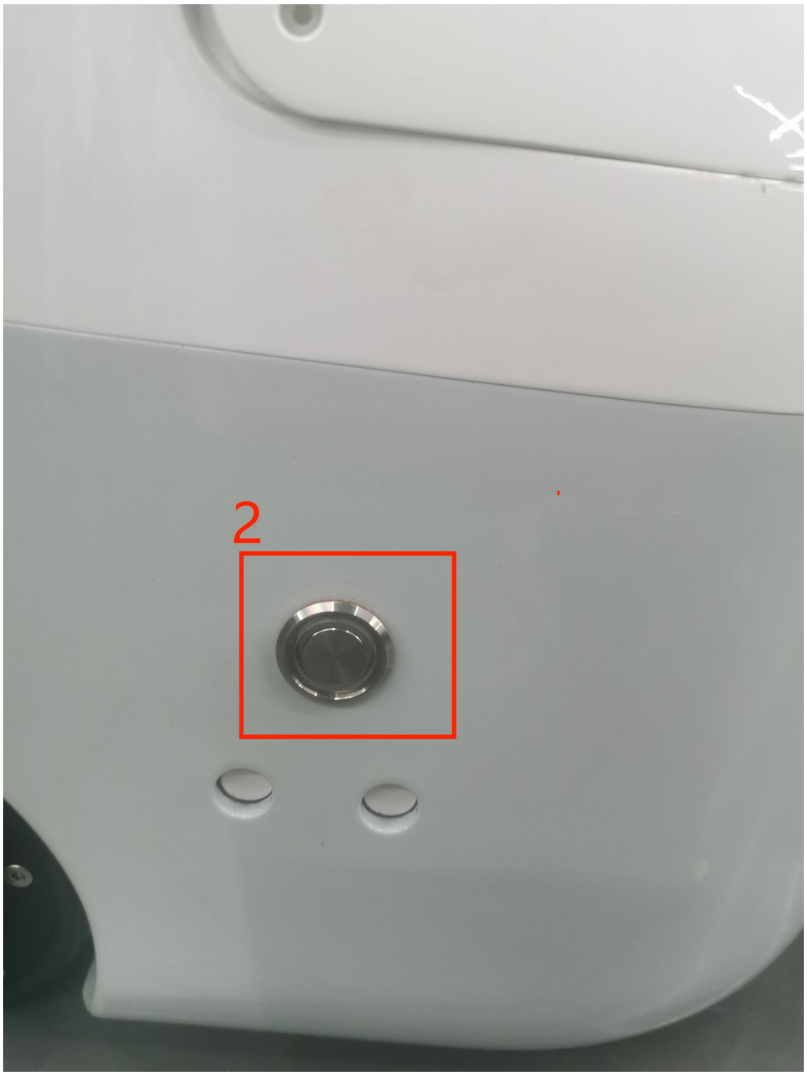


Figure 2 Chassis Left View

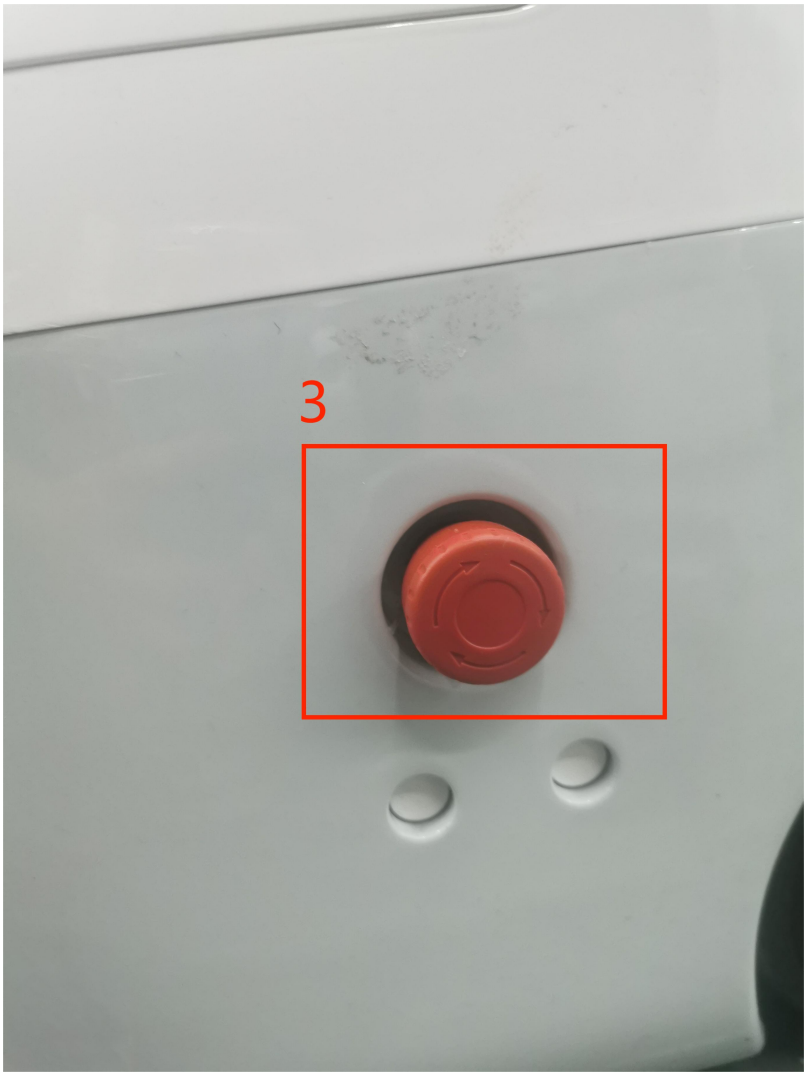


Figure 3 Chassis Right View

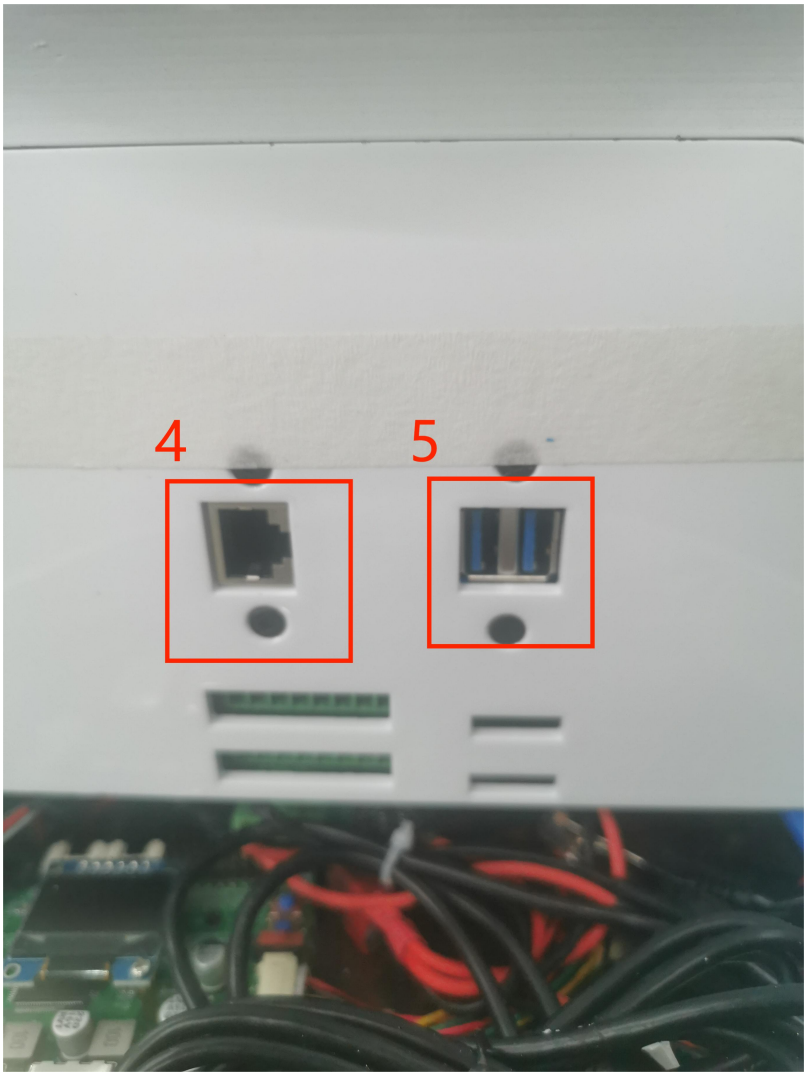


Figure 4 Chassis Internal View

Chassis Interface Description

Number	Interface	Definition	Function	Remarks
1	Power Input Interface	DC24V Input	DC24V Input	
2	Switch	Power Switch	Control input power on/off	With light (lights up when powered)
3	Emergency Stop Interface	STOP	Emergency stop circuit interface	
4	Ethernet Port			
5	USB3.0	USB3.0*2	Can connect external devices or USB drives	

1 Power Input Interface: This interface is connected to the DC24V power adapter interface

2 Power Switch: Controls the on/off of the main power input, when turned off, the controller is also powered off

3 Emergency Stop Circuit Terminal: Connected to the emergency stop button box, can be used to control the robot emergency stop

> **Note**: The emergency stop switch must be connected during robot use, and ensure that the emergency stop switch circuit is in a connected state.

4 Ethernet

5 USB3.0 Interface: A data connection interface using the serial bus standard 3.0; users can use the USB interface to copy program files, or connect peripherals such as a mouse and keyboard

End Interface Overview

Figure 5 Left Arm End View

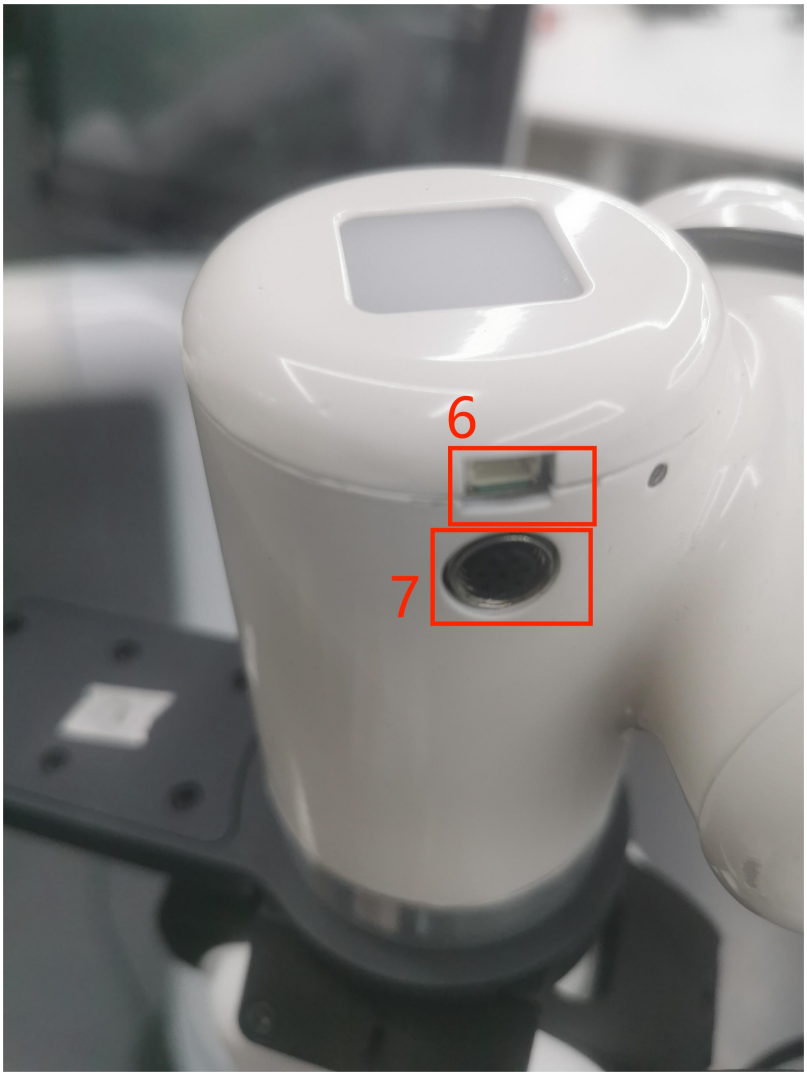
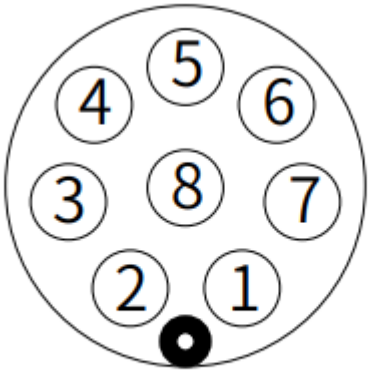


Figure 6 Right Arm End View

End Interface Description

Number	Interface	Definition	Function	Remarks
6	4-pin USB Terminal	External Interface	Connects to camera	
7	M8 Aviation Socket	End Tool IO Interface	Interacts with external devices	

As shown in the figure, the M8 aviation socket I/O diagram, the Mercury X1 robot provides one input and two outputs.



FRONT VIEW

The definitions of each tool I/O port are shown in the table below. Note that the tool I/O, whether input or output, is of PNP type, and the wiring method is the same as the bottom output interface.

Number	Signal	Explanation	Matching M8 Wire Color
1	GND	DC24V Negative	White
2	OUT1	Tool Output Interface 1	Brown
3	OUT2	Tool Output Interface 2	Green
4	485A	Reserved, Not Developed	Yellow
5	24V	DC24V Positive	Gray
6	IN1	Tool Input Interface 1	Pink
7	IN2	Tool Input Interface 2	Blue
8	485B	Reserved, Not Developed	Purple

USB Terminal: Used to connect to the camera

If you have read all the content of this chapter, you can continue to the next chapter.

← Previous Chapter | Next Chapter →

User Instructions

> This chapter is an important part that every user of this product must read carefully. It covers key information about product use, transportation, storage, and maintenance, aiming to ensure user safety and efficiency when operating the product. Additionally, this chapter details the responsibilities for product failures or damages caused by not following these guidelines.

1 Safety Instructions

Overview

This chapter provides general safety information for personnel involved in the installation, maintenance, and repair of the Elephant Robot. Please read and understand the contents and precautions of this chapter carefully before handling, installing, or using the product.

Hazard Identification

The safety of collaborative robots is based on the correct configuration and use of the robot. Additionally, even if all safety instructions are followed, operators may still cause injury or damage. Therefore, it is very important to understand the safety risks of using the robot to prevent them.

Tables 1-1 to 1-3 list common safety risks that may occur during the use of the robot:

Table 1-1 Risk Levels and Safety Risks

1 Personal injury or robot damage caused by improper operation of the robot.
2 If the robot is not fixed as required, such as missing screws or loose screws, or if the base's locking ability is insufficient to support the robot's high-speed movement, the robot will tip over, causing personal injury or robot damage.
3 The robot's safety functions fail to work due to incorrect safety function configuration or lack of safety protection tools.

Table 1-2 Safety Risk Tips

警告
1 Do not stay within the robot's movement range when debugging programs. Improper safety configuration may not avoid collisions, causing personal injury.
2 The connection between the robot and other equipment may bring new dangers, requiring a comprehensive risk assessment.
3 Be careful of scratches and punctures caused by sharp surfaces such as other equipment in the working environment or the robot's end effector.
4 Robots are precision machines; stepping on them may cause damage. Improper placement during transportation may cause vibrations, affecting internal parts and causing damage. Therefore, ensure stability and mechanical integrity in all situations.
5 If the clamped object is not removed before the robot is powered off (when the clamping is not secure), it may cause damage to the end effector or the clamped object may fall and cause injury due to power failure.
6 There is a risk of accidental movement of the robot. Do not stand under any axis of the robot under any circumstances!
7 Compared to ordinary mechanical equipment, robots have more degrees of freedom and a larger range of motion. Failure to stay within the range of motion may cause accidental collisions.

Table 1-3 Potential Electrical Shock Hazards

1 Using non-original cables may pose unknown dangers.
2 Electrical equipment coming into contact with liquids may cause leakage hazards.
3 Incorrect electrical connections may cause electric shock.
4 Be sure to turn off the power of the controller and related equipment and unplug the power plug before replacement. Operating with power on may cause electric shock or malfunction.

Safety Precautions

Follow these safety rules when using the robotic arm:

- The robotic arm is an electrical device. Non-professionals should not change the circuit at will, otherwise, it may cause damage to the equipment or the human body.
- When operating the robotic arm, follow local laws and regulations. The safety precautions and dangers, warnings, and cautions described in this manual are only supplementary to local safety regulations.
- Use the robotic arm in the specified environment. Exceeding the specifications and load conditions of the robotic arm will shorten the product's service life and even damage the equipment.
- Under no circumstances should personnel installing, operating, and maintaining the Mercury arm be strictly trained on safety precautions and the correct methods of operating and maintaining the robot.
- Under no circumstances should this product be used in a humid environment for a long time. This product is a precision electronic component, and long-term exposure to a humid environment will damage the equipment.
- Under no circumstances should this product be used in a humid environment for a long time. This product is a precision electronic component, and long-term exposure to a humid environment will damage the equipment.
- High-corrosion cleaning is not suitable for cleaning the robotic arm, and anodized parts are not suitable for immersion cleaning.
- Unknowingly, do not use the equipment without installing the base to avoid damaging the equipment or causing accidents. The equipment should be used in a fixed environment without obstacles.
- Do not use other power adapters to power the device. If the equipment is damaged due to the use of non-standard adapters, after-sales service is not included.

- Do not disassemble, dismantle, or unscrew the screws or casing of the robotic arm. If disassembled by oneself, warranty service is not provided.
- Untrained personnel should not repair faulty products and disassemble the robotic arm without authorization. If the product fails, please contact Mercury technical support engineers in time.
- If the product is discarded, comply with relevant laws and properly dispose of industrial waste to protect the environment.
- Children should use the equipment at some point, forcing someone to monitor the process and turn it off after completion.
- Do not reach into the movement range of the robot arm when the robot is moving to avoid collisions.
- It is strictly forbidden to change, remove, or modify the nameplate, instructions, icons, and signs of the robotic arm and related equipment.
- Handle and install with care. Place the robot gently according to the instructions on the packing box and place it correctly according to the direction of the arrow. Otherwise, the machine may be damaged.

- **Do not burn other product drivers from the Atom terminal or use non-officially recommended firmware. If the equipment is damaged due to the user burning other firmware, it is not covered by after-sales service.**

- Power specifications: **Use official power supply**
- USB Type-C usage specifications: **Do not connect to the power board**

If you have any questions or suggestions about the content of this manual, please log in to the Elephant Robotics official website and submit relevant information:

https://www.elephantrobotics.com

Do not use the robotic arm for the following purposes:

- Medical care costs in life-critical applications.
- Buying a bus may cause an environmental explosion.
- Lent directly without risk assessment.
- Costs of using low-level safety functions.
- Lo-fi does not meet the performance parameters of robot use.

Disclaimer

Please read and understand the following disclaimer before using the product:

- **Safe Use:** This product is designed for specific application scenarios. Ensure that all safety guidelines and operating manuals are followed during use. Users should receive appropriate training on product use and understand and comply with all relevant safety regulations.

- **Limitation of Liability:** The manufacturer is not responsible for any direct, indirect, incidental, special, or consequential damages resulting from the use or misuse of the product or any matters related to the product. This disclaimer does not cover or exclude liabilities that cannot be excluded by law.

- **Technical Support:** Please read the product documentation carefully during installation and use, and seek technical support from the manufacturer if necessary. For technical support issues, refer to the official documentation provided by the manufacturer or contact the relevant support channels.
- **Software Updates:** The manufacturer may provide updates to the product firmware or software. Users should regularly check and apply these updates to ensure product performance and safety.
- **Regular Maintenance:** Users should inspect and maintain the product according to the regular maintenance guidelines provided by the manufacturer. Regular maintenance and inspections help ensure the long-term performance of the product.
- **Customization and Modification:** Do not customize, modify, or alter the product without explicit permission from the manufacturer. Any unauthorized modifications may void the product warranty and may have unpredictable effects on safety and performance.
- **Legal Compliance:** Users should ensure that their use complies with all applicable laws and regulations. In some regions, the use of the product may be subject to specific regulations.

By using the Mercury X1 Wheeled Dual-Arm Humanoid Robot, you agree to and accept these disclaimers. The manufacturer reserves the right to change product specifications, functions, and disclaimers without notice.

2 Transportation and Storage

Logistics Transportation Requirements

Temperature	0°C~50°C
Relative Humidity	20%~70%
Direction During Transportation	Robot head up, arms down
External Conditions During Transportation	Fixed with wooden frame to prevent squeezing
Wooden Frame Size	57*57*120

Internal Packaging

External Packaging

2 Equipment Storage

Temperature	0°C~50°C
Relative Humidity	20%~70%
Direction During Transportation	Robot head up, arms down
Stacking Requirements	Cannot be stacked
Storage Environment	Indoor
Other Environmental Requirements	- Keep away from dust, oil smoke, salt, iron filings, etc. - Keep away from flammable, corrosive liquids and gases. - Must not come into contact with water. - Avoid transmitting shocks and vibrations. - Keep away from strong electromagnetic interference sources.

3 Maintenance and Care

As a robot manufacturer, we emphasize ensuring that customers can properly and safely maintain and upgrade their robotic equipment. To this end, we provide the following detailed maintenance and care guidelines, including common maintenance items and parts for repair or upgrade. Please read carefully.

Common Maintenance Items and Recommended Cycles

Maintenance Item	Description	Recommended Cycle
Visual Inspection	Check for obvious damage, foreign object accumulation, or wear on the robot	Daily
Structural Cleaning	Clean the robot's structural components with a clean, dry cloth, avoiding moisture and corrosive cleaners	Daily
Fastener Inspection	Check and tighten all bolts and connectors	Daily
Lubrication	Lubricate joints and moving parts with manufacturer-recommended lubricants	Every 3 months
Cable and Wiring Inspection	Check cables and wiring for damage or wear	Monthly
Electrical Connection Inspection	Ensure all electrical connections are secure, free of corrosion or damage	Monthly
Software Updates	Check and update control software and applications	Whenever updates are available
Software Data Backup	Regularly back up key software configurations and data	Quarterly
Firmware Updates	Regularly check and update firmware to get the latest features and security patches	Whenever updates are available
Sensor and Device Inspection	Check sensors and other key devices to ensure they are working properly	Monthly
Emergency Stop Function Test	Regularly test the emergency stop function to ensure its reliability	Monthly
Environmental Condition Monitoring	Monitor the working environment's temperature, humidity, dust, etc., to ensure it meets the robot's operating specifications	Continuous monitoring

Maintenance Item	Description	Recommended Cycle
Safety Configuration Review	Regularly check and confirm the robot's safety configurations, such as speed limits and working range settings	Monthly
Preventive Maintenance Plan Execution	Perform regular inspections and maintenance according to the manufacturer's maintenance plan	As per manufacturer's guidelines

Guidelines for Independently Changing Robot Hardware

We understand that customers may have the need to upgrade or repair robot hardware on their own. Before performing any upgrade operations, please read the relevant product parameters in detail and confirm with our official personnel whether such operations are allowed. Unauthorized operations may cause product failure and are not covered by the warranty.

Material Requirements

Officially manufactured or recommended materials: All parts and components required for repairs and upgrades must be officially manufactured or explicitly recommended by us. This includes but is not limited to electronic components, sensors, motors, connectors, and any other replaceable parts.
Material acquisition: Customers can purchase the required repair and upgrade materials through our official channels. This ensures the quality and compatibility of the parts.

Repair or Upgrade Process

Customer self-repair: Customers are responsible for completing the repair work. We will provide detailed repair guides and manuals to guide customers through the repair steps.
Follow official guidance: Repair operations should strictly follow the official guidance provided by us. Any deviation from official guidance may cause equipment damage.

Responsibility and Warranty Policy

Responsibility Division:

Manufacturer: Provide official guidance for repairs and upgrades, officially manufactured or recommended materials, and handle issues caused by manufacturing defects.

Customer: Responsible for completing repairs according to official guidance and using official parts.

Warranty Policy:

Warranty valid: The warranty is valid only if the repair operations fully comply with our guidance and use official parts.
Warranty invalid: If the customer does not follow official guidance or uses non-official parts for repairs or upgrades, any damage caused will not be covered by the warranty.

Precautions

Safety first: Before performing any repair or upgrade operations, ensure that all safety guidelines are followed, including power-off and the use of appropriate protective equipment.
Technical support: If you encounter problems during the repair process, it is recommended to stop the operation and contact our technical support team for assistance.
We strongly recommend that customers strictly follow these guidelines to ensure the safe and effective operation of the robotic equipment. Improper repair operations may cause equipment damage and affect the warranty status. If you need further guidance or support, please contact our professional technical team in time.

4 Frequently Asked Questions

How to Ask Questions Gracefully

1 When asking questions in various places, you may encounter the following phenomena:

- No response after asking a question.
- The question is answered after a long time.
- The other party always complains that you are too inexperienced.

2 Before asking a question, make sure you have studied this manual.

Many questions will be resolved in this process. Avoid asking questions in QQ groups, forums, issues, or emails right from the start. Many questions that are explained in the documentation may not receive timely responses from the community. To save everyone's time and create a better community environment, please understand each other and grow together.

3 When asking questions, try to do the following, which will greatly increase the chances of quickly resolving the issue:

Make sure to clarify what happened, what I did, including:

- What effect or function do I want to achieve?
- How did I do it to achieve this effect, what was the specific process?
- What errors occurred during the implementation, what were the phenomena (for example, what was the reported error, what was the **complete** error content)?
- Did I carefully read the error message, did the error message indicate the cause and solution?
- Based on these error messages, can I solve the problem by thinking carefully?
- Can I find a solution to the problem by searching the documentation, issues, and using search engines?

4 If I really can't solve the problem myself and need help, consider:

- Who to ask, where to ask, who is more likely to answer my question? And how timely is it?
- What data and phenomena should I provide to make them willing to quickly help me solve the problem?
    - Provide my purpose (to let the responder know what you are doing)
    - Provide the complete implementation process and the phenomena that occurred during the process (to allow the responder to follow your process and reproduce the problem)
    - Point out where the error occurred, indicating where the phenomenon or result differs from my expectations! (Let the responder know where it did not meet expectations)
    - Provide the error message, which needs to be complete, with as many screenshots and logs as possible. Don't be stingy with a small screenshot or only provide part of the log (because the responder may not have done this for a long time and may have forgotten some details, they need to quickly recall through screenshots and complete logs; and detailed logs can quickly locate the problem)
- How to ask questions in a way that seems sincere, even if I am inexperienced, everyone is willing to answer

5 Question Template

Try to ask questions gracefully, without adding unnecessary tone words, complaining words, carefully consider each word and punctuation, and think from the responder's perspective on how to quickly help solve the problem. Too few words may not describe the problem clearly, and too many words may make people impatient.

6 Title

No matter where you ask questions (including `QQ groups`), give your question a title of about 30 words, clarifying the central idea of the problem, including:

- The type of problem, whether it is a question, a bug submission, or an experience sharing, etc. Let everyone know what you want to do at a glance on a screen full of text
- Clarify the central idea of the problem in one sentence, such as `Running the camera example program, error reset fail, possibly a hardware issue`

So the comprehensive title can be like this:

- `【Mercury Question】 Running the camera example program, error reset fail, possibly a hardware issue`

Such titles should **not** appear:

- `Ah, why is my board not working again`
- `Why can't my code run`
- `Why is my screen black`
- `【Mercury Question】 Received the development board, the screen is red with a line of small text, why`
- `I ran the xxx program and there was a problem`

You can ask like this:

- `【Mercury Question】 My board cannot start after I connected the power supply in reverse. How can I determine where the board is burned, and if possible, how can I revive it`

7 Content

First, from the responder's perspective, if asked a question:

- First, know what the other party wants to do, what goal to achieve
- What steps did they follow to achieve this goal
- What specific steps were taken, and where did the problem occur, so I can try to reproduce the phenomenon by following their steps. If this problem seems difficult to solve and there are no reproduction steps, it may take a lot of time to reproduce, so I will solve other problems first
- What exactly is the problem, if they only say there is a problem, how do I know what the problem is, maybe they are feeling unwell? So this is very important, they need to explain the phenomenon when the problem occurs and point out how it differs from expectations, otherwise, I have to guess and compare with expectations, which increases the time to solve the problem
- When a problem occurs, I may need their log files to analyze the source code, otherwise, it may be difficult to solve the problem, so this problem can be put aside for later

In summary, you can ask questions like this:

- Clearly state your goal, what you want to achieve, and what the expected phenomenon should be
- Have I referred to any documentation, code, or tutorials

- How to reproduce the error: specifically what was done, write down each step in detail until the problem occurs
- Clearly describe the phenomenon when the error occurs and how it differs from expectations, proving that the problem indeed occurred
- Attach log files, screenshots, and even videos. Logs and screenshots must be complete, do not only provide a small part. The responder may notice some issues you didn't pay attention to from your complete logs and screenshots, which is very important
- Additionally, paste the code with proper formatting, so it is readable and can be run directly
- Finally, express gratitude to the community friends who answered the questions

If you have read all the content of this chapter, you can continue to the next chapter.

Driver Related

1 About Python

Q: What do the parameters in the API send_base_coords([x, y, z, rx, ry, rz], speed) mean? What do rx, ry, and rz correspond to in Euler angles? What is the rotation order of Euler angles? What is the range of each parameter?

- A: The parameters in the array are the coordinates of the Mercury X1 end, and speed is the velocity. rx, ry, and rz correspond to RPY, which are roll, pitch, and yaw respectively. The Euler angle order is zyx, which is its own coordinate. The range for x, y, z is -350 to 350, -350 to 350, -41 to 523.9 (undefined range, if exceeded, an inverse kinematics no solution prompt will be returned). The range for rx, ry, rz is -180 to 180.

Q: Are the Python APIs the same for different versions of the robotic arm?

- A: The API is the same.

2 About ROS

Q: Can you provide the rviz model files and programming examples?

- A: They can be found on our GitHub. "https://github.com/elephantrobotics/mercury_x1_ros"

Q: Why do I get a permission error "/dev/ttyUSB0" when starting the rviz model file using ROS?

- A: This is because the serial port permission is not given. You should type sudo chmod 777 port name in the terminal. For example:

```
sudo chmod 777 /dev/ttyUSB0
```

Q: Why do I get the error _init_() takes exactly 2 arguments (3 given) when running the ROS slider control and model follow commands?

- A: The pymercury library is not installed and started.

Q: Why is the angle of mercury_X1 inconsistent with the model angle after opening the rviz model when using ROS?

- A: It is very likely that the zero position of mercury_X1 is not calibrated, and the zero position of mercury_X1 needs to be calibrated.

Software Issues

About ROS1

Q: When switching to ~/catkin_ws/src in the terminal and using git to install or update mercury_x1_ros, the target path “mercury_x1_ros” already exists. What is the reason?

- A: This means that there is already a “mercury_x1_ros” package in ~/catkin_ws/src. You need to delete it in advance and then re-execute the git operation.

Q: When running rosrun, the terminal error prompts “could not open port /dev/ttyUSB0: Permission: '/dev/ttyUSB0'”. Why?

- A: Insufficient serial port permissions. Enter “sudo chmod 777 /dev/ttyUSB0” in the terminal to grant permissions.

Q: Why can't ROS programs run in VSCode?

- A: Because the VSCode terminal cannot load the ROS environment, you need to run it in the system terminal.

Q: When running rosrun, the terminal prompts “Unable to register with master node [http://localhost:11311]: master may not be running yet. Will keep trying.” What is the reason?

- A: Before running the ROS program, you need to open the ROS node by entering “roscore” in the terminal.

Q: When running rosrun, the terminal error prompts “could not open port /dev/ttyUSB0: No such file or directory: '/dev/ttyUSB1'”. Why?

- A: Serial port error. You need to confirm the actual serial port of the current robotic arm. You can check it with `ls /dev/tty*`.

Q: In Ubuntu 18.04, 'catkin_make' fails to build the code, and the terminal prompts 'Project 'cv_bridge' specifies '/usr/include/opencv' as an include directory, but it does not exist.' and other error messages.

- A: The OpenCV path in the configuration file does not match the actual system path. You need to use the sudo command to modify the configuration file (path is “/opt/ros/melodic/share/cv_bridge/cmake/cv_bridgeConfig.cmake”). The actual OpenCV path of the system is under “/usr/include/”.

Q: Simply cloning the mercury_x1_ros package and then directly running the rosrun program results in errors such as “package 'mercury_x1_ros' not found” or errors like files not found?

- A: The newly cloned mercury_x1_ros needs to build the ROS environment compiled code. Enter in the terminal:

```
bash
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
```

About Robotic Arm Control

Q: The robotic arm does not move when sending angles or coordinates to it.

- A: Use `get_angles()` to read the angles of the robotic arm. If the angles return empty, check if the robotic arm is powered on, use `power_on()` to enable the robotic arm; check if the port number is used correctly. If there are angles, check if the angles exceed the movement range as shown in Table 1. If they exceed the range, use `release_all_servos()` to relax all joints (note that the joints will fall after relaxation, you need to catch them), align the zero scale lines of joints 1~5, 7, and make joint 6 perpendicular to the zero scale line by 90 degrees. Then use `focus_all_servos()` to lock the joints, and use `set_servo_calibration(1)~set_servo_calibration(7)` to calibrate the zero points of each joint in sequence, then use `get_angles()` to read the current joint angles. If the returned data is [0, 0, 0, 0, 0, 90, 0], the calibration is successful, otherwise, repeat the previous calibration steps.

Table 1-1 Joint Angle Movement Range

Joint	Range
J1	-175 ~ +175
J2	-65 ~ +115
J3	-175 ~ +175
J4	-180 ~ +10
J5	-175 ~ +175
J6	-20 ~ + 173
J7	-180 ~ +180

Q: The camera cannot be opened.

- A: Open the local camera to see if it can switch to the left and right arm cameras. If a certain camera cannot be viewed, re-plug the USB port or change the USB port. If it can switch to the left and right cameras, check if the camera port has changed (the port may change after restarting).

Q: The adaptive gripper cannot be controlled.

- A: Check the power indicator of the adaptive gripper. The indicator should be steady on when normal. If the indicator flashes, re-plug the connection cable between the gripper and the robotic arm. If the indicator returns to steady on, it is normal. If the indicator is steady on but still cannot be controlled, use `set_gripper_mode(0)` to change the gripper usage mode.

Q: The base cannot move.

Enter in the terminal:

```
roslaunch turn_on_mercury_robot turn_on_mercury_robot.launch
rostopic echo /PowerVoltage
```

Check the battery level. If the battery level is below 21V, the base cannot be used. Please charge it before continuing to use it.

Hardware Issues

1 About Structure

Q: What is the role of Atom in the robotic arm?

- A: Atom mainly controls the kinematic algorithms of the robotic arm, including forward and inverse kinematics, solution selection, acceleration and deceleration, speed synchronization, multi-power interpolation, coordinate transformation, etc. Programs related to Atom are not yet open source.

Q: Why does the motor automatically cut off the power during use?

- A: The motor has overheat protection for long-term use. This phenomenon is normal and can be used again after a few minutes.

2 About Parameters

Q: What is the unit of speed for the robotic arm?

- A: The running speed is 180 degrees/second.

Q: Can the adaptive gripper not be fully closed?

- A: There will be a certain gap between the jaws, they are not fully closed. You can adjust it by increasing the thickness of the gap between them.

Q: How to fix the USB camera at the end of the robotic arm?

- A: It needs to be fixed with a flange, which can be purchased separately.

If you have purchase intentions or any parameter questions, please send an email to this address. Email: "sales@elephantrobotics.com"

If the listed issues cannot help you solve the problem and you have more after-sales questions, please send an email to this address. Email: "support@elephantrobotics.com"

← Previous Chapter | Next Chapter →

1 Product Standard List

Product Standard Checklist

Serial Number	Product
1	Mercury X1 Wheeled Humanoid Robot
2	Product Manual
3	Certificate of Qualification
4	User Instructions
5	Shipping List
6	Power Supply (24V)

Note: After the packaging box is in place, please confirm that the robot packaging is intact. If there is any damage, please contact the logistics company and the supplier in your area in time. After unpacking, please check the actual contents in the box against the item list.

---

2 Product Unboxing Guide

Product Unboxing Graphic Guide

Why You Need to Follow the Steps Below to Remove the Product

In this section, we strongly recommend following the specified steps to remove the product. This not only helps ensure that the product is not damaged during transportation but also minimizes the risk of accidental failure. Please read the following graphic guide carefully to ensure that your product is safe during the unboxing process.

- **1** Check the box for damage. If there is any damage, please contact the logistics company and the supplier in your area in time.
- **2** Open the packaging box and take out the user instructions, foam packaging cover, Mercury X1 Wheeled Humanoid Robot, matching power supply, and shipping list.
- **3** Before proceeding to the next step, please ensure that each step has been completed to prevent unnecessary damage or omissions.

Note: After removing the product, please carefully check the appearance of each item. Please check the actual items in the box against the item list.

Unboxing Image

3 Gripper Camera Installation Guide

4 Power-On Detection Guide

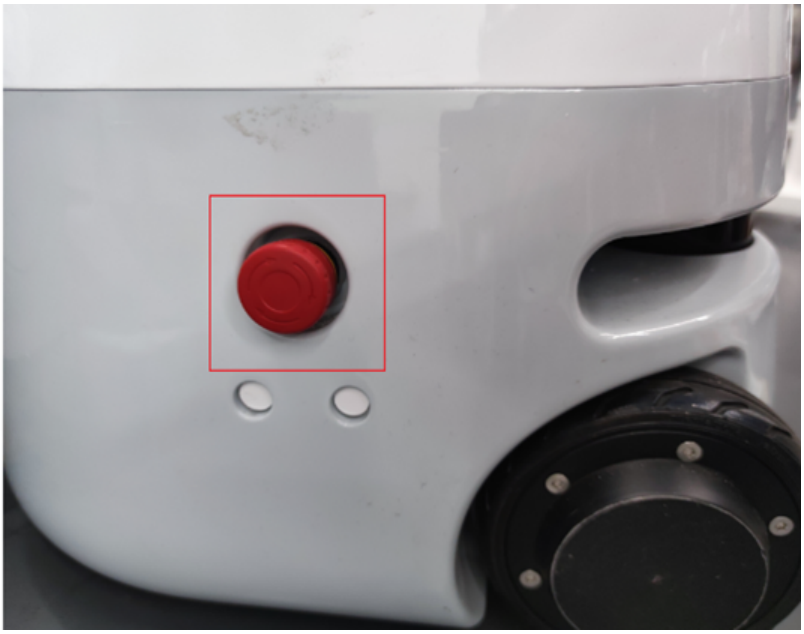
External Cable Connection

Please read the **Chapter Safety Instructions** carefully before operation to ensure safe operation. At the same time, connect the power adapter to the base.

Connect the power adapter to the robotic arm, or power it through the built-in power supply, and move the robot to an open area to prevent collisions.

Power Interface Position at the Rear of the Base

Power Switch Position



Emergency Stop Switch Position

After pressing the emergency stop switch, the robot will stop moving immediately. To reset, turn the emergency stop switch clockwise, and it will pop up automatically. Then you can power on and resume control.

Click the power switch and ensure that the emergency stop is in the reset state to start using the robotic arm.

Bottom LED Light On

Enter the Power-On Interface

Screen On

Enter the login interface and input the power-on password **Elephant**

Basic Function Detection

Check if the Dual Arms Can Move Normally:

1. Open the terminal and enter the Python environment.
2. Import the pymycobot package, initialize the left and right arms, and power on.

3. Use get_angle() to get the joint angles. If the angles can be obtained, the power-on is successful.

4. Calibrate the zero point, and use get_angle() again to get the joint angles. When the return is [0.0, 0.0, 0.0, 0.0, 0.0, 90.0, 0.0], the zero point calibration is successful.

5. Control the left and right arms to see if they can move. If they move successfully, the robotic arm is normal.

```
er@er-desktop:~$ python
Python 3.8.10 (default, Nov 22 2023, 10:22:35)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pymycobot import Mercury
>>> mr = Mercury("/dev/ttyACM0")
>>> ml = Mercury("/dev/ttyTHS0")
>>> ml.power_on()
1
>>> mr.power_on()
1
>>> ml.get_angles()
[-362.31, -98.48, -254.57, 15.59, -31.44, 3.67, 1.29]
>>> mr.get_angles()
[-7.07, -95.34, -111.56, 19.93, -713.43, -71.2, 3.35]
```

Get Joint Angles After Power-On

Check if the Mobile Base Can Move Normally:

1. Enter the following command in the terminal to start the base control:

```
roslaunch turn_on_mercury_robot turn_on_mercury_robot.launch
```

```
er@er-desktop:~$ roslaunch turn_on_mercury_robot turn_on_mercury_robot.launch
... logging to /home/er/.ros/log/dc68a2e6-fe33-11ee-bacd-e8fb1c355a09/roslaunch-e
r-desktop-6122.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://er-desktop:40689/

SUMMARY
========

PARAMETERS
 * /if_akm_yes_or_no: no
 * /mercury_robot/gyro_frame_id: gyro_link
 * /mercury_robot/odom_frame_id: odom
 * /mercury_robot/robot_frame_id: base_footprint
 * /mercury_robot/serial_baud_rate: 115200
 * /mercury_robot/usart_port_name: /dev/wheeltec_con...
 * /robot_description: <?xml version="1....
 * /robot_pose_ekf/base_footprint_frame: base_footprint
 * /robot_pose_ekf/freq: 30.0
 * /robot_pose_ekf/imu_used: True
 * /robot_pose_ekf/odom_data: odom
 * /robot_pose_ekf/odom_used: True
 * /robot_pose_ekf/output_frame: odom
 * /robot_pose_ekf/sensor_timeout: 2.0
 * /robot_pose_ekf/vo_used: False
 * /rosdistro: noetic
 * /rosversion: 1.16.0
```

1. Open a new terminal and enter the following command to start keyboard control of the mobile base. Follow the prompts to control the keys. If the car moves normally, the mobile base is normal.

```
roslaunch mercury_x1_teleop keyboard_teleop.launch
```

Check if the LiDAR is Normal

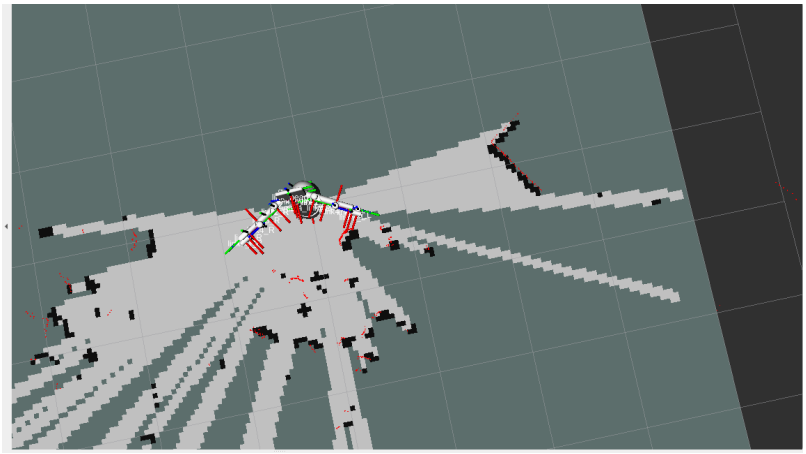
1. Open the terminal and enter:

```
roslaunch turn_on_mercury_robot mapping.launch
```

2. Open a new terminal and enter:

```
roslaunch turn_on_mercury_robot cobotx_rviz.launch
```

If the automatically opened rviz has a white area as shown in the figure, the LiDAR scan is normal.



← Previous Chapter | Next Chapter →

Software Development Guide

1 Usage Environment

The Mercury X1 wheeled humanoid robot is developed and used based on the built-in Ubuntu system.

2 Development Environment

To meet the diverse application needs of the robot in different scenarios, we have adapted the robot to multiple programming languages. So far, we have adapted the following mainstream programming languages, and we believe you can use any of these languages for development. Please be sure to follow the instructions carefully. Any missed steps may result in the corresponding language not running successfully. We wish you success in using the robot.

- Python
  Our robot supports Python, and the development of the Python API library is becoming more and more complete. The robot's joint angles, coordinates, gripper, and other aspects can be controlled through Python.
- ROS1
  ROS (Robot Operating System) is an open-source robot operating system that provides unlimited possibilities for robot development and control. Our robot can be controlled through ROS's rich control functions in a modular way. Whether it is joint control, path planning, or perception feedback, ROS provides corresponding tools and libraries, making the control process more flexible and efficient.
- ROS2
  ROS 2 (Robot Operating System 2) is a flexible software framework designed for robot software development. Our robot can use the hardware abstraction, device drivers, library functions, visualization tools, message passing, and package management services and functions provided by ROS 2, making application development more efficient and modular.

← Previous Chapter | Next Page →

Environment Configuration

> pymycobot is a Python library developed by Elephant Robot for robot control.

Linux

Python 3.8.10 is pre-installed on the system, and the `pymycobot` control library is already installed, so users do not need to install it themselves.

pymycobot Installation

You can install pymycobot by entering the command in the terminal:

```
pip install pymycobot
```

pymycobot Uninstallation

You can uninstall pymycobot by entering the command in the terminal:

```
pip uninstall pymycobot
```

pymycobot Update

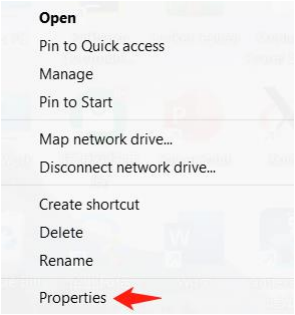
You can update pymycobot by entering the command in the terminal:

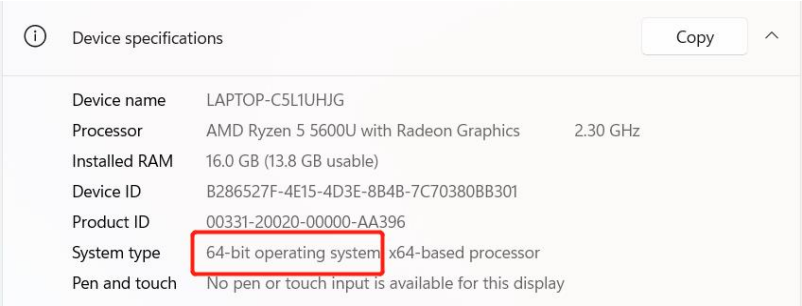
```
pip install pymycobot -U
```

Windows

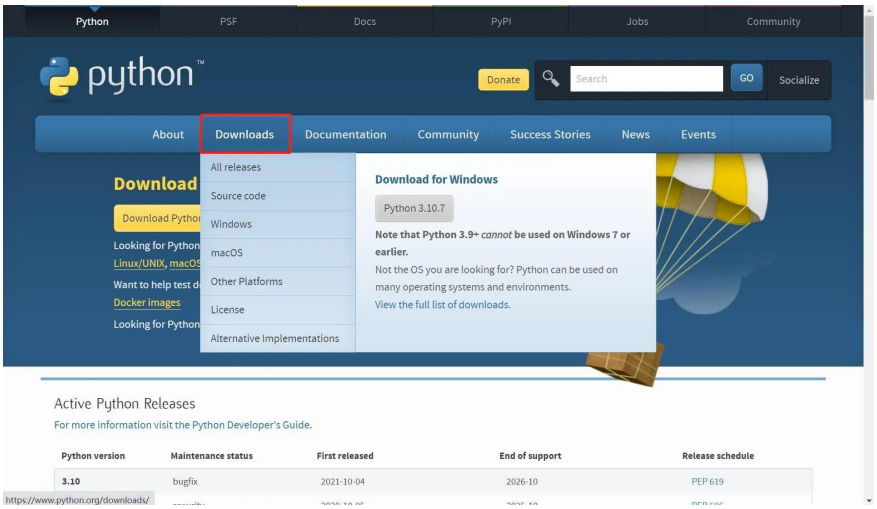
1.1 Installing Python

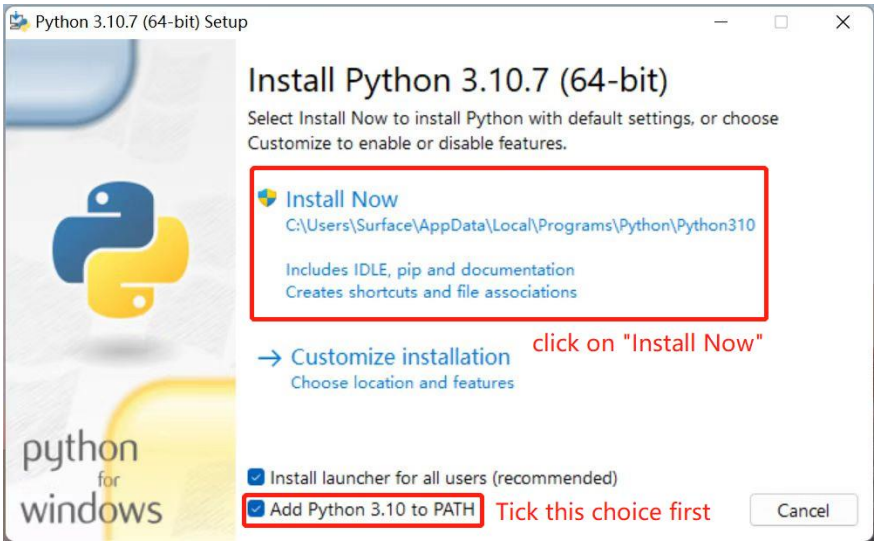
Note: Before installation, check the operating system of your PC. Right-click on the "My Computer" icon and select "Properties". Install the corresponding Python version.

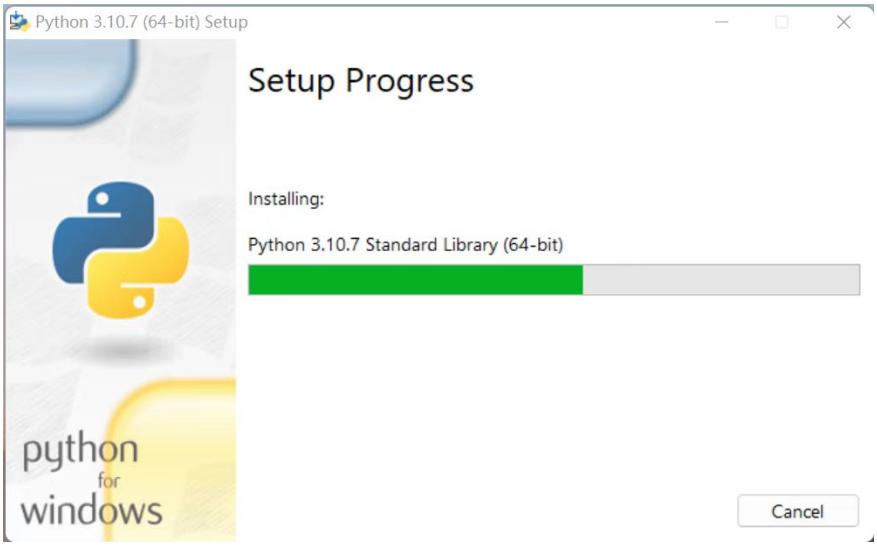




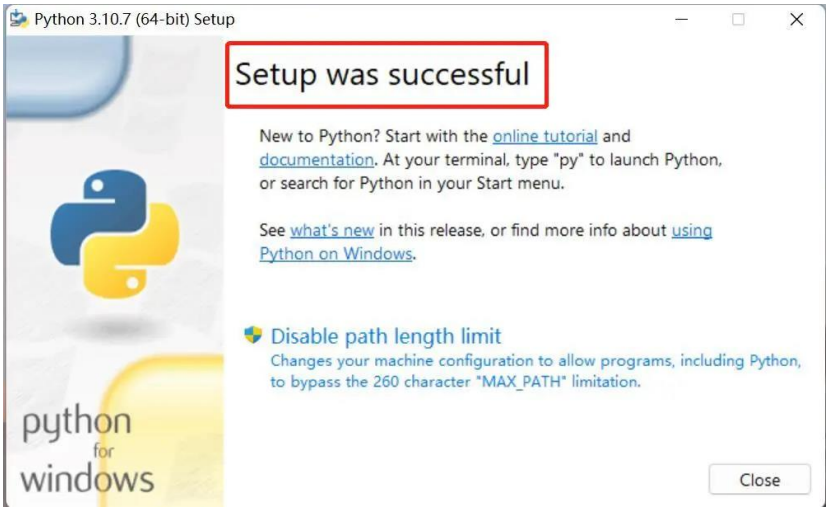
- **Go to http://www.python.org/download/ to download Python.**
- **Click "Download" to start downloading. Check "Add Python 3.10 to PATH". Click "Install Now" to start the installation.**







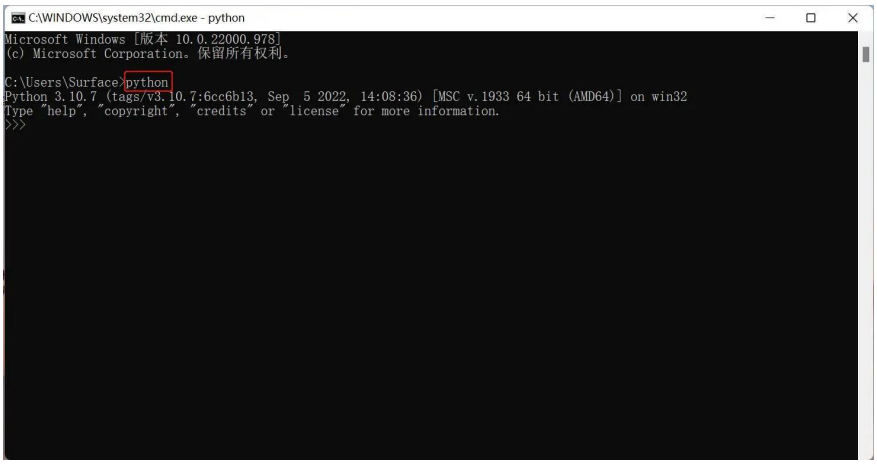
- **Download and installation complete.**



1.2 Running Python

Open the Command Prompt window (Win+R, type "cmd" and press "Enter"). Type "Python".

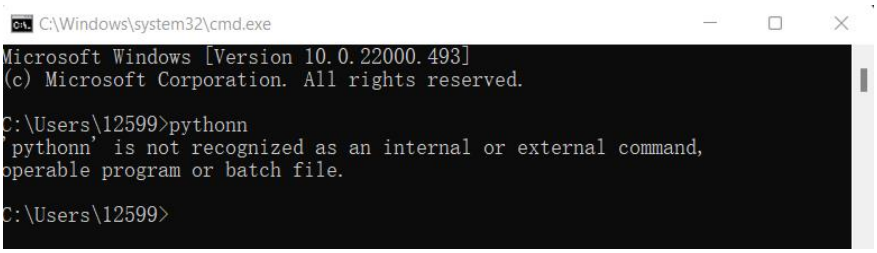
Successful Installation:



This indication on the screen means that Python has been successfully installed. The prompt ">>>" indicates the Python interactive environment. If you enter a piece of Python code, you will get the execution result immediately.

Error Report:

If you enter an incorrect command, such as "pythonn", the system may report an error.



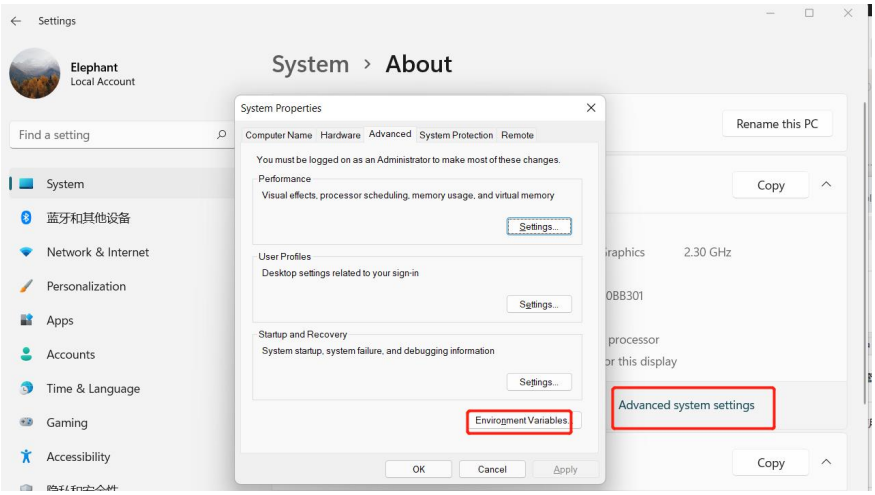
> **Note:** Generally, this error is caused by insufficient environment configuration. Refer to **1.3 Environment Configuration** to solve the problem.

1.3 Environment Variable Configuration

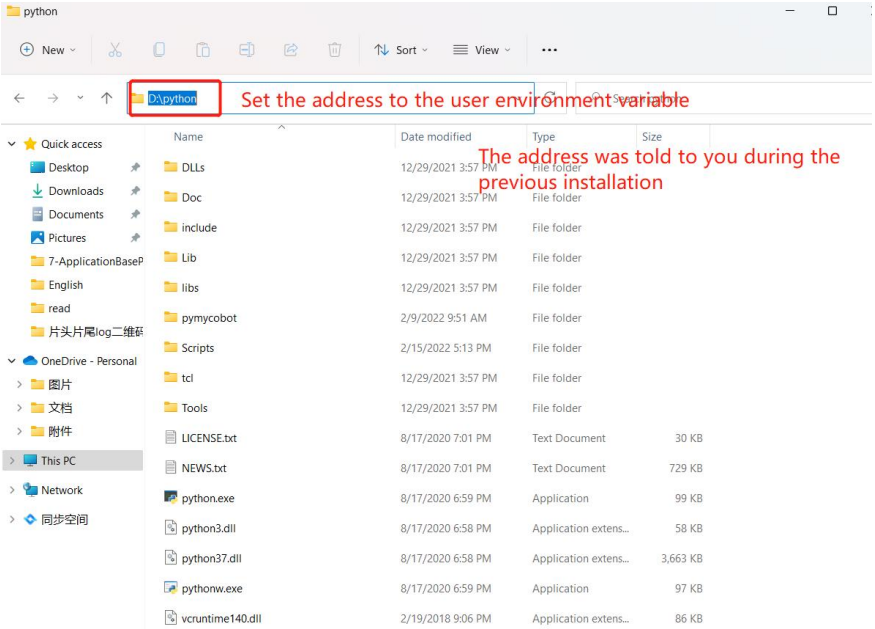
Windows follows the Path environment variable settings to search for **python.exe**. Otherwise, an error will be reported. If you did not check "Add Python 3.9 to PATH" during installation, you need to manually add the path where python.exe is located to the environment variables or re-download Python. Remember to check "Add Python 3.9 to PATH".

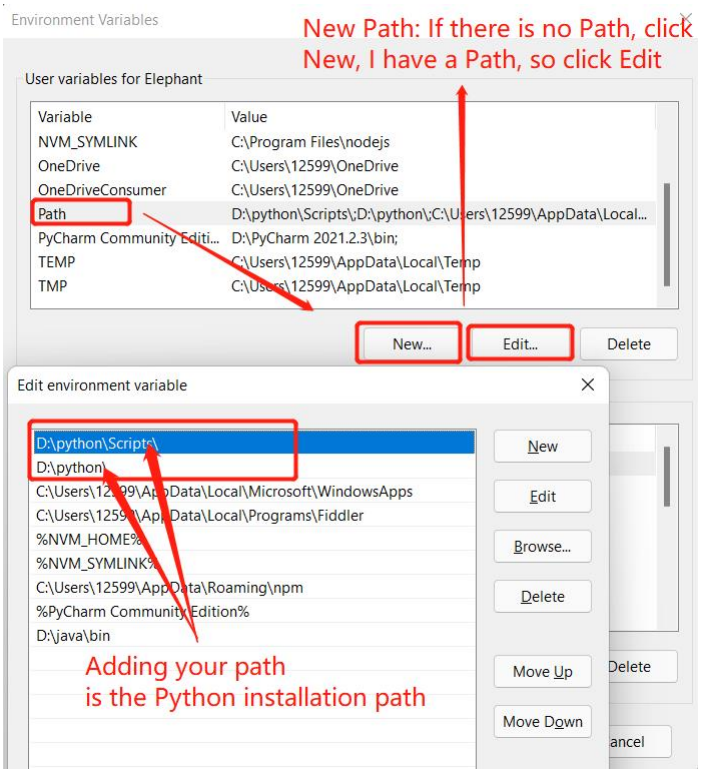
Follow these steps to manually add Python to the environment variables.

- Right-click the "My Computer" icon --> Properties -> Advanced system settings -> Environment Variables

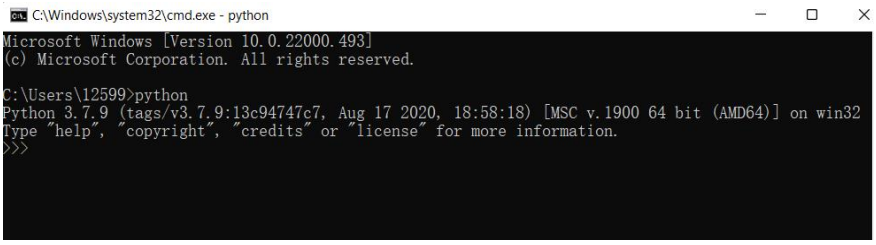


- Environment variables include user variables and system variables. For user variables, users can use their downloaded programs through the "cmd" command. Write the absolute path of the target program into the user variables.





- After configuration is complete, open the Command Prompt window (Win+R; type "cmd" and press "Enter"), then type "Python".



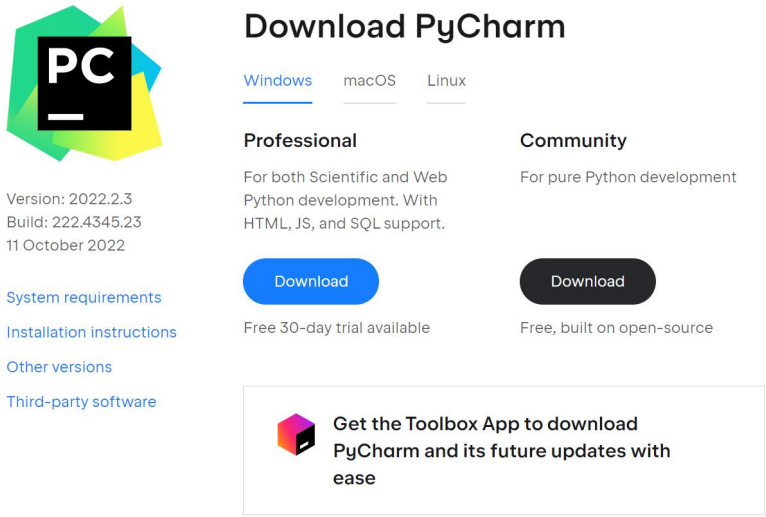
2 Installing PyCharm

PyCharm is a powerful, cross-platform Python editor. Follow these steps to download and install PyCharm.

Go to **PyCharm** to download PyCharm.

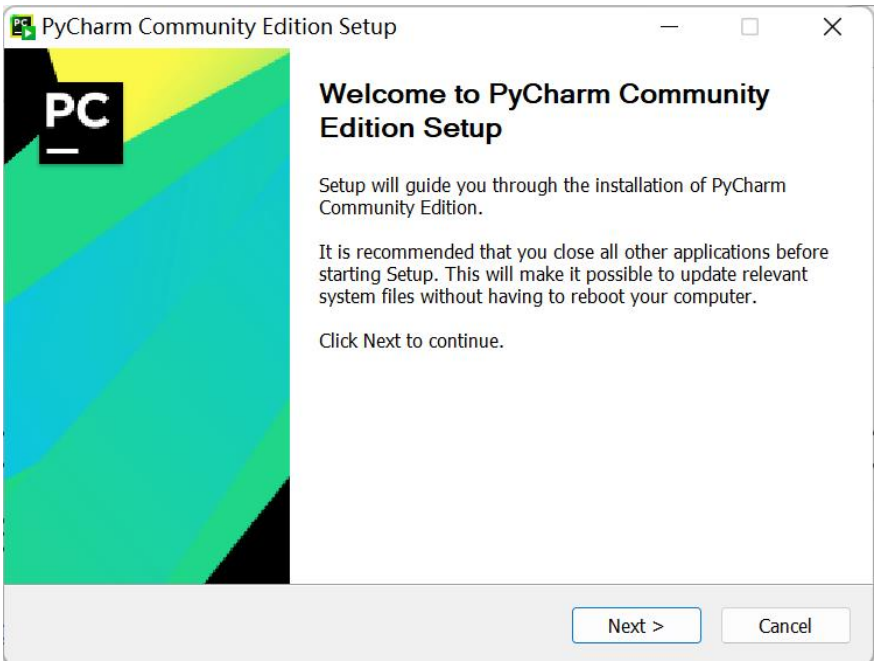
2.1 Download and Installation

Official website view:

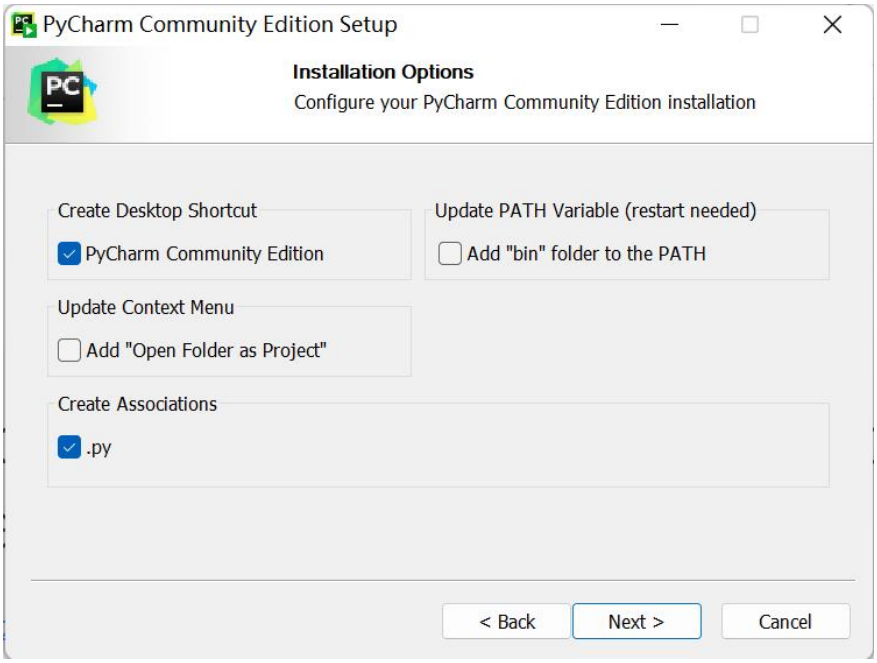


It is recommended to install the free version.

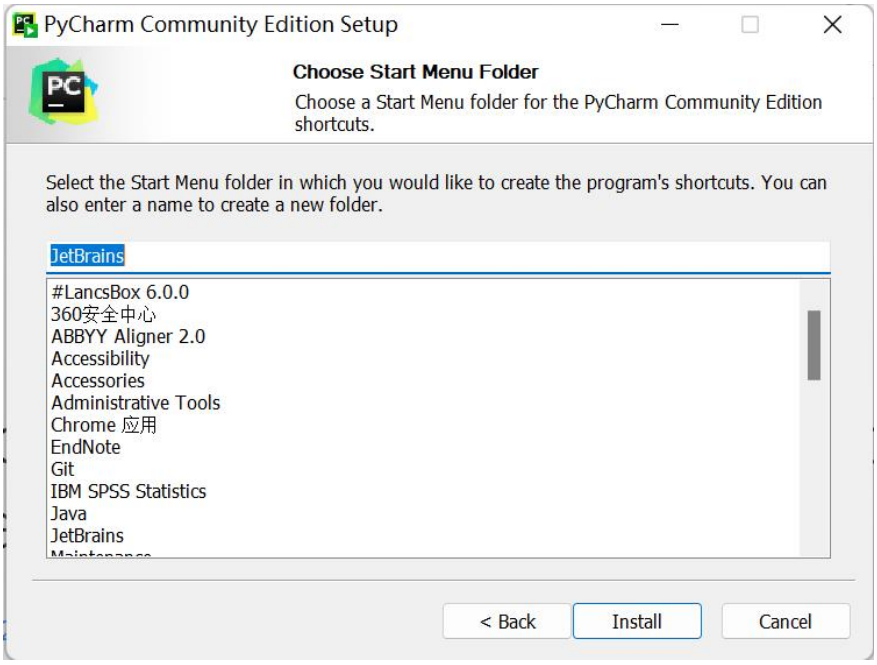
- Click "Next":



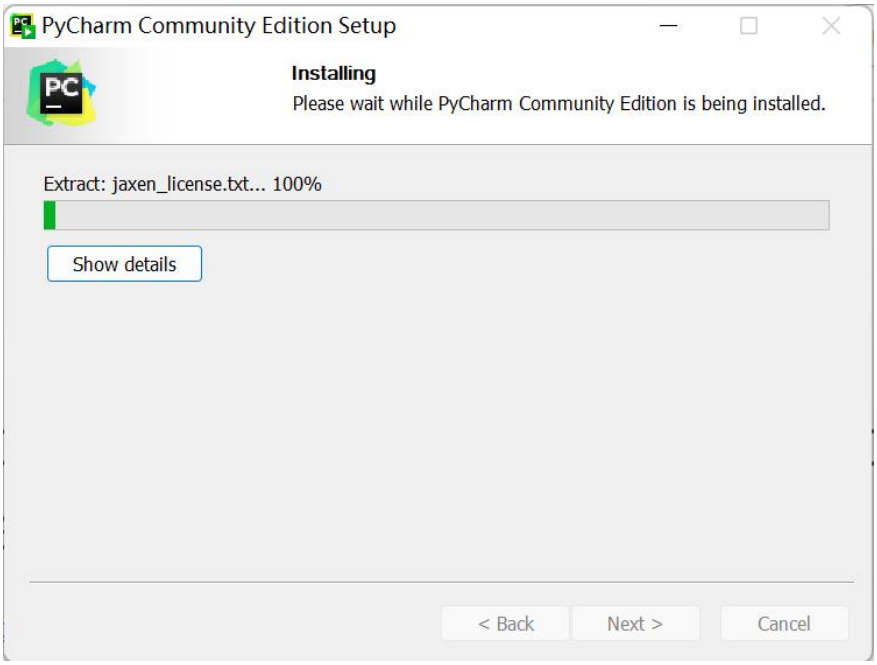
- Select options according to your needs, then select "Next":



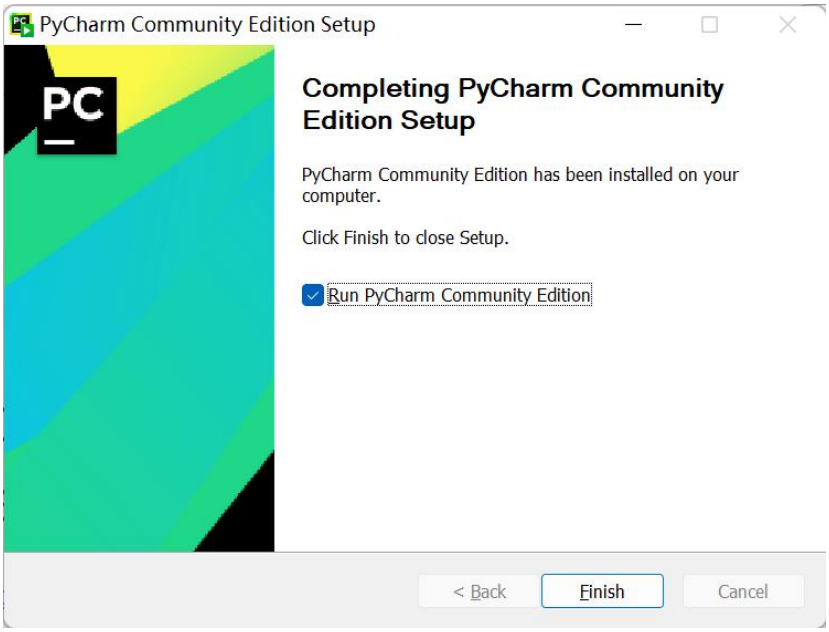
- Click "Install":



- Installation:

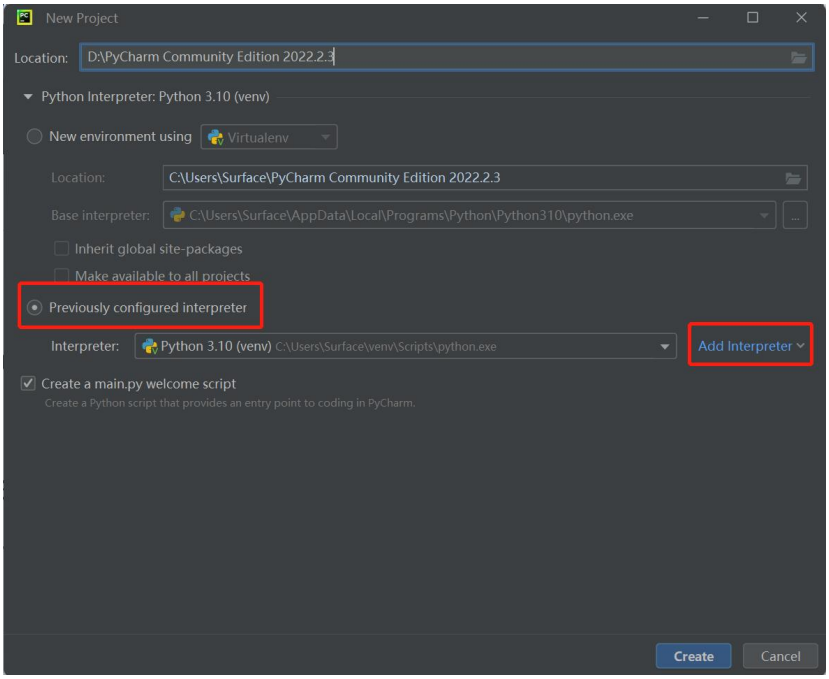


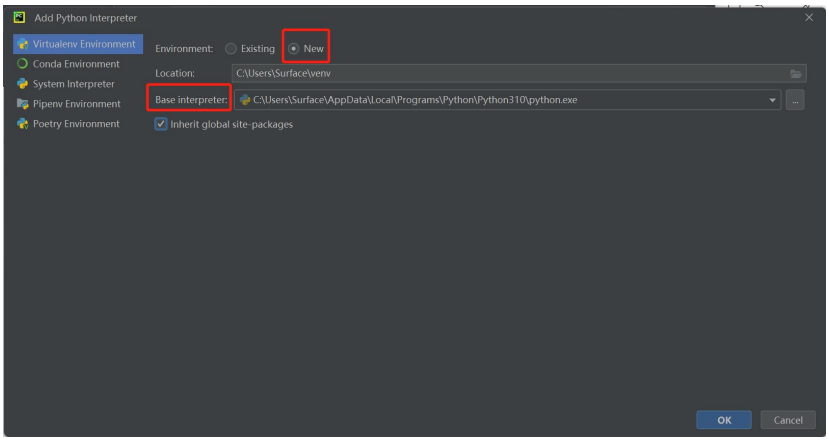
- Click "Finish"



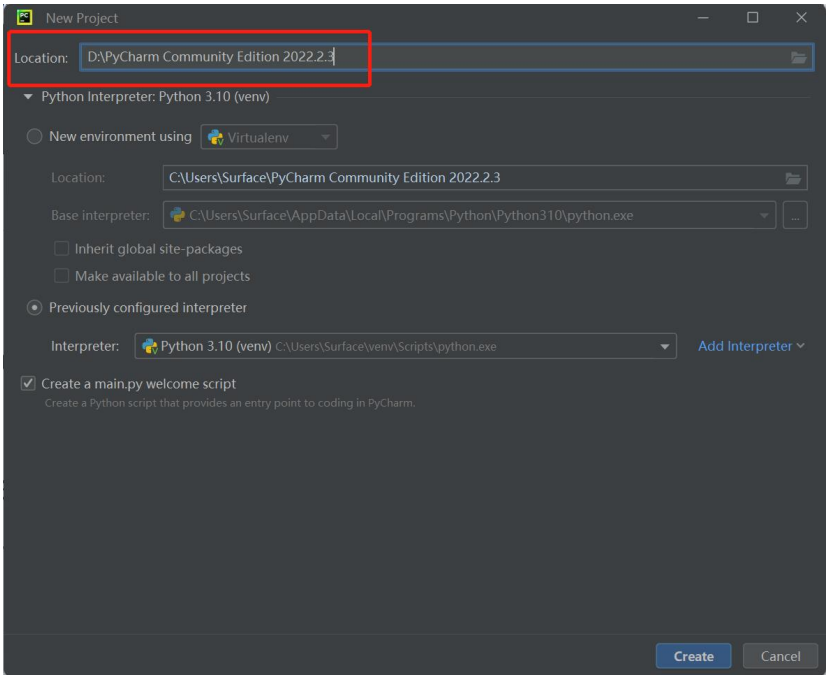
2.2 Creating a New Project

- Click "+Create New Project":
- `Interpreter` is used to interpret Python programs. Select "Add Interpreter" -> "New" to add a base interpreter.





- `Location` refers to the location where the Python files are saved. Choose a file to place your program.



- Click "Create", and an example will appear:
- Right-click the option indicated by the red arrow, then create a new Python file.
- Type the name of the new file.

3 Preparations

- pymycobot Installation. Enter the command "pip install pymycobot --upgrade --user" through the terminal (Win+R) "cmd".

```
pip install pymycobot --upgrade --user
```

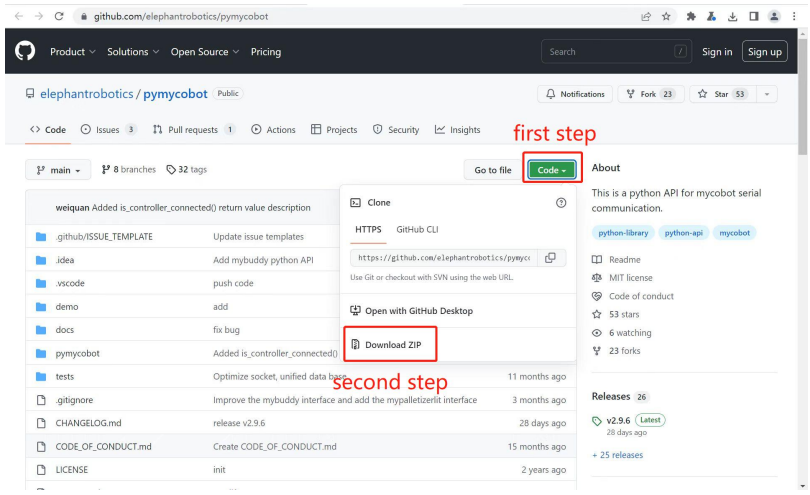
- Source Code Installation. Open the terminal (Win+R, type `cmd`), then enter the following commands to install.

```
git clone https://github.com/elephantrobotics/pymycobot.git <your-path>
#Fill in your installation address in <your-path>, do not choose the current def

cd <your-path>/pymycobot
#Enter the pymycobot folder of the downloaded package.

#Run one of the following commands according to your Python version.
# install
 python2 setup.py install
# or
 python3 setup.py install
```

- Update pymycobot

```
pip install pymycobot --upgrade
```

> **Note:**
>
> 1. If there are no red wavy lines under the code, it means pymycobot has been successfully installed.
> 2. If red wavy lines appear, please manually download pymycobot from **https://github.com/elephantrobotics/pymycobot** and place it in the Python library.



Basic Usage of Python

```
mr = Mercury('/dev/right_arm')

ml.power_on()
mr.power_on()

print(ml.get_angles())
print(mr.get_angles())
```

← Previous Page | Next Page →

6.1 Python API

[toc]

6.1.1 API usage instructions

API (Application Programming Interface), also known as Application Programming Interface functions, are predefined functions. When using the following function interfaces, please import our API library at the beginning by entering the following code, otherwise it will not run successfully:

```
# Example
from pymycobot import Mercury

ml = Mercury('/dev/left_arm')
mr = Mercury('/dev/right_arm')

ml.power_on()
mr.power_on()

print(ml.get_angles())
print(mr.get_angles())
```

1. System Status

`get_system_version()`

- **function:** get system version
- **Return value:** system version

`get_robot_type()`

- **function:** get robot id
- **Return value:** Definition Rule: Actual machine model. For example, the Mercury A1 model is 4500

`get_atom_version()`

- **function:** Get the end version number
- **Return value:** End parameters(`float`)

`get_robot_status()`

- **function:** Upper computer error security status
- **Return value:** 0 - Normal. other - Robot triggered collision detection

2. Overall Status

`power_on()`

- **function:** atom open communication (default open)
  - Attentions： After executing poweroff or pressing emergency stop, it takes 7 seconds to power on and restore power
- **Return value:**
  - `1` - Power on completed.
  - `0` - Power on failed

`power_off()`

- **function:** Power off of the robotic arm
- **Return value:**
  - `1` - Power on completed.
  - `0` - Power on failed

`is_power_on()`

- **function:** judge whether robot arms is powered on or not
- **Return value:**
  - `1` : power on
  - `0` : power off
  - `-1` : error

`release_all_servos()`

- **function:** release all robot arms
  - Attentions： After the joint is disabled, it needs to be enabled to control within 1 second
- **Parameters**： `data` （optional）： The way to relax the joints. The default is damping mode, and if the 'data' parameter is provided, it can be specified as non damping mode (1- Undamping).
- **Return value:**
  - `1` - release completed.
  - `0` - release failed

`focus_all_servos()`

- **function:** Turn on robot torque output
  - **Return value:**
    - `1` : complete
    - `0` : failed
    - `-1` : error

3.MDI Mode and Operation

`get_angles()`

- **function:** get the degree of all joints
- **Return value**: `list` a float list of all degree

`get_angle()`

- **function:** Get single joint angle
- **Parameters**： joint_id (int): 1 ~ 7
- **Return value:** Array of angles corresponding to joints

`send_angle(id, degree, speed)`

- **function:** send one degree of joint to robot arm
- **Parameters:**
  - `id` : Joint id(`genre.Angle`), range int 1-7
  - `degree` : degree value(`float`) Arm joint range of motion | Joint Id | range | | ---- | ---- | | 1 | -178 ~ 178 | | 2 | -74 ~ 130 | | 3 | -178 ~ 178 | | 4 | -180 ~ 10 | | 5 | -178 ~ 178 | | 6 | -20 ~ 273 | | 7 | -180 ~ 180 |

    Body joint range of motion. Joint 11 is the chin camera. Joint 12 is the neck. Joint 13 is the lumbar joint | Joint Id | range | | ---- | ---- | | 11 | -60 ~ 0 | | 12 | -138 ~ 188 | | 13 | -118 ~ 118 |

  - `speed` ： the speed and range of the robotic arm's movement 1~100

`send_angles(angles, speed)`

- **function：** Send all angles to all joints of the robotic arm
- **Parameters:**
  - `angles` : a list of degree value(`List[float]`), length 7
  - `speed` : (`int`) 1 ~ 100

`get_coords()`

- **function:** Obtain robot arm coordinates from a base based coordinate system
- **Return value:** a float list of coord:[x, y, z, rx, ry, rz]

`send_coord(id, coord, speed)`

- **function:** send one coord to robot arm
- **Parameters:**
  - `id` :send one coord to robot arm, 1-6 corresponds to [x, y, z, rx, ry, rz]
  - `coord` : coord value(`float`) | Coord Id | range | | ---- | ---- | | 1 | -466 ~ 466 | | 2 | -466 ~ 466 | | 3 | -240 ~ 531 | | 4 | -180 ~ 180 | | 5 | -180 ~ 180 | | 6 | -180 ~ 180 |
  - `speed` : (`int`) 1-100

`send_coords(coords, speed, mode)`

- **function:**: Send overall coordinates and posture to move the head of the robotic arm from its original point to your specified point
- **Parameters:**
  - coords: : a list of coords value `[x,y,z,rx,ry,rz]` ,length6
  - speed `(int)` : 1 ~ 100

`pause()`

- **function:** Control the instruction to pause the core and stop all movement instructions

`is_paused()`

- **function:** Check if the program has paused the move command
- **Return value:**
  - `1` - paused
  - `0` - not paused
  - `-1` - error

`resume()`

- **function:** resume the robot movement and complete the previous command

`stop()`

- **function:** stop all movements of robot
- **Return value**:
  - `1` - stopped
  - `0` - not stop
  - `-1` - error

`is_in_position(data, flag)`

- **function** : judge whether in the position.
- **Parameters:**
  - data: Provide a set of data that can be angles or coordinate values. If the input angle length range is 7, and if the input coordinate value length range is 6
  - flag data type (value range 0 or 1)
    - `0` : angle
    - `1` : coord
- **Return value**:
  - `1` - true
  - `0` - false
  - `-1` - error

`is_moving()`

- **function:** judge whether the robot is moving
- **Return value:**
  - `1` moving
  - `0` not moving
  - `-1` error

4. JOG Mode and Operation

`jog_angle(joint_id, direction, speed)`

- **function:** jog control angle
- **Parameters**:
  - `joint_id` : Represents the joints of the robotic arm, represented by joint IDs ranging from 1 to 7
  - `direction(int)` : To control the direction of movement of the robotic arm, input `0` as negative value movement and input `1` as positive value movement
  - `speed` : 1 ~ 100

`jog_coord(coord_id, direction, speed)`

- **function:** jog control coord.
- **Parameters:**
  - `coord_id` : (`int`) Coordinate range of the robotic arm: 1~6
  - `direction` :(`int`) To control the direction of machine arm movement, `0` - negative value movement, `1` - positive value movement
  - `speed` : 1 ~ 100

`jog_increment_angle(joint_id, increment, speed)`

- **function:** Single joint angle increment control
- **Parameters**:
  - `joint_id` : 1-7
  - `increment` : Incremental movement based on the current position angle
  - `speed` : 1 ~ 100

`jog_increment_coord(coord_id, increment, speed)`

- **function:** Single joint angle increment control
- **Parameters**:
  - `joint_id` : axis id 1 - 6.
  - `increment` : Incremental movement based on the current position coord
  - `speed` : 1 ~ 100

5. Coordinate controlled attitude deviation angle

`get_solution_angles()`

- **function:** Obtain the value of zero space deflection angle

- **Return value**: Zero space deflection angle value

`set_solution_angles(angle, speed)`

- **function:** Obtain the value of zero space deflection angle
- **Parameters:**
  - `angle` : Input the angle range of joint 1, angle range -90 to 90
  - `speed` : 1 - 100.

6. Joint software limit operation

`get_joint_min_angle(joint_id)`

- **function:** Read the minimum joint angle
- **Parameters:**
  - `joint_id` : Enter joint ID (range 1-7)
- **Return value**: `float` Angle value

`get_joint_max_angle(joint_id)`

- **function:** Read the maximum joint angle
- **Parameters:**
  - `joint_id` : Enter joint ID (range 1-7)
- **Return value:** `float` Angle value

`set_joint_min(id, angle)`

- **function:** Set minimum joint angle limit
- **Parameters:**
  - `id` : Enter joint ID (range 1-7)
  - `angle` : Refer to the limit information of the corresponding joint in the send_angle() interface, which must not be less than the minimum value

`set_joint_max(id, angle)`

- **function:** Set minimum joint angle limit
- **Parameters:**
  - `id` : Enter joint ID (range 1-7)
  - `angle` : Refer to the limit information of the corresponding joint in the send_angle() interface, which must not be greater than the maximum value

7. Joint motor control

`is_servo_enable(servo_id)`

- **function:** Detecting joint connection status
- **Parameters:** `servo id` 1-7

- **Return value:**
  - `1` : Connection successful
  - `0` : not connected
  - `-1` : error

`is_all_servo_enable()`

- **function:** Detect the status of all joint connections
- **Return value:**
  - `1` : Connection successful
  - `0` : not connected
  - `-1` : error

`set_servo_calibration(servo_id)`

- **function:** The current position of the calibration joint actuator is the angle zero point
- **Parameters**:
  - `servo_id` : 1 - 7

`release_servo(servo_id)`

- **function:** Set the specified joint torque output to turn off
- **Parameters**:
  - `servo_id` : 1 ~ 7
- **Return value:**
  - `1` : release successful
  - `0` : release failed
  - `-1` : error

`focus_servo(servo_id)`

- **function**: Set the specified joint torque output to turn on
- **Parameters**: `servo_id` : 1 ~ 7
- **Return value:**
  - `1` : focus successful
  - `0` : focus failed
  - `-1` : error

`set_break (joint_id, value)`

- **function:** Set break point
- **Parameters**:
  - `joint_id` : int. joint id 1 - 7
  - `value` : int. 0 - disable, 1 - enable
- **Return value:** 0 : faile; 1 : success

`get_servo_speeds()`

- **function**: Get the movement speed of all joints
- **Return value**: unit step/s

`get_servo_currents()`

- **function**: Get the movement current of all joints
- **Return value**: 0 ~ 5000 mA

`get_servo_status()`

- **function**: Get the movement status of all joints
- **Return value**: a value of 0 means no error

`servo_restore(joint_id)`

- **function**: Clear joint abnormalities
- **Parameters**:
  - `joint_id` : int. joint id 1 - 7

8. Robotic arm end IO control

`set_digital_output(pin_no, pin_signal)`

- **function:** set IO statue
- **Parameters**
  - `pin_no` (int): Pin number
  - `pin_signal` (int): 0 / 1

`get_digital_input(pin_no)`

- **function:** read IO statue
- **Parameters**: `pin_no` (int)
- **Return value**: signal

9. Robotic arm end gripper control

`set_gripper_state(flag, speed, _type_1=None)`

- **function**: Adaptive gripper enable
- **Parameters**:
  - `flag (int)` : 0 - open 1 - close, 254 - release
  - `speed (int)` : 1 ~ 100
  - `_type_1 (int)` :
    - `1` : Adaptive gripper (default state is 1)
    - `2` : A nimble hand with 5 fingers
    - `3` : Parallel gripper

        - `4` : Flexible gripper

`set_gripper_value(gripper_value, speed, gripper_type=None)`

- **function**: Set the gripper value
- **Parameters**:
    - `gripper_value (int)` : 0 ~ 100
    - `speed (int)` : 1 ~ 100
    - `gripper_type (int)` :
        - `1` : Adaptive gripper (default state is 1)
        - `2` : A nimble hand with 5 fingers
        - `3` : Parallel gripper
        - `4` : Flexible gripper

`set_gripper_calibration()`

- **function**: Set the current position of the gripper to zero

`set_gripper_enabled(value)`

- **function**: Adaptive gripper enable setting
- **Parameters**:
    - `value` 1: Enable 0: Release

`set_gripper_mode(mode)`

- **function**: Set gripper mode
- **Parameters**:
    - `value` :
        - 0: Transparent transmission mode
        - 1: normal mode

`get_gripper_mode()`

- **function**: Get gripper mode
- **Return value**:
    - 0: Transparent transmission mode
    - 1: normal mode

10. Button function at the end of the robot arm

`is_btn_clicked()`

- **function**: Get the status of the button at the end of the robot arm

- **Return value**:
  - 0: no clicked
  - 1: clicked

`set_color(r, g, b)`

- **function**: Set the color of the end light of the robotic arm
- **Parameters**:
  - `r (int)` : 0 ~ 255
  - `g (int)` : 0 ~ 255
  - `b (int)` : 0 ~ 255

11. Drag Teaching

`drag_teach_save()`

- **function:** Start recording and dragging teaching points.
  - Note: In order to display the best sports effect, the recording time should not exceed 90 seconds

`drag_teach_pause()`

- **function:** Pause sampling

`drag_teach_execute()`

- **function:** Start dragging the teach-in point, executing it only once.

`drag_teach_clean()`

- **function:** clear sample.

12. Cartesian space coordinate parameter setting

`set_tool_reference(coords)`

- **function:** Set tool coordinate system.
- **Parameters**: `coords` : (`list`) [x, y, z, rx, ry, rz].
- **Return value:** NULL

`get_tool_reference(coords)`

- **function:** Get tool coordinate system.
- **Return value:** `oords` : (`list`) [x, y, z, rx, ry, rz]

`set_world_reference(coords)`

- **function:** Set world coordinate system.
- **Parameters**: `coords` : (`list`) [x, y, z, rx, ry, rz].
- **Return value:** NULL

`get_world_reference()`

- **function:** Get world coordinate system.
- **Return value:** `list` [x, y, z, rx, ry, rz].

`set_reference_frame(rftype)`

- **function:** Set base coordinate system.
- **Parameters**: `rftype` : 0 - base 1 - tool.

`get_reference_frame()`

- **function:** Set base coordinate system.
- **Return value:**
  - `0` - base
  - `1` - tool.

`set_movement_type(move_type)`

- **function:** Set movement type.
- **Parameters**:
  - `move_type` : 1 - movel, 0 - moveJ.

`get_movement_type()`

- **function:** Get movement type.
- **Return value:**
  - `1` - movel
  - `0` - moveJ

`set_end_type(end)`

- **function:** Get end coordinate system
- **Parameters:**
  - `end (int)` : `0` - flange, `1` - tool

`get_end_type()`

- **function:** Obtain the end coordinate system
- **Return value:**
  - `0` - flange
  - `1` - tool

13. Circular motion

`write_move_c(transpoint, endpoint, speed)`

- function：Arc trajectory motion
- Parameters： `transpoint(list)` ： Arc passing through point coordinates `endpoint (list)` ： Arc endpoint coordinates `speed(int)` ： 1 ~ 100

14. Set bottom IO input/output status

`set_basic_output(pin_no, pin_signal)`

- **function**：Set Base IO Output
- **Parameters**：
  - `pin_no` (`int`) Pin port number, range 1 ~ 6
  - `pin_signal` (`int`): 0 - low. 1 - high

`get_basic_input(pin_no)`

- **function:** Read base IO input, range 1 ~ 6
- **Parameters:**
  - `pin_no` (`int`) pin number
- **Return value:** 0 - low. 1 - high

15. Set up 485 communication at the end of the robotic arm

`tool_serial_restore()`

- **function**：485 factory reset

`tool_serial_ready()`

- **function:** Set up 485 communication
- **Return value:** 0 : not set 1 : Setup completed

`tool_serial_available()`

- **function:** Set up 485 communication
- **Return value:** 0-Normal 1-Robot triggered collision detection

`tool_serial_read_data()`

- **function:** Read fixed length data. Before reading, read the buffer length first. After reading, the data will be cleared
- **Parameters**： data_len (int): The number of bytes to be read, range 1 ~ 45
- **Return value:** 0 : not set 1 : Setup completed

`tool_serial_write_data()`

- **function:** End 485 sends data， Data length range is 1 ~ 45 bytes

- **Return value:** 0-Normal 1-Robot triggered collision detection

`tool_serial_flush()`

- **function:** Clear 485 buffer
- **Return value:** 0-Normal 1-Robot triggered collision detection

`tool_serial_peek()`

- **function:** View the first data in the buffer, the data will not be cleared
- **Return value:** 1 byte data

`tool_serial_set_baud(baud)`

- **function:** Set 485 baud rate, default 115200
- **Parameters**: baud (int): baud rate
- **Return value:** NULL

`tool_serial_set_timeout(max_time)`

- **function:** Set 485 timeout in milliseconds, default 30ms
- **Parameters**
  - `max_time` : (int): timeout
- **Return value:** NULL

6.1.2 Base control interface

API Usage Instructions

```
# demo
from pymycobot import ChassisControl

mc = ChassisControl()

# Get battery voltage
voltage = mc.get_power_voltage()
print(voltage)

# Set the chassis RGB light strip to green
mc.set_color(255, 0, 0)
```

get_power_voltage()

- **Function:** Get the battery voltage.
- **Return value:** Battery voltage (floating point type), unit: Volt

get_ultrasonic_value()

- **Function:** Get all ultrasonic data.
- **Return value:** List of all ultrasonic data, unit: millimeter.

go_straight(speed)

- **Function:** Control the chassis to move forward.
- **Parameters:**
  - speed: (floating point type) the speed of chassis movement, the range is `0 ~ 1`, unit: m/s.

go_back(speed)

- **Function:** Control chassis to move backward.
- **Parameters:**
  - speed: (floating point type) chassis movement speed, range is `-1 ~ 0`, unit: m/s.

turn_left(speed)

- **Function:** Control the chassis to rotate to the left.
- **Parameters:**
  - speed: (floating point type) the speed of chassis movement, ranging from `0 ~ 1`, unit: m/s.

turn_right(speed)

- **Function:** Control chassis to rotate right.
- **Parameters:**
  - speed: (floating point type) chassis movement speed, range is `-1 ~ 0`, unit: m/s.

stop()

- **Function:** Stop motion.

set_color(r, g, b)

- **Function:** Set the color of the chassis RGB light strip.
- **Parameters:**
  - r (int): 0 ~ 255
  - g (int): 0 ~ 255
  - b (int): 0 ~ 255

← Previous Page | Next Page →

Case 1

Set the color of the end lights to blue

```
from pymycobot import Mercury

ml = Mercury("/dev/ttyTHS0")
mr = Mercury("/dev/ttyACM0")
# Robot powered on
ml.power_on()
mr.power_on()

ml.set_color(0,0,255)
mr.set_color(0,0,255)
```

Case 2

Angle control

```
from pymycobot import Mercury
import time
ml = Mercury("/dev/ttyTHS0")
mr = Mercury("/dev/ttyACM0")
# Robot powered on
ml.power_on()
mr.power_on()
# Single angle control
ml.send_angle(1, 90, 40)
mr.send_angle(1, 90, 40)
time.sleep(3)
ml.send_angle(1, 0, 40)
mr.send_angle(1, 0, 40)
time.sleep(3)

# All angle controls
ml.send_angles([0, 0, 90, 0, 0, 90, 0], 40)
mr.send_angles([0, 0, 90, 0, 0, 90, 0], 40)
time.sleep(3)
ml.send_angles([0, 0, 0, 0, 0, 90, 0], 40)
mr.send_angles([0, 0, 0, 0, 0, 90, 0], 40)
```

← Previous Page | Next Page →

Drag Teaching

drag_teach_save()

Start recording and dragging teaching points.

- Note: In order to display the best sports effect, the recording time should not exceed 90 seconds

drag_teach_pause()

Pause sampling.Pause sampling and enable the robotic arm again

drag_teach_execute()

Start dragging the teach-in point, executing it only once.

Case

```
from pymycobot import Mercury
import time
ml = Mercury("/dev/ttyTHS0")
mr = Mercury("/dev/ttyACM0")

ml.power_on()
mr.power_on()
# Left arm begins trajectory recording
ml.drag_teach_save()
# Recording duration is 10 seconds
time.sleep(10)
# Stop recording
ml.drag_teach_pause()
time.sleep(1)
# Start executing the recorded motion trajectory, only once
ml.drag_teach_execute()
```

← Previous Page | Next Chapter →

ROS

ROS is an open-source meta-operating system used for robots. It provides an operating system with expected services, including hardware abstraction, low-level device control, implementation of common functions, messages transfered between processes, and package management. It also provides the tools and library functions needed to obtain, compile, write, and run codes across computers.

The "graph" of ROS runtime is a loosely coupled peer-to-peer process network based on a ROS communication infrastructure. ROS implements several different communication methods, including a services mechanism based on synchronous RPC-style communication, a topics mechanism based on asynchronous streaming media data, and a parameter server for data storage.

ROS is not a real-time framework, but it can be embedded in real-time programs. Willow Garage's PR2 robot uses a system called pr2_etherCAT to send or receive ROS messages in real time. ROS may also be seamlessly integrated with Orocos real-time toolkits.

ROS Logo：

1 Design goals and characteristics of ROS

Many people ask "what are differences between ROS and other robot software platforms?" This question is difficult to answer. Because ROS is not a framework that integrates most functions or features. In fact, the main goal of ROS is to provide code reuse support for robot R&D. ROS is a framework (i.e. nodes) for distributed processes, which are encapsulated in program and function packages that can be easily shared and distributed. ROS also supports a federated system similar to a code repository, and this system also enables the collaboration and distribution of a project. This design enables the development and realization of a project to to be decided completely independently from the file system to the user interface (no limit by ROS). At the same time, all projects can be integrated with the basic tools of ROS.

To support the primary goals of sharing and collaboration, the ROS framework has several other features:

- Lean: ROS is designed to be as lean as possible so that codes written for ROS can be used with other robot software frameworks. The inevitable conclusion from this is that ROS can be easily integrated into other robot

software platforms: ROS can already be integrated with OpenRAVE, Orocos and Player.

- ROS insensitive libraries: The preferred development model of ROS is written with clean library functions that do not depend on ROS.
- Language independence: The ROS framework can simply be implemented in any modern programming language. ros has implemented Python version, C++ version and Lisp version. It also has experimental libraries for Java and Lua versions.
- Loose coupling: The function modules in ROS are encapsulated in independent function packages or meta-function packages, which are easy to share. The modules in the function package are run in units of nodes. With ROS standard IO used as the interface, developers does not need to pay attention to the internal implementation of the module, as long as they understand the interface rules, they can achieve reuse and point-to-point loose coupling between modules.
- Convenient testing: ROS has a built-in unit/integration testing framework called rostest, which can easily install or uninstall test modules.
- Scalable: ROS is applicable to large runtime systems and large development processes.
- Free and open-source: It has many developers and many function packages.

2 Why ROS is used?

Through ROS, we can realize the simulated control of robot arms in a virtual environment.

we will visualize robot arms through **rviz**, operates our robot arms in a variety of ways.

We will learn how to control our products through the platform in ros in the following chapters.

← Previous Section | Next Page →

Linux system environment

The system comes with Ubuntu (V-20.04) system when leaving the factory, with a built-in development environment. There is no need to set up and manage it. Just update the `mercury_x1_ros` package.

`mercury_x1_ros` is a ROS1 package launched by Elephant Robot for its Mercury X1 series robotic arms.

ROS1 Project address: http://github.com/elephantrobotics/mercury_x1_ros

Robotic arm API driver library address:
https://github.com/elephantrobotics/pymycobot

1 Update the mercury_ros package

In order to ensure that users can use the latest official packages in a timely manner, you can enter the `/home/er/` folder through the file manager, open the ROS1 environment terminal, and then run the update command:

```
# Clone the code on github
git clone https://github.com/elephantrobotics/mercury_x1_ros.git # Please check the at
cd ~/catkin_ws     # Back to work area
catkin_make # Build the code in the workspace
source devel/setup.bash # add environment variable
```

Note: If the `mercury_x1_ros` folder already exists in the `/home/er` directory, you need to delete the original `mercury_x1_ros` first and then execute the above command. Among them, `er` in the directory path is the user name of the system. If there are any inconsistencies, please modify them.

So far, the ROS1 environment construction has been completed.You can learn the basics of ROS or ROS use cases.

← Previous Page | Next Page →

1 ROS project structure

1.1 catkin workspace

Catkin workspace is the directory where catkin software packages are created, modified, and compiled. Catkin's workspace can be intuitively described as a warehouse, which contains various ROS project projects to facilitate system organization, management and calling.

- **Create workspace**:

```
mkdir -p ~/catkin_ws/src  # Create a folder
cd ~/catkin_ws/src        # Enter the folder
catkin_init_workspace     # Initialize the current directory into a ROS workspace
cd ..                     # Return to the parent directory
catkin_make               # Build the code in the workspace
```

The structure of catkin is very clear. It includes three paths: src, build, and devel. It may also include others under some compilation options. But these three folders are the default for the catkin compilation system. Their specific functions are as follows:

```
src/: ROS catkin software package (source code package)

build/: cache information and intermediate files of catkin (CMake)

devel/: Generated target files (including header files, dynamic link libraries, static
```

A simple workspace looks like this:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

1.2 ROS software package

Package is not only a software package on Linux, but also the basic unit of catkin compilation. The object we use catkin_make to compile is each ROS package.

```
+--PACKAGE
     +-- CMakeLists.txt
     +-- package.xml
     +-- src/
     +-- include/
     +-- scripts/
     +-- msg/
     +-- srv/
     +-- urdf/
     +-- launch/
```

- **CMakeLists.txt:** Defines the package name, dependencies, source files, target files and other compilation rules of the package. It is an essential component of the package.
- **package.xml:** Describes the package name, version number, author, dependencies and other information of the package, which is an indispensable component of the package.
- **src/:** stores ROS source code, including C++ source code (.cpp) and Python module (.py)
- **include/:** stores the header files corresponding to the C++ source code
- **scripts/:** stores executable scripts, such as shell scripts (.sh), Python scripts (.py)
- **msg/:** stores messages in custom format (.msg)
- **srv/:** stores services in custom formats (.srv)
- **urdf/:** stores the robot's model description (.urdf or .xacro) and 3D model files (.sda, .stl, .dae, etc.)
- **launch/:** stores launch files (.launch or .xml)

Create your own package:

- Command format:

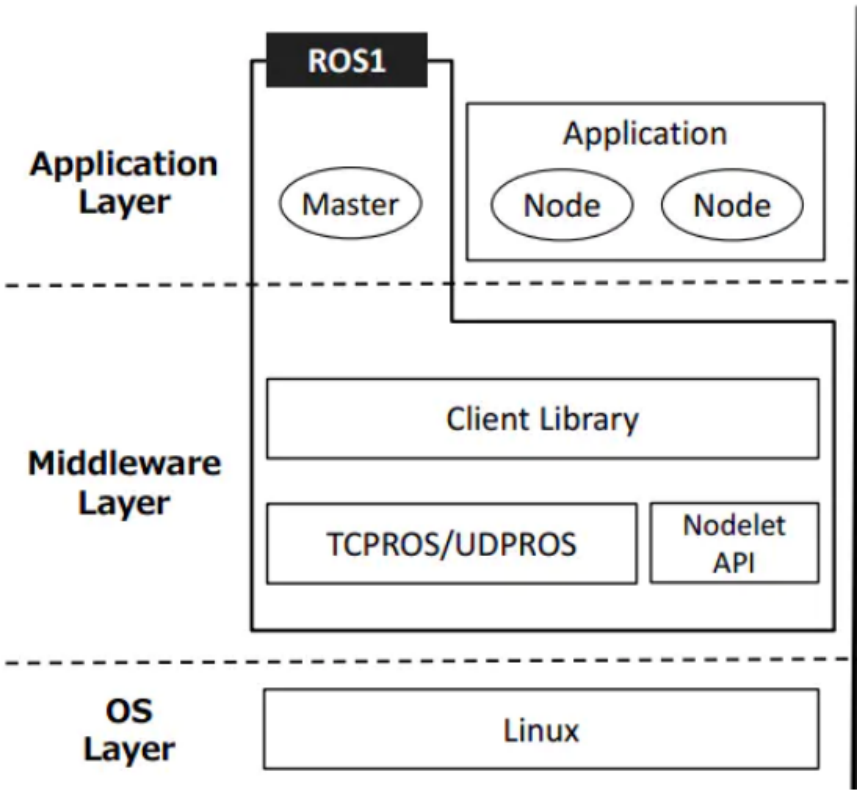
The catkin_create_pkg command will ask you to enter package_name. If necessary, you can also add some other dependent software packages later:

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

- For example:

```
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

2 ROS communication architecture



2.1 Master and node

1 Master

Node manager. Each node must register with the master before starting and manage the communication between nodes.

2 roscore

Starting the master will also start rosout (log management) and parameter server (parameter manager)

3 nodes

ROS processes and instances of running executable files in pkg.

```
$rosrun [pkg_name] [node_name] #Start
$rosnode list #List currently running node information
$rosnode info [node_name] #Display detailed information of a node
$rosnode kill [node_name] #End a node
```

4 launch

Start the master and multiple nodes.

```
$roslaunch [pkg_name] [file_name.launch]
```

2.2 Service and Topic

We provide some services and topics for interacting with mycobot.

1 Service

Enter in the command line:

```
source ~/catkin_ws/devel/setup.bash # Add environment variables
roslaunch mycobot_320_communication communication_service.launch
```

Support parameters:

- port：concatenate serial string
- baud：baud rate

Open a new command line:

```
# Display active service information
rosservice list

#/get_joint_angles
#/get_joint_coords
#/set_joint_angles
#/set_joint_coords
#/switch_gripper_status
#/switch_pump_status
```

Related commands and instructions:

command	Detailed description
rosservice list	Display active service information
rosservice info [service name]	Display information about the specified service
rosservice type [service name]	Show service type
rosservice find [service name]	Find a service for a specified service type
rosservice uri [service name]	show ROSRPC URI service
rosservice args [service name]	show service parameters
rosservice call [service name] [parameters]	Request service with input parameters

2 Topic

Enter in the command line:

```
source ~/catkin_ws/devel/setup.bash
roslaunch mycobot_320_communication communication_topic.launch
```

Support parameters:

- port：concatenate serial string
- baud：baud rate

Open a new command line:

```
# Display active service information
rostopic list

#/mycobot/angles_goal
#/mycobot/coords_goal
#/mycobot/angles_real
#/mycobot/coords_real
#/mycobot/pump_status
#/mycobot/gripper_status
```

Related commands and instructions:

Command	Detailed description
rostopic list	Display active topic list
rostopic echo [topic name]	Display the message content of the specified topic in real time
rostopic find [type name]	Display threads with messages of the specified type
rostopic type [topic name]	Displays the message type of the specified topic
rostopic bw [topic name]	Display the message bandwidth of the specified topic（bandwidth）
rostopic hz [topic name]	Display the message data publishing cycle of the specified topic
rostopic info [topic name	Display information about the specified topic
rostopic pub [topic name] [message type] [parameters]	Post a message with the specified topic name

The difference between service and topic:

	service	topic
Synchronization	Asynchronous	Synchronous
communication model	pub/sub	server/client
underlying protocol	ROSTCP/ROSUDP	ROSTCP/ROSUDP
Feedback Mechanism	No	Yes
buffer	Yes	No
Real-time	Weak	Strong
Node Relationship	Many-to-Many	One-to-Many
Applicable Scenarios	Data Transmission	Logical Processing

you can go toservice and topic learn more about the use of these two features

2.3 Introduction to msg and srv

- msg：The msg file is a simple text file describing the fields of a ROS message. They are used to generate source code for messages in different languages (c++ or python, etc.).

- srv： srv files are used to describe services. It consists of two parts: the request (request) and the response (response). msg files are stored in the msg directory of the package, and srv files are stored in the srv directory.

1 rosmsg

rosmsg is a command line tool for displaying information about ROS message types.

rosmsg demo:

```
rosmsg show       # Show message description
rosmsg info       # Display message information
rosmsg list       # list all messages
rosmsg md5        # Display md5 encrypted message
rosmsg package    # Display all messages under a feature pack
rosmsg packages   # List feature packs that contain messages
```

- rosmsg list will list all msgs in the current ROS
- rosmsg packages List all packages containing messages
- rosmsg package List all msgs under a package

```
//rosmsg package  # Package names
rosmsg package turtlesim
```

- rosmsg show Show message description

```
//rosmsg show # message name
rosmsg show turtlesim/Pose
# result:
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity
```

- rosmsg info Works the same as rosmsg show
- rosmsg md5 A check algorithm to ensure the consistency of data transmission

2 rossrv

rossrv is a command-line tool for displaying information about ROS service types, and uses a syntax that is highly similar to rosmsg.

```
rossrv show       # Display service message details
rossrv info       # Display information about service messages
rossrv list       # List all service information
rossrv md5        # Display md5 encrypted service messages
rossrv package    # Display all service messages under a package
rossrv packages   # Show all packages that contain service messages
```

- rossrv list Will list all srv messages in the current ROS
- rossrv packages List all packages that contain service messages
- rossrv package List all msgs under a package

```
//rossrv package # Package names
rossrv package turtlesim
```

- rossrv show Show message description

```
//rossrv show # message name
rossrv show turtlesim/Spawn
# result:
float32 x
float32 y
float32 theta
string name
---
string name
```

- rossrv info The effect is the same as rossrv show
- rossrv md5 Use md5 checksum (encryption) for service data

3 Introduction to URDF

- Unified Robot Description Format，Unified Robot Description Format, abbreviated as URDF. The urdf package in ROS contains a C++ parser for URDF, and URDF files describe robot models in XML format. *URDF cannot be used alone, it needs to be combined with Rviz or Gazebo. URDF is just a file that needs to be rendered into a graphical robot model in Rviz or Gazebo.

3.1 urdf file description

Code example:

Only part of the code is intercepted here for display:

```
<?xml version="1.0"?>
<robot  xmlns:xacro="http://www.ros.org/wiki/xacro" name="firefighter" >

<xacro:property name="width" value=".2" />



  <link name="base">
    <visual>
      <geometry>
       <mesh filename="package://mercury_description/urdf/mercury_a1/base.dae"/>
      </geometry>
      <origin xyz = "0.0 0 0 " rpy = " 0 0 0"/>
    </visual>
    <collision>
      <geometry>
       <mesh filename="package://mercury_description/urdf/mercury_a1/base.dae"/>
        </geometry>
        <origin xyz = "0.0 0 0 " rpy = " 0 0 0"/>
    </collision>
  </link>

  <link name="joint1">
    <visual>
      <geometry>
       <mesh filename="package://mercury_description/urdf/mercury_a1/joint1.dae"/>
      </geometry>
      <origin xyz = "0.0 0 -0.08 " rpy = " 0 0 0"/>
    </visual>
    <collision>
     <geometry>
       <mesh filename="package://mercury_description/urdf/mercury_a1/joint1.dae"/>
      </geometry>
      <origin xyz = "0.0 0 -0.08 " rpy = " 0 0 0"/>
    </collision>
  </link>

  <link name="joint2">
    <visual>
      <geometry>
       <mesh filename="package://mercury_description/urdf/mercury_a1/joint2.dae"/>
      </geometry>
    <origin xyz = "0.0 0 0.0 " rpy = " 3.14159 0 -1.5708 "/>
    </visual>
    <collision>
      <geometry>
       <mesh filename="package://mercury_description/urdf/mercury_a1/joint2.dae"/>
      </geometry>
      <origin xyz = "0.0 0 0.0 " rpy = " 3.14159 0 -1.5708"/>
```

```
    </collision>
  </link>

  <joint name="joint1_to_base" type="revolute">
    <axis xyz="0 0 1"/>
    <limit effort = "1000.0" lower = "-3.1066" upper = "3.1066" velocity = "0"/>
    <parent link="base"/>
    <child link="joint1"/>
    <origin xyz= "0 0 0.175" rpy = "0 0 0"/>
  </joint>

  <joint name="joint2_to_joint1" type="revolute">
    <axis xyz="0 0 1"/>
    <limit effort = "1000.0" lower = "-1.4311" upper = "2.2689" velocity = "0"/>
    <parent link="joint1"/>
    <child link="joint2"/>
    <!-- <origin xyz= "0 0  0" rpy = "1.5708 3.14159 0"/>   -->
    <origin xyz= "0 0  0" rpy = "-1.5708 0 0"/>
  </joint>

</robot>
```

It can be seen that the urdf file is not complicated, it is mainly composed of two parts, `link` and `joint`, which are repeated continuously.

3.2 link section

The link element describes a rigid body with inertial, visual features, and collision properties

3.2.1 Attributes

name: The name used to describe the link itself

3.2.2 element

- `<inertial>` (optional)
  - Inertia properties of connecting rods
  - `<origin>` (optional, defaults to identity if not specified)
    - Defines the reference coordinate of the inertial reference system relative to the connecting rod coordinate system. The coordinate must be defined at the center of gravity of the connecting rod, and its coordinate axis may not be parallel to the main axis of inertia.
    - xyz (optional, defaults to zero vector) Represents the offset in the x , y , z x,y,zx,y,z directions, in meters.
    - rpy(optional: defaults to identity if not specified) Indicates the rotation of the coordinate axis in the RPY direction, in radians.
  - `<mass>` Mass properties of connecting rods

- `<inertia>` 3×3 rotational inertia matrix, consisting of six independent quantities: ixx, ixy, ixz, iyy, iyz, izz。
- `<visual>` (optional)
  - Visual properties of the connecting rod. It is used to specify the shape of the link display (rectangle, cylinder, etc.). There can be multiple visual elements in the same link, and the shape of the link is formed by two elements. In general, the model is more complex and can be drawn through soildwork to generate stl calls, and simple shapes such as adding end effectors can be directly written. At the same time, the position of the geometry can be adjusted according to the gap between the theoretical model and the actual model.
  - `<namel>` (optional) The name of the connecting rod geometry.
  - `<origin>` (optional，defaults to identity if not specified)
    - The geometry coordinate system relative to the coordinate system of the connecting rod.
    - xyz (optional: defaults to zero vector) Represents the offset in the x , y , z x,y,zx,y,z directions, in meters.
    - rpy (optional: defaults to identity if not specified) Indicates the rotation of the coordinate axis in the RPY direction, in radians.

- `<geometry>` （required）
  - The shape of the visualization, which can be one of the following:
  - `<box>` A rectangle with elements including length, width, and height. The origin is in the center.
  - `<cylinder>` Cylinder, elements include radius and length. center of origin.
  - `<sphere>` Sphere, element containing the radius. The origin is in the center.
  - `<mesh>` The grid, as determined by the file, also provides a scale to define its boundaries. Collada .dae files are recommended, .stl files are also supported, but must be a local file.
- `<material>` (optional)
  - Visualize the component's material. It can be defined outside the link tag, but it must be inside the robot tag. When defining outside the link tag, the name of the link must be quoted.
  - `<color>` (optional) Color, consisting of red/green/blue/alpha, in the range [0,1].
  - `<texture>` (optional) Material properties, defined by the file.

- `<collision>` (optional)
  - Collision properties of the link. Collision properties differ from visual properties of connecting rods, and simple collision models are often used to simplify calculations. The same link can have multiple collision attribute labels, and the collision attribute representation of the link is composed of the set of geometric shapes defined by it.
  - `<name>` (optional) Specifies the name of the connecting rod geometry
  - `<origin>` (optional，defaults to identity if not specified)
    - The reference coordinate system of the collision component is relative to the reference coordinate system of the link coordinate system.

        - xyz (optional, default zero vector) Represents the offset in the x , y , z x,y,zx,y,z directions, in meters.
        - rpy (optional, defaults to identity if not specified) Indicates the rotation of the coordinate axis in the RPY direction, in radians.
    - `<geometry>` Same as the geometry element description above

Detailed elements and the role of each element can go to official documentation to view

3.3 joint part

The joint section describes the kinematics and dynamics of the joint and specifies safety limits for the joint.

3.3.1 properties of joint:

name：

Specifies a unique name for the joint

type：

Specifies the type of joint, where type can be one of the following:

- revolute - A hinged joint that rotates along an axis, the range of which is specified by the upper and lower bounds.
- Continuous - A continuous hinged joint that rotates around an axis with no upper and lower bounds.
- Prismatic - A sliding joint that slides along an axis, the range of which is specified by upper and lower limits. +Fixed - this is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require axes, calibration, dynamics, limits or safety_controller。
- Floating - This joint allows motion in all 6 degrees of freedom.
- Plane - This joint allows movement in a plane perpendicular to the axis.

3.3.2 elements of joint

- `<origin>` (optional， defaults to identity if not specified) In the transformation from parent link to child link, the joint is located at the origin of the child link. Modifying this parameter can adjust the position of the connecting rod. It can be used to adjust the error between the actual model and the theoretical model, but it is not recommended to modify it greatly, because this parameter affects the connecting rod stl The position of , easily affects the collision detection effect.
    - xyz (optional: default to zero vector) Represents the offset in the x , y , z x,y,zx,y,z axis directions, in meters.
    - rpy (optional: default to zero vector) Represents the angle of rotation around a fixed axis: roll is around the x-axis, pitch is around the y-axis, and yaw is around the z-axis, expressed in radians.
- `<parent>` (required)

    - The name of the parent link is a mandatory attribute.
    - link The name of the parent link is the name of the link in the robot structure tree.
- `<child>` (required)
    - The name of the child link is a mandatory attribute.
    - link The name of the child link is the name of the link in the robot structure tree.
- `<axis>` (optional: defaults to (1,0,0))
    - The joint's axis is in the joint's coordinate system. This is the axis of rotation (revolute joint), the axis of movement of the prismatic joint, and the standard plane of the planar joint. This axis is specified in the joint coordinate system. Modifying this parameter can adjust the axis around which the joint rotates. It is often used to adjust the rotation direction. If the model rotation is opposite to the actual one, just multiply by -1. Fixed and floating joints do not need this element.
    - xyz(required) x , y , z x, y, zx, y, z components representing axis vectors, as normalized vectors.
- `<calibration>` (optional)
    - The reference point of the joint, used to correct the absolute position of the joint.
    - rising (optional) When the joint is moving forward, the reference point triggers a rising edge.
    - falling (optional) When the joint is moving forward, the reference point triggers a falling edge.
- `<dynamics>` (optional)
    - This element is used to specify the physical properties of the joint. Its value is used to describe the modeling performance of the joint, especially during simulation.

`<limit>` (Required when the joint is a rotation or translation joint)

- This element is a joint kinematics constraint.
- lower (optional, default to 0) Specify the attribute of the lower bound of the joint's motion range (the unit of the revolute joint is radians, and the unit of the prismatic joint is meters). This attribute is ignored for continuous joints.
- upper (optional, defaults to 0) Specify the attribute of the upper bound of the joint's motion range (the unit of the revolute joint is radians, and the unit of the prismatic joint is the meter). This attribute is ignored for continuous joints.
- effort (required) This property specifies the maximum force at which the joint will run.
- velocity (required) This property specifies the maximum speed of the joint runtime.

`<mimic>` (optional)

- This tag is used to specify a defined joint to mimic an existing joint. The value of this joint can be calculated using the following formula: `value = multiplier * other_joint_value + offset`

- joint(required) The name of the joint to mimic.
- multiplier(optional) Specify the multiplier factor in the above formula.
- offset(optional) Specify the offset term in the above formula. Default value is 0

`<safety_controller>` (optional)

- This element is a security control limit. The data under this element will be read into move_group, but it is invalid in practice. Move_group will skip this limit and directly read the parameter content under limit. At the same time, setting this element may cause planning failure.
- soft_lower_limit (optional, defaults to 0) This attribute specifies the lower bound of the joint security control boundary, which is the starting limit point of the joint security control. This value needs to be greater than the lower value in the above limit.
- soft_upper_limit (optional, defaults to 0) This attribute specifies the upper bound of the joint security control boundary, which is the starting limit point of the joint security control. This value needs to be less than the upper value in the above limit.
- k_position(optional, defaults to 0) This attribute is used to describe the relationship between position and velocity.
- k_velocity(required) This property is used to describe the relationship between force and velocity.

Detailed elements and the role of each element can go to http://wiki.ros.org/urdf/XML/joint to view.

4 Commonly used command tools

In ROS, there are many commonly used command line tools, which can help you develop, debug, manage ROS nodes, etc. The following are some commonly used ROS command line tools:

4.1 Compile workspace

```
caktin_make
```

4.2 roscore

Start the ROS master node. Before running a ROS node, you usually need to start roscore first

```
roscore
```

4.3 rosrun

Run the specified ROS node.

```
rosrun package_name node_name
```

4.4 roslaunch

Use the Launch file to start one or more ROS nodes.

```
roslaunch package_name launch_file.launch
```

4.5 rosnode

View running ROS node information.

```
rosnode list
rosnode info node_name
```

4.6 rostopic

View information about running ROS topics.

```
rostopic list
rostopic echo topic_name
```

4.7 rosservice

View and call ROS services.

```
rosservice list
rosservice call service_name
```

4.8 rosparam

Get and set ROS parameters.

```
rosparam get parameter_name
rosparam set parameter_name value
```

4.9 rosmsg

View ROS message types.

```
rosmsg show message_type
```

4.10 rosdep

Install dependencies of ROS packages.

```
rosdep install package_name
```

4.11 Environment variables

View the ROS_PACKAGE_PATH environment variable

```
echo $ROS_PACKAGE_PATH
```

← Previous Page | Next Page →

Brief introduction and use of rviz

rviz is a 3D visualization platform in ROS. On one hand, it can realize the graphical display of external information, and on the other hand, it can also release control information to an object through rviz, realizing the monitoring and control of a robot.

1 Introduction to rviz interface

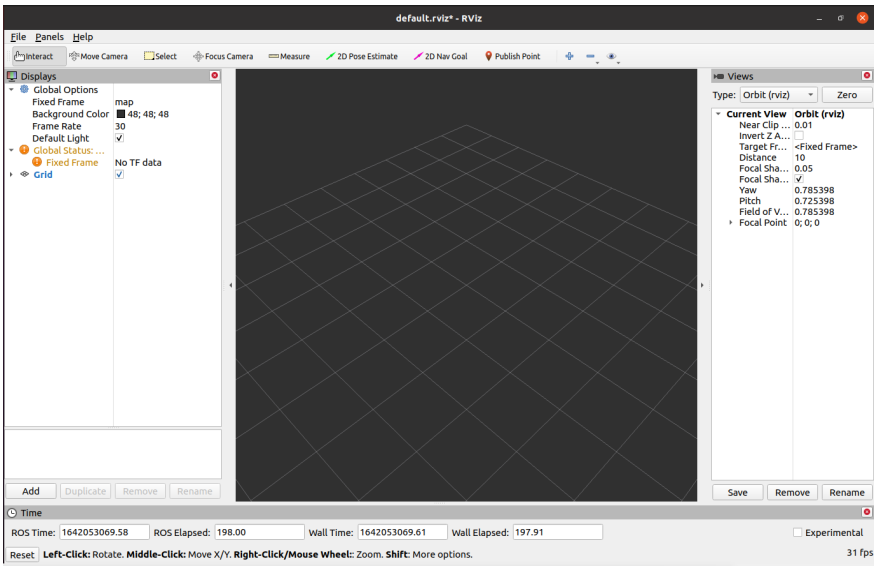
Open a new ROS1 terminal (shortcut key: `Ctrl+Alt+T`) and enter the following command:

```
roscore
```

Then open a new ROS1 terminal (shortcut key: `Ctrl+Alt+T`) and input the following command to open rviz.

```
rosrun rviz rviz
# or
rviz
```

Open rviz, and the following interface will be displayed:



1.1 Introduction of all areas

- There is a list of monitors on the left. The monitor is a device that draws something in a 3D world and may have some options available in the display list.
- On the top is a toolbar, which allows the user to use various function buttons to select tools with multiple functions.

- The middle part is the 3D view: It is a main screen where various data can be viewed in three dimensions. The background color, fixed frame, grid, etc. of the 3D view can be set in detail in the Global Options and Grid items displayed on the left.
- Below is the time display area, including system time and ROS time.
- The right side is the observation angle setting area where different observation angles can be set.

We only give a rough introduction in this part. If you want to know more details, go to User Guide.

If you want to know more information about rviz, go to Official documents.

← Previous Page | Next Page →

Mercury X1 Controls

Here we mainly introduce how to control the movement of Mercury X1 through a series of related instructions.

1. Chassis underlying communication

First, start the underlying communication and map construction program of the chassis. Open the ROS1 environment terminal and run the command:

```
roslaunch turn_on_mercury_robot mapping.launch
```

Output the following information:

```
... logging to /home/er/.ros/log/34bcf3be-0606-11ef-8293-e8fb1c355a09/roslaunch-er-des
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://er-desktop:43117/

SUMMARY
========

PARAMETERS
 * /LD14/disable_max: [180]
 * /LD14/disable_min: [120]
 * /LD14/flag_parted: True
 * /if_akm_yes_or_no: no
 * /lsm10_v2/baud_rate: 460800
 * /lsm10_v2/disable_max: [260]
 * /lsm10_v2/disable_min: [100]
 * /lsm10_v2/frame_id: laser
 * /lsm10_v2/max_distance: 30.0
 * /lsm10_v2/min_distance: 0.0
 * /lsm10_v2/scan_topic: scan
 * /lsm10_v2/serial_port: /dev/wheeltec_lidar
 * /lsm10_v2/truncated_mode: 1
 * /lsn10/disable_max: [10, 40, 70, 180]
 * /lsn10/disable_min: [0, 30, 60, 90]
 * /lsn10/max_distance: 30.0
 * /lsn10/min_distance: 0.0
 * /lsn10/truncated_mode: 0
 * /mercury_robot/gyro_frame_id: gyro_link
 * /mercury_robot/odom_frame_id: odom
 * /mercury_robot/robot_frame_id: base_footprint
 * /mercury_robot/serial_baud_rate: 115200
 * /mercury_robot/usart_port_name: /dev/wheeltec_con...
 * /robot_description: <?xml version="1....
 * /robot_pose_ekf/base_footprint_frame: base_footprint
 * /robot_pose_ekf/freq: 30.0
 * /robot_pose_ekf/imu_used: True
 * /robot_pose_ekf/odom_data: odom
 * /robot_pose_ekf/odom_used: True
 * /robot_pose_ekf/output_frame: odom
 * /robot_pose_ekf/sensor_timeout: 2.0
 * /robot_pose_ekf/vo_used: False
 * /rosdistro: noetic
 * /rosversion: 1.16.0
 * /rplidarNode/angle1_end: 50.0
 * /rplidarNode/angle1_start: 40.0
 * /rplidarNode/angle2_end: 140.0
```

```
 * /rplidarNode/angle2_start: 130.0
 * /rplidarNode/angle3_end: 230.0
 * /rplidarNode/angle3_start: 220.0
 * /rplidarNode/angle4_end: 320.0
 * /rplidarNode/angle4_start: 310.0
 * /rplidarNode/angle_end: 360.0
 * /rplidarNode/angle_start: 0.0
 * /rplidarNode/distance_max: 30.0
 * /rplidarNode/distance_min: 0.0
 * /rplidarNode/is_parted: False
 * /slam_gmapping/angularUpdate: 0.0436
 * /slam_gmapping/astep: 0.05
 * /slam_gmapping/base_frame: base_footprint
 * /slam_gmapping/delta: 0.05
 * /slam_gmapping/iterations: 5
 * /slam_gmapping/kernelSize: 3
 * /slam_gmapping/lasamplerange: 0.005
 * /slam_gmapping/lasamplestep: 0.005
 * /slam_gmapping/linearUpdate: 0.05
 * /slam_gmapping/llsamplerange: 0.01
 * /slam_gmapping/llsamplestep: 0.01
 * /slam_gmapping/lsigma: 0.075
 * /slam_gmapping/lskip: 0
 * /slam_gmapping/lstep: 0.05
 * /slam_gmapping/map_update_interval: 0.01
 * /slam_gmapping/maxRange: 5.0
 * /slam_gmapping/maxUrange: 4.0
 * /slam_gmapping/minimumScore: 30
 * /slam_gmapping/odom_frame: odom
 * /slam_gmapping/ogain: 3.0
 * /slam_gmapping/particles: 8
 * /slam_gmapping/resampleThreshold: 0.5
 * /slam_gmapping/sigma: 0.05
 * /slam_gmapping/srr: 0.01
 * /slam_gmapping/srt: 0.02
 * /slam_gmapping/str: 0.01
 * /slam_gmapping/stt: 0.02
 * /slam_gmapping/temporalUpdate: -1.0
 * /slam_gmapping/xmax: 5.0
 * /slam_gmapping/xmin: -5.0
 * /slam_gmapping/ymax: 4.0
 * /slam_gmapping/ymin: -4.0

NODES
  /
    base_to_camera (tf/static_transform_publisher)
    base_to_gyro (tf/static_transform_publisher)
    base_to_laser (tf/static_transform_publisher)
```

```
    base_to_link (tf/static_transform_publisher)
    joint_state_publisher (joint_state_publisher/joint_state_publisher)
    lsm10_v2 (lsm10_v2/lsm10_v2)
    mercury_robot (turn_on_mercury_robot/mercury_robot_node)
    robot_pose_ekf (robot_pose_ekf/robot_pose_ekf)
    robot_state_publisher (robot_state_publisher/robot_state_publisher)
    save_map (world_canvas_msgs/save)
    slam_gmapping (gmapping/slam_gmapping)

auto-starting new master
process[master]: started with pid [7403]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 34bcf3be-0606-11ef-8293-e8fb1c355a09
process[rosout-1]: started with pid [7414]
started core service [/rosout]
process[lsm10_v2-2]: started with pid [7421]
process[save_map-3]: started with pid [7422]
process[slam_gmapping-4]: started with pid [7427]
port = /dev/wheeltec_lidar, baud_rate = 460800
open_port /dev/wheeltec_lidar  OK !
process[mercury_robot-5]: started with pid [7434]
process[base_to_link-6]: started with pid [7437]
process[base_to_laser-7]: started with pid [7447]
[ INFO] [1714380923.257953390]: Data ready
[ INFO] [1714380923.271190031]: tringai_robot serial port opened
process[base_to_camera-8]: started with pid [7455]
process[base_to_gyro-9]: started with pid [7461]
process[joint_state_publisher-10]: started with pid [7464]
process[robot_state_publisher-11]: started with pid [7476]
process[robot_pose_ekf-12]: started with pid [7488]
[ INFO] [1714380923.815804921]: output frame: odom
[ INFO] [1714380923.822399818]: base frame: base_footprint
[ INFO] [1714380924.077547181]: Initializing Odom sensor
[ INFO] [1714380924.582680406]: Odom sensor activated
[ INFO] [1714380925.085858055]: Initializing Imu sensor
[ INFO] [1714380925.087556508]: Kalman filter initialized with odom measurement
[ INFO] [1714380925.091287721]: Imu sensor activated
[ INFO] [1714380925.206262818]: Laser is mounted upwards.
 -maxUrange 4 -maxUrange 5 -sigma     0.05 -kernelSize 3 -lstep 0.05 -lobsGain 3 -aste
 -srr 0.01 -srt 0.02 -str 0.01 -stt 0.02
 -linearUpdate 0.05 -angularUpdate 0.0436 -resampleThreshold 0.5
 -xmin -5 -xmax 5 -ymin -4 -ymax 4 -delta 0.05 -particles 8
[ INFO] [1714380925.222595401]: Initialization complete
update frame 0
update ld=0 ad=0
Laser Pose= 0.0368556 0.000564671 3.14
m_count 0
```

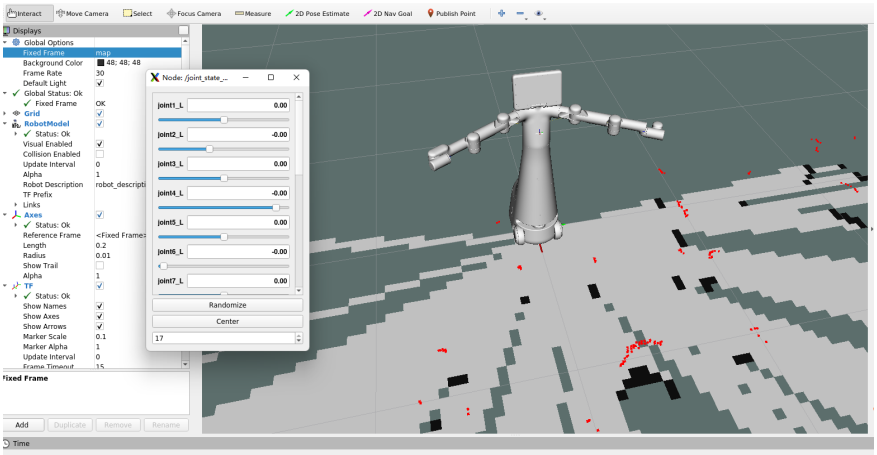
```
Registering First Scan
update frame 4089
update ld=0.0500112 ad=4.53872e-06
Laser Pose= 0.0383474 0.001601 3.14
```

2. Load URDF model

After the underlying communication program is started, the Mercury X1 URDF model is loaded through the launch file, a ROS1 environment terminal is opened, and then the command is run.

```
roslaunch turn_on_mercury_robot slider_control.launch
```

The effect diagram is as follows:



You can then control the movement of the joint model in rviz by dragging the slider.

3.Joint control

After the URDF model is successfully loaded, if you want the real Mercury X1 to move along with it, you need to open another ROS1 environment terminal and run the command:

Then run the command:

```
rosrun turn_on_mercury_robot slider_control.py
```

Please note: Since the robot arm will move to the current position of the model when the command is entered, please make sure that the model in rviz does not have mold penetration before you use the command Do not quickly drag the slider after connecting the robotic arm to prevent damage to the robotic arm

4 Chassis Control

After the joint control program is started, if you want the chassis car in the real Mercury X1 to run together, you need to start the keyboard control program of the chassis car, open a ROS1 environment terminal, and then run the command:

```
rosrun turn_on_mercury_robot mercury_keyboard.py
```

← Previous Page | Next Section →

ROS2 introduction

The predecessor of ROS2 is ROS, and ROS is the Robot Operating System (Robot Operating System). But ROS itself is not an operating system, but a software library and toolset. The emergence of Ros solved the communication problem of each component of the robot. Later, more and more robot algorithms were integrated into ROS. ROS2 inherited ROS, which is more powerful and better than ROS.

1 Design goals and features of ROS2

ROS2 has the historical mission of changing the era of intelligent robots. At the beginning of the design, it was considered to meet the needs of various robot applications.

- **Multi-Robot Systems:** In the future, robots will not be independent individuals, and communication and collaboration between robots are also required. ROS2 provides standard methods and communication mechanisms for the application of multi-robot systems.
- **Cross-platform:** Robot application scenarios are different, and the control platforms used will also be very different. In order to allow all robots to run ROS2, ROS2 can run on Linux, Windows, MacOS, and RTOS across platforms.
- **Real time:** Robot motion control and many behavior strategies require the robot to be real-time. For example, the robot must reliably detect pedestrians in front of it within 100ms, or complete kinematics and dynamics calculations within 1ms. ROS2 is a real-time like this Basic requirements are provided.
- **Productization:** A large number of robots have entered our lives, and there will be more and more in the future， ROS2 can not only be used in the robot research and development stage, but also can be directly installed in the product and go to the consumer market. This also poses a huge challenge to the stability and robustness of ROS2.
- **Project management:** Robot development is a complex system engineering. The project management tools and mechanisms for the whole process of design, development, debugging, testing, and deployment will also be reflected in ROS2, making it easier for us to develop a robot.

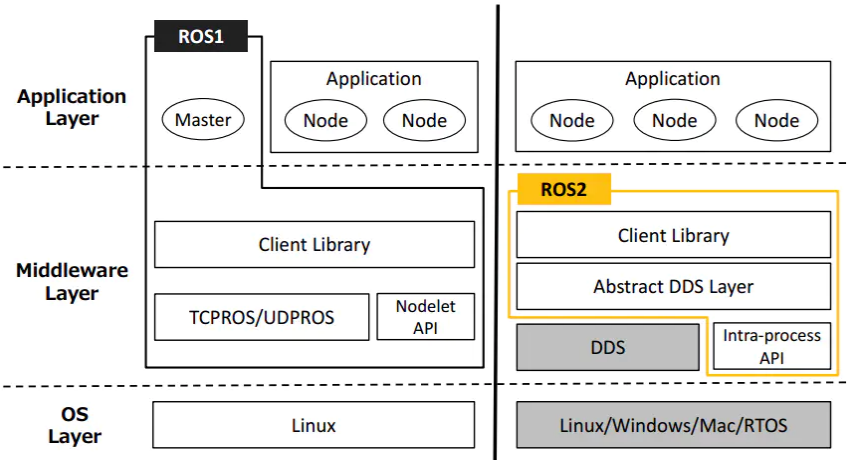
2 Release Version

The release version and maintenance cycle corresponding to ROS2 and Ubuntu.

ROS2 version	release date	Maintenance deadline	Ubuntu version
Dashing	2019.5	2021.5	Ubuntu 18.04 (Bionic Beaver)
Eloquent	2019.11	2020.11	Ubuntu 18.04 (Bionic Beaver)
Foxy	2020.6	2023.5	Ubuntu 20.04(Focal Fossa)
Galactic	2021.5	2022.11	Ubuntu 20.04(Focal Fossa)
Humble	2022.5	2027.5	Ubuntu 22.04(Jammy Jellyfish)

3 Comparison of ROS and ROS2

ROS2 redesigned the system architecture. The architecture changes between the two generations of ROS are as follows:



- **OS Layer:** In ROS2, it can be built on linux or other systems, even bare metal without an operating system.
- **Middleware Layer:** The communication system of ROS1 is based on TCPROS/UDPROS, while the communication system of ROS2 is based on DDS. DDS is a standard solution for data publishing/subscribing in distributed real-time systems.
- **Application Layer:** ROS1 relies on ROS Master, while in ROS2, a discovery mechanism called "Discovery" is used between nodes to help establish connections with each other.

ROS has designed a complete set of communication mechanisms (topics, services, parameters, actions) to simplify robot development. Through this mechanism, the various components of the robot can be connected. This

mechanism has designed a node called Ros Master, and the communication of all other components must go through the master node. Once the master node hangs up, it will cause the communication of the entire robot system to collapse! Therefore, the instability of Ros cannot be used to make some high-risk robots such as automatic driving. In addition, there are the following disadvantages:

- Communication based on TCP has poor real-time performance and high system overhead
- Unfriendly to python3 support
- Messaging mechanism is not compatible
- No encryption mechanism, low security

ROS2 first removes the master node that exists in ROS. After removing the master node, each node can discover each other through the DDS node, each node is equal, and can realize one-to-one, one-to-many, and many-to-many communication. After using DDS for communication, reliability and stability have been enhanced.

Compared with **ROS** that only supports Linux systems, **ROS2** also supports windows, mac, and even RTOS platforms

---

← Previous Section | Next Page →

Linux system environment

The system comes with Ubuntu (V-20.04) system and built-in **ROS2 Galactic** development environment. There is no need to build and manage it. You only need to update the `mercury_x1_ros2` package.

`mercury_x1_ros2` is a ROS2 package launched by Elephant Robot for its Mercury X1 series robotic arms.

ROS2 project address: http://github.com/elephantrobotics/mercury_x1_ros2

Robotic arm API driver library address:
https://github.com/elephantrobotics/pymycobot

1 Update mercury_x1_ros2 package

In order to ensure that users can use the latest official packages in a timely manner, you can enter the `/home/er` folder through the file manager, open the ROS2 environment terminal, and then run the command update:

```
# Clone the code on github
git clone https://github.com/elephantrobotics/mercury_x1_ros2.git # Please check the a
cd ~/mercury_x1_ros2      # Back to work area
colcon build --symlink-install # Build the code in the workspace, --symlink-install: A
source install/setup.bash # add environment variables



# Compile the function package separately:
# If you only compile "turn_on_mercury_robot", you need to execute the command:
colcon build --packages-select turn_on_mercury_robot
```

Note: If the `mercury_x1_ros2` folder already exists in the `/home/er` directory, you need to delete the original `mercury_x1_ros2` before executing the above command. Among them, `er` in the directory path is the user name of the system.

So far, the ROS2 environment construction has been completed.You can learn the basics of ROS2 or ROS2 use cases

---

← Previous Page | Next Page →

1 ROS2 project structure

1.1 colcon workspace

The colocn workspace is the directory where software packages are created, modified, and compiled. Colcon's workspace can be intuitively described as a warehouse, which contains various ROS project projects to facilitate system organization, management and calling.

- **Create workspace**:

```
mkdir -p ~/colcon_ws/src # Create folder
cd ~/colcon_ws/ # Enter the folder
colcon build # Build the code in the workspace.
```

Note: colcon supports option `--symlink-install`. This allows for faster iteration by changing installed files by changing files in the source space (such as Python files or other uncompiled resources). Avoid the need to recompile every time you modify your python script.

```
colcon build --symlink-install
```

A ROS workspace is a directory with a particular structure. Commonly there is a `src` subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically the directory starts otherwise empty.

colcon does out of source builds. By default it will create the following directories as peers of the `src` directory:

```
src/: colcon package for ROS2 (source code package)

build/: The location where intermediate files are stored. For each package, a subfolde

install/: The installation location of each package. By default, each package will be

log/: Contains various logging information about each colcon call.
```

The directory structure of a ROS2 workspace is as follows:

```
WorkSpace --- Customized workspace.
     |--- build: The directory where intermediate files are stored. A separate subdire
     |--- install: Installation directory, a separate subdirectory will be created for
     |--- log: Log directory, used to store log files.
     |--- src: Directory used to store function package source code.
         |-- C++ function package
             |-- package.xml: package information, such as: package name, version, aut
             |-- CMakeLists.txt: Configure compilation rules, such as source files, de
             |-- src: C++ source file directory.
             |-- include: header file directory.
             |-- msg: message interface file directory.
             |-- srv: Service interface file directory.
             |-- action: action interface file directory.
         |-- Python function package
             |-- package.xml: package information, such as: package name, version, aut
             |-- setup.py: similar to CMakeLists.txt of C++ function package.
             |-- setup.cfg: Function package basic configuration file.
             |-- resource: resource directory.
             |-- test: stores test-related files.
             |-- Directory with the same name of the function package: Python source f
```

1.2 ROS2 package

Package is not only a software package on Linux, but also the basic unit of colcon compilation. The object we use `colcon build` to compile is each ROS2 package.

Create your own package:

- The command syntax for creating a software package using Python is:

```
ros2 pkg create --build-type ament_python <package_name>
```

- For example:

```
ros2 pkg create --build-type ament_python --node-name my_node my_package
```

2 Basic tool commands

In this chapter, you will learn about the common command tools of ROS2.

2.1 Topics

ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages. Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

Specific reference: Official Tutorials

- **topics help**

```
ros2 topics -h
```

- **Start turtlesim and keyboard control**

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

- **Node Relationship Diagram**

```
rqt_graph
```

- **Learn about topic-related commands**

```
ros2 topics -h
```

- **topics list**

```
ros2 topic list
ros2 topic list -t # Display the corresponding message type
```

- **View topic content**

```
ros2 topic echo <topic_name>
ros2 topic echo /turtle1/cmd_vel
```

- **Display topic-related information, type**

```
ros2 topic info <topic_name>
# Output /turtle1/cmd_vel topic related information
ros2 topic info /turtle1/cmd_vel
```

- **Display interface related information**

```
ros2 interface show <msg_type>
# Output geometry_msgs/msg/Twist interface related information
ros2 interface show geometry_msgs/msg/Twist
```

- **Issue an order**

```
ros2 topic pub <topic_name> <msg_type> '<args>'
# Issue speed command
ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0
# Issue speed commands at a certain frequency
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y:
```

- **See how often topics are posted**

```
ros2 topic hz <topic_name>
# Output /turtle1/cmd_vel publish frequency
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y:
```

2.2 Nodes

Each node in ROS should be responsible for a single, module purpose (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc). Each node can send and receive data to other nodes via topics, services, actions, or parameters. A full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes.

Specific reference: Official Tutorials

- **nodes help**

```
ros2 nodes -h
```

- **Start turtlesim and keyboard control**

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

- **View the node list**

```
ros2 node list
```

- **View Node Relationship Diagram**

```
rqt_graph
```

- **Remapping**

```
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
ros2 node list
```

- **View node information**

```
ros2 node info <node_name>
ros2 node info /my_turtle
```

2.3 Services

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model, versus topics' publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.

Specific reference: Official Tutorials

- **services help**

```
ros2 service -h
```

- **Start turtlesim and keyboard control**

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

- **View the service list**

```
ros2 service list
# Display service list and message type
ros2 service list -t
```

- **View the message types received by the service**

```
ros2 service type <service_name>
ros2 service type /clear
```

- **Find services that use a certain message type**

```
ros2 service find <type_name>
ros2 service find std_srvs/srv/Empty
```

- **View Service Message Type Definitions**

```
ros2 interface show <type_name>.srv
ros2 interface show std_srvs/srv/Empty.srv
```

- **Call the service command to clear the walking track**

```
ros2 service call <service_name> <service_type>
ros2 service call /clear std_srvs/srv/Empty
```

- **Spawn a new turtle**

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: 'turtle2'
```

2.4 Parameters

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists. In ROS 2, each node maintains its own parameters. For more background on parameters, please see the concept document.

Specific reference: Official Tutorials

- **parameters help**

```
ros2 param -h
```

- **Start turtlesim and keyboard control**

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

- **View service list**

```
ros2 param list
```

- **Get the parameter value**

```
ros2 param get <node_name> <parameter_name>
ros2 param get /turtlesim background_g
```

- **Set parameter values**

```
ros2 param set <node_name> <parameter_name> <value>
ros2 param set /turtlesim background_r 150
```

- **Export parameter values**

```
ros2 param dump <node_name>
ros2 param dump /turtlesim
```

- **Import parameters independently**

```
ros2 param load <node_name> <parameter_file>
ros2 param load /turtlesim ./turtlesim.yaml
```

- **Start the node and import parameters at the same time**

```
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
ros2 run turtlesim turtlesim_node --ros-args --params-file ./turtlesim.yaml
```

2.5 Actions

Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions are preemptable (you can cancel them while executing). They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result.

Specific reference: Official Tutorials

- **action help**

```
ros2 action -h
```

- **Start turtlesim and keyboard control**

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

Press G|B|V|C|D|E|R|T to achieve rotation, press F to cancel

- **View the server and client of the node action**

```
ros2 node info /turtlesim
```

- **View action list**

```
ros2 action list
ros2 action list -t # show action type
```

- **view action info**

```
ros2 action info <action>
ros2 action info /turtle1/rotate_absolute
```

- **View action message content**

```
ros2 interface show turtlesim/action/RotateAbsolute
```

- **Send action target information**

```
ros2 action send_goal <action_name> <action_type>
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta
# With feedback information
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta
```

2.6 RQt

RQt is a graphical user interface framework that implements various tools and interfaces in the form of plugins. One can run all the existing GUI tools as dockable windows within RQt! The tools can still run in a traditional standalone method, but RQt makes it easier to manage all the various windows in a single screen layout.

Specific reference: Official Tutorials

You can run any RQt tools/plugins easily by:

```
rqt
```

- **rqt help**

```
rqt -h
```

- **Start turtlesim and keyboard control**

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

- Action Type Browser: / Plugins -> Actions ->Action Type Browser
- parameter reconfiguration: / Plugins -> configuration ->Parameter Reconfigure
- Node grap: /Node Graph
- control steering: /Plugins -> Robot Tools -> Robot Steering
- service invocation: /Plugins -> Services -> Service Caller

- Service Type Browser: Plugins -> Services -> Service Type Browser
- message release: Plugins -> Topics -> Message Publisher
- Message Type Browser: Plugins -> Topics -> Message Type Browser
- topic list: Plugins -> Topics -> Topic Monitor
- draw a graph: Plugins -> Visualization -> Plot
- **View logs: rqt_console**

```
ros2 run rqt_console rqt_console
ros2 run turtlesim turtlesim_node
ros2 topic pub -r 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0
```

2.7 TF2

tf2 is the transform library, which lets the user keep track of multiple coordinate frames over time. tf2 maintains the relationship between coordinate frames in a tree structure buffered in time and lets the user transform points, vectors, etc. between any two coordinate frames at any desired point in time.

Specific reference: Official Tutorials

Let's start by installing the demo package and its dependencies.

```
sudo apt-get install ros-foxy-turtle-tf2-py ros-foxy-tf2-tools ros-foxy-tf-transformat
```

- **follow**
- launch starts 2 little turtles, the first little turtle automatically follows the second one

```
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

- Control the movement of the first little turtle through the keyboard

```
ros2 run turtlesim turtle_teleop_key
```

- **View TF tree**

```
ros2 run tf2_tools view_frames.py
evince frames.pdf
```

- **View the relationship between two coordinate systems**

```
ros2 run tf2_ros tf2_echo [reference_frame] [target_frame]
ros2 run tf2_ros tf2_echo turtle2 turtle1
```

- **View TF relationships on rviz**

```
ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz
```

2.8 URDF

URDF is the Unified Robot Description Format for specifying robot geometry and organization in ROS.

Specific reference: Official Tutorials

- **Complete syntax**

```
<robot>
    # describe:
    # Parameters: name=""
    #  Child node:
        <link>
            # Description:
            # Parameters: name=""
            # Child node:
                <visual>
                    # describe:
                    # Parameters:
                    # child nodes:
                        <geometry>
                            # description
                            # parameters
                            # Child node:
                                <cylinder />
                                    # Description:
                                    # Parameters:
                                        # length="0.6"
                                        # radius="0.2"
                                <box />
                                    # description
                                    # Parameters:size="0.6 0.1 0.2"
                                <mesh />
                                    #  Description
                                    #Parameters: filename="package://urdf_tutorial/mes
                        <collision>
                            # Description: collision element, prioritized
                            # parameters
                            # child node
                                <geometry>
                        <inertial>
                            # description
                            # parameters
                            # Child nodes:
                                <mass />
                                    # description: mass
                                    # Parameters: value=10
                                <inertia />
                                    # Description: Inertia
                                    # Parameters: i+"Cartesian product of xyz" (9 in t
                        <origin />
                            # Description:
                            # Parameters:
                                # rpy="0 1.5 0"
                                # xyz="0 0 -0.3"
                        <material />
```

```
                        # Description
                        # Parameters: name="blue"
        <joint>
            # Description
            # Parameters:
                # name=""
                # type=""
                    # fixed
                    # prismatic
            # child node
                <parent />
                    # Description
                    # Parameters: link=""
                <child />
                    # Description:
                    # Parameters: link=""
                <origin />
                    # Description:
                    # Parameters: xyz="0 -0.2 0.25"
                <limit />
                    # Description
                    # Parameters:
                        # effort="1000.0"    maximum effort
                        # lower="-0.38"      Joint upper limit (radians)
                        # upper="0"          Joint lower limit (radians)
                        # velocity="0.5"     Maximum velocity
                <axis />
                    # Description:  Press ? axis rotation
                    # Parameters: xyz="0 0 1", along the Z axis
        <material>
            # Description:
            # Parameters: name="blue"
            # child node:
                <color />
                    # description:
                    # Parameters: rgba="0 0 0.8 1"
```

- **Install dependent libraries**

```
sudo apt install ros-foxy-joint-state-publisher-gui ros-foxy-joint-state-publisher
sudo apt install ros-foxy-xacro
```

- **Download the source code**

```
cd ~/dev_ws
git clone -b ros2 https://github.com/ros/urdf_tutorial.git src/urdf_tutorial
```

- **Compiling the source code**

```
colcon build --packages-select urdf_tutorial
```

- **Running the example**

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/01-myfirst.urdf
```

2.9 Launch

The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them, and ROS-specific conventions which make it easy to reuse components throughout the system by giving them each a different configuration. It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.

Launch files written in Python, XML, or YAML can start and stop different nodes as well as trigger and act on various events.

Specific reference: Official Tutorials

Setup

Create a new directory to store your launch files:

```
mkdir launch
```

Writer the launch file

Let's put together a ROS 2 launch file using the turtlesim package and its executables. As mentioned above.

Copy and paste the complete code into the launch/turtlesim_mimic_launch.py file:

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

Run the ros2 launch file

To run the launch file created above, enter into the directory you created earlier and run the following command:

The syntax format is:

```
ros2 launch <package_name> <launch_file_name>
```

```
cd launch
ros2 launch turtlesim_mimic_launch.py
```

- **launch help**

```
ros2 launch -h
```

- **running node**

```
ros2 launch turtlesim multisim.launch.py
```

- **Check the parameters of the launc file**

```
ros2 launch turtlebot3_fake_node turtlebot3_fake_node.launch.py -s
ros2 launch turtlebot3_fake_node turtlebot3_fake_node.launch.py --show-arguments
ros2 launch turtlebot3_bringup robot.launch.launch.py -s
```

- **Run the launch file with parameters**

```
ros2 launch turtlebot3_bringup robot.launch.launch.py usb_port:=/dev/opencr
```

- **Run the node and debug**

```
ros2 launch turtlesim turtlesim_node.launch.py -d
```

- **Only output node description**

```
ros2 launch turtlesim turtlesim_node.launch.py -p
```

- **running components**

```
ros2 launch composition composition_demo.launch.py
```

2.10 Run

run is used to run a single node, component program

- **run help**

```
ros2 run -h
```

- **running node**

```
ros2 run turtlesim turtlesim_node
```

- **Run node with parameters**

```
ros2 run turtlesim turtlesim_node --ros-args -r __node:=turtle2 -r __ns:=/ns2
```

- **Run component container**

```
ros2 run rclcpp_components component_container
```

- **running components**

```
ros2 run composition manual_composition
```

2.11 Package

A package can be considered a container for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.

Package creation in ROS 2 uses ament as its build system and colcon as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

Specific reference: Official Tutorials

Creating a workspace

Create a new directory for every new workspace. The name doesn't matter, but it is helpful to have it indicate the purpose of the workspace. Let's choose the directory name ros2_ws, for "development workspace":

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

- **pkg help**

```
ros2 pkg -h
```

- **List Feature Packs**

```
ros2 pkg executable turtlesim
```

- **Output a function package executable program**

```
ros2 pkg executable turtlesim
```

- **Create a Python package**

Make sure you are in the src folder before running the package creation command.

```
cd ~/ros2_ws/src
```

The command syntax for creating a new package in ROS 2 is:

```
ros2 pkg create --build-type ament_python <package_name>
# you will use the optional argument --node-name which creates a simple Hello World ty
ros2 pkg create --build-type ament_python --node-name my_node my_package
```

- **Build a package**

Putting packages in a workspace is especially valuable because you can build many packages at once by running colcon build in the workspace root. Otherwise, you would have to build each package individually.

```
# Return to the root of your workspace:
cd ~/ros2_ws
# Now you can build your packages:
colcon build
```

- **Source the setup file**

To use your new package and executable, first open a new terminal and source your main ROS 2 installation.

Then, from inside the ros2_ws directory, run the following command to source your workspace:

```
source install/setup.bash
```

Now that your workspace has been added to your path, you will be able to use your new package's executables.

- **Use the package**

To run the executable you created using the --node-name argument during package creation, enter the command:

```
ros2 run my_package my_node
```

← Previous Page | Next Page →

Brief introduction and use of rviz2

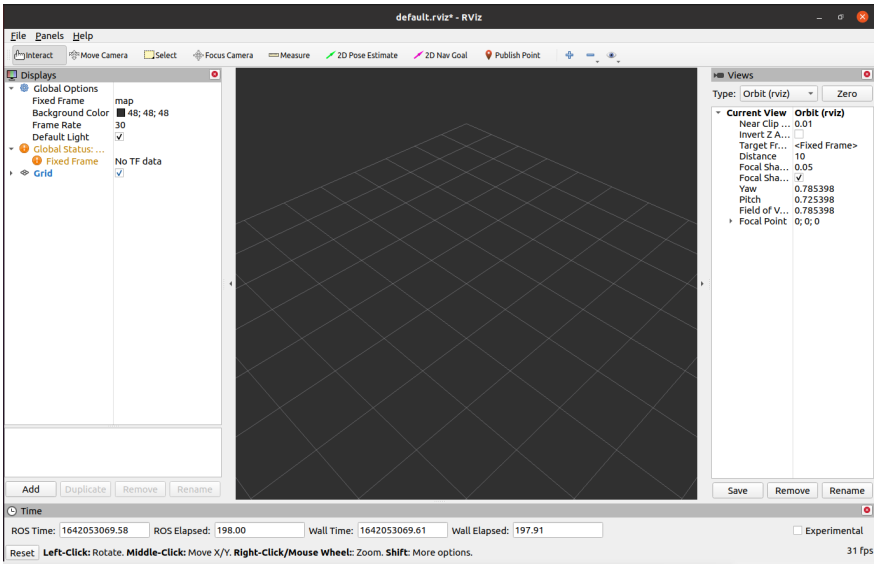
Rviz2 is a visualization tool for displaying messages in the robot environment, providing a 3D perspective to view the robot's status and activities. It can help developers better understand the current status and activities of the robot, as well as other visual messages. Rviz2 provides a series of visualization tools that can help developers better understand the status and activities of robots, such as visual coordinate systems, laser scanning messages, point cloud messages, robot models, etc. Using Rviz2, robotic systems can be easily viewed and debugged to better achieve robotic goals.

1 Introduction to rviz2

Open the ROS2 environment terminal, enter the command to open rviz2:

```
ros2 run rviz2 rviz2
# or
rviz2
```

Open rviz2 and display the following interface:



1.1 Introduction of each area

- On the left is the list of monitors, a monitor is something that draws something in the 3D world and may have some options available in the display list. Including functions such as adding, deleting, copying, renaming plug-ins, displaying plug-ins, and setting plug-in properties.
- Above is the toolbar, which allows users to use various function buttons to select tools with multiple functions

- The middle part is the 3D view: it is the main screen where various data can be viewed in 3D. The background color, fixed frame, grid, etc. of the 3D view can be set in detail in the Global Options and Grid items displayed on the left.
- Below is the time display area, including system time and ROS time.
- The right side is the observation angle setting area, and different observation angles can be set.

We only give a rough introduction in this part. If you want to know more detailed content, you can go to the user guide to view it.

If you want to know more information about rviz, you can go to the official documentation to view it.

← Previous Page | Next Page →

Mercury X1 Controls

Here we mainly introduce how to control the movement of Mercury X1 through a series of related instructions.

1. Chassis underlying communication

First, start the underlying communication and map construction program of the chassis, and load the URDF model of Mercury X1.

Open the ROS2 environment terminal and run the command:

```
ros2 launch slam_gmapping slam_gmapping.launch.py
```

You can then control the movement of the joint model in rviz2 by dragging the slider.

2.Joint control

After the URDF model is successfully loaded, if you want the real Mercury X1 arms to move with it, you need to open another ROS2 environment terminal and run the command:

```
ros2 run mercury_x1_control slider_control
```

Please note: Since the robot arm will move to the current position of the model when the command is entered, please make sure that the model in rviz does not have mold penetration before you use the command Do not quickly drag the slider after connecting the robotic arm to prevent damage to the robotic arm

3 Chassis Control

After the joint control program is started, if you want the chassis car in the real Mercury X1 to move with it, you need to start the keyboard control program of the chassis car, open a ROS2 environment terminal, and then run the command:

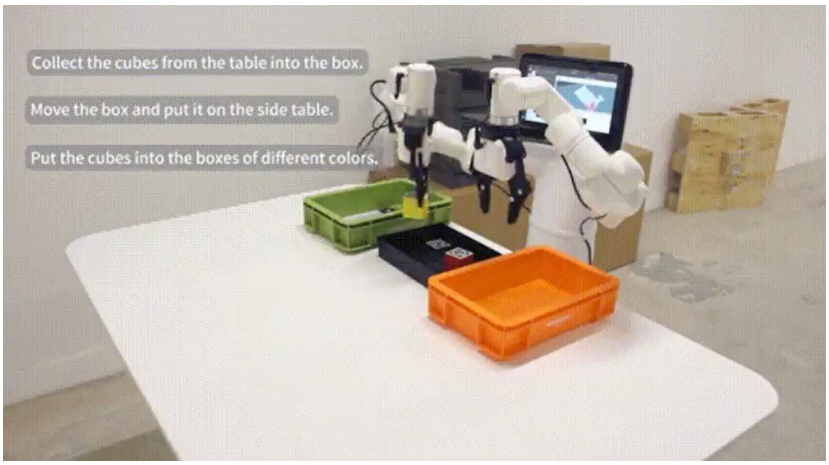
```
ros2 run mercury_x1_control mercury_keyboard
```

← Previous Page | Next Section →

6 Robot Usage Scenarios

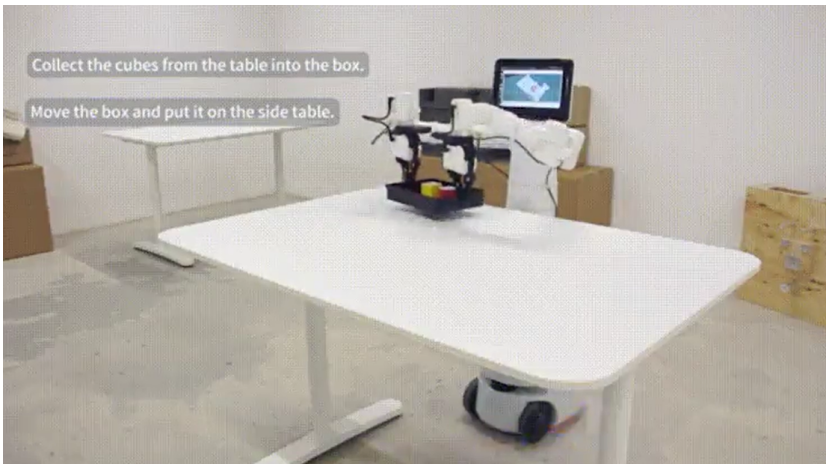
This chapter presents classic robotic arm usage scenarios to showcase the product's applications in representative settings. This includes typical applications of the Mercury X1 wheeled humanoid robot in various fields, highlighting the product's versatility and applicability. Through these cases, users can gain an in-depth understanding of the robotic arm's flexibility and efficiency in practical applications, providing references for their use in specific scenarios.

- QR Code Recognition and Grasping
  The Mercury X1 wheeled humanoid robot's arm is equipped with a 2D camera and gripper to achieve QR code recognition and grasping. The QR code uses stag code, and the stag.detectMarkers function is used to obtain the stag code's camera coordinates. The coordinates are then transformed to the base coordinates using matrix transformation, and the coordinate control function is called for grasping.

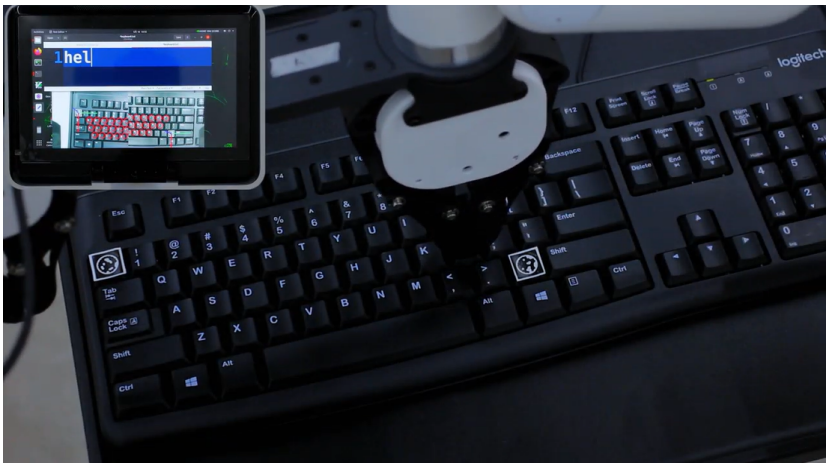
  

- Spatial Movement Case
  The chassis can be remotely controlled using a keyboard, or the mercury_ros_api library can be used for movement control and navigation of the base.

- Mobile Grasping Case
  This case combines the previous two cases, using navigation to move the Mercury X1 wheeled humanoid robot to a designated location, and then achieving grasping through stag vision recognition.



- Keyboard typing case
  This case involves identifying the stage codes in the upper left and upper right corners, calculating the two-dimensional coordinates, three-dimensional coordinates, scaling ratios, and rotation angles of all key positions, and displaying the key positions on the screen. Clicking is achieved by inputting specified keys.



Preparations for Using Vision Recognition

1. Materials and Power-On Preparation

1.1 Material Preparation

Materials needed for vision recognition tracking include: 1) Mercury series robots 2) Vision sensors and stag code accessories 3) Gripper and related accessories

1.2 Power-On Preparation

Connect the power adapter to the robotic arm, or use the built-in power supply, and move the robot to an open area to prevent collisions.

Power interface at the rear of the base

Power switch location

Emergency stop switch location

Pressing the emergency stop switch will immediately stop the robot. To reset, rotate the emergency stop switch clockwise, and it will automatically pop up. Then, power on to resume control.

Click the power switch and ensure the emergency stop is reset to start using the robotic arm.

2. Mercury System Debugging

2.1 Powering On the System

The robotic arm supports multiple ways to power on. This document only introduces using Python commands to control the power on. Enter the following commands in the Terminal to power on the robot:

```
python
from pymycobot import Mercury
mr = Mercury("/dev/right_arm")  #connect to Mercury right arm
ml = Mercury("/dev/left_Arm")  #connect to Mercury left arm
ml.power_on()
mr.power_on()
```

Before sending the command, ensure the emergency stop switch is reset. When you hear the **joint brake sound**, it means the power-on is successful! If you do not hear the brake sound, use the following commands to repeatedly power on and off:

First, send the power-off command

```
ml.power_off()
mr.power_off()
```

Wait a few seconds and then send the power-on command

```
ml.power_on()
mr.power_on()
```

If you still do not hear the brake sound, contact an engineer for assistance.

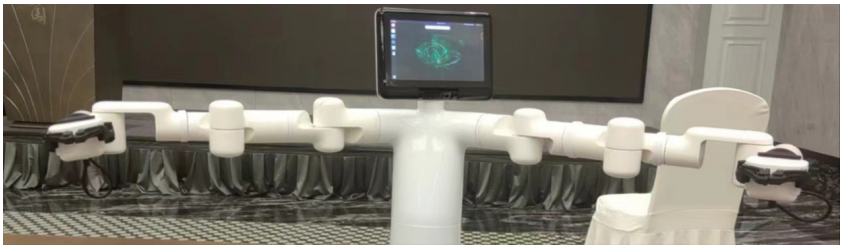
2.2 Robotic Arm Zero Calibration

Before using the robotic arm, ensure the zero position is correct. You can calibrate the zero position using the following method:

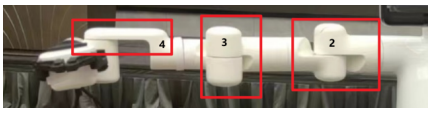
1) Use the relax command to release the joint motors (**Note! After relaxing, hold the joints to prevent the robotic arm from falling and getting damaged!**)

```
ml.release_all_servos()
mr.release_all_servos()
```

2) Drag the robotic arm back to the zero position



You can determine the zero position through the following details:



3) Send the command to lock the motors

```
ml.focus_all_servos()
mr.focus_all_servos()
```

4) Check if the zero position is correct

Enter the command to get the joint angles and query the current position

```
ml.get_angles()
mr.get_angles()
```

If the returned joint angles are close to [0, 0, 0, 0, 90, 0], the zero position is considered correct.

5) Zero Calibration

If the joint angles read at the zero position differ significantly from [0, 0, 0, 0, 90, 0], calibrate the joint zero position

```
for i in range(1,7):
  ml.set_servo_calibration(i)
  mr.set_servo_calibration(i)
```

After calibration, read the joint information. If the return is [0, 0, 0, 0, 90, 0], the calibration is successful.

```
ml.get_angles()
mr.get_angles()
```

2.3 Robotic Arm Movement

You can verify the robotic arm's movement function using the following methods:

1) Ensure the zero position is correct, then use the return to zero command to control the robotic arm back to the zero position:

```
ml.over_limit_return_zero()
mr.over_limit_return_zero()
```

2) Send the joint movement command to verify if the robotic arm movement is normal

```
ml.send_angle(1,20,10)
mr.send_angle(1,20,10)
```

At this point, you should observe the robotic arm's joint 1 rotating. After the movement is complete, use the command to read the joint information

```
ml.get_angles()
mr.get_angles()
```

If the returned value for joint 1 is 20°, the movement is considered normal.

2.4 Exception Handling

If the robotic arm movement is abnormal in 2.3, use the following methods to observe the exception information:

1) Get motor exception information

```
ml.get_robot_status()
mr.get_robot_status()
```

2) Get movement exception information

```
ml.get_error_information()
mr.get_error_information()
```

Provide the returned information from the above interfaces to the engineer for processing.

3. Camera and Gripper Installation Method

The installation method of the camera and gripper corresponds to the hand-eye matrix of vision recognition. If the installation method changes, re-calibrate the hand-eye matrix!

First, move the robotic arm back to the zero position:

```
ml.over_limit_return_zero()
mr.over_limit_return_zero()
```

Install the gripper and camera according to the following diagram:

4. Camera Parameter Configuration

4.1 Find Camera Number

Since Mercury has multiple vision sensors, you need to find the corresponding numbers for the left and right arm cameras before configuring the camera parameters:

1) First, initialize the camera and fill in the numbers 0-6 in the Camera_ID field in sequence

```
obj = camera_detect(Camera_ID = 0, mark_size, mtx, dist, direction)
```

```
obj = camera_open_loop()
```

Turn on the camera, observe the camera screen, and mark the corresponding number of the camera. If the camera cannot be turned on, change the Camera_ID and repeat this operation.

2) Change the Camera_ID and record the camera number in sequence according to step 1

```
obj = camera_detect(Camera_ID = 1, mark_size, mtx, dist, direction)
```

```
obj = camera_open_loop()
```

3) Modify the corresponding numbers of the left and right arm cameras in the following positions of the py script

`ml_obj = camera_detect(Camera_ID = left, mark_size, mtx, dist, direction)`

`mr_obj = camera_detect(Camera_ID = right, mark_size, mtx, dist, direction)`

4.2 Camera Intrinsic Parameter Calibration

5. Gripper Function Debugging

Use the following interfaces to calibrate the gripper zero position:

1) Gripper Initialization

```
ml.set_gripper_mode(0)
mr.set_gripper_mode(0)
```

2) Relax the Gripper

```
ml.set_gripper_enabled(0)
mr.set_gripper_enabled(0)
```

Manually open the gripper to the maximum.

3) Set the Gripper Zero Position

```
ml.set_gripper_calibration()
mr.set_gripper_calibration()
```

4) Control the Gripper to Close

```
ml.set_gripper_value(0,100)
mr.set_gripper_value(0,100)
```

At this point, you should observe the gripper closing quickly.

Gripper Exception Troubleshooting:

1) The gripper does not move - repeat steps 1-4. **If it still does not move, contact our engineer.** 2) Step 4 does not fully close - in step 2, do not open to the maximum, then set the zero position.

6. System Comprehensive Test

Execute the following py script, and you should observe the following phenomena:

1) The robotic arm returns to the zero position. 2) The gripper opens and then closes. 3) The robotic arm moves to the following position. 4) The gripper opens and closes.

If the phenomena do not match the above, repeat the movement test and gripper test chapters.

Sure! Let's continue with the comprehensive test script.

```
from pymycobot import *
import numpy as np
import time

mr = Mercury("/dev/right_arm")  #connect to Mercury right arm
ml = Mercury("/dev/left_Arm")  #connect to Mercury left arm

ml.over_limit_return_zero()
mr.over_limit_return_zero()
time.sleep(10)

ml.set_gripper_mode(0)
mr.set_gripper_mode(0)
time.sleep(1)
ml.set_gripper_value(100,10)
mr.set_gripper_value(100,10)
time.sleep(3)
ml.set_gripper_value(0,10)
mr.set_gripper_value(0,10)
time.sleep(3)

ml.send_angles([45, 20, -45, 45, 120, 45], 10)
mr.send_angles([-45, 20, 45, 45, 120, -45], 10)
time.sleep(5)
la = ml.get_angles()
ra = mr.get_angles()
print("left", la)
print("right", ra)

ml.set_gripper_value(100,10)
mr.set_gripper_value(100,10)
time.sleep(3)
ml.set_gripper_value(0,10)
mr.set_gripper_value(0,10)
time.sleep(3)
```

← Previous Chapter | Next Page →

QR Code Recognition and Grasping

Case video

Using the `camera_detect` Vision Recognition Package

After completing the preparations in Chapters 1-6, you can achieve vision recognition by calling a few simple interfaces from the `camera_detect` library of Mercury.

Interface Introduction:

```
obj = camera_detect(Camera_ID, mark_size, mtx, dist, direction)
```

- **Parameters:**
  - `camera_ID` : Camera ID
  - `mark_size` : Stag code side length in mm
  - `direction` : 0: Left arm, 1: Right arm
- **Function:** Initialize the camera and robotic arm

```
obj.camera_open_loop()
```

- **Function:** Display the camera screen

```
obj.stag_identify_loop()
```

- **Function:** Display the camera screen and identify and print the stag code's **camera coordinates**

```
coords = obj.stag_robot_identify(direction)
```

- **Parameters:**
  - `direction` : 0: Left arm, 1: Right arm
  - `return` : Stag code's robotic arm coordinates
- **Function:** Calculate the stag code's **robotic arm coordinates**

`obj.vision_trace(catch_mode, direction)`

- **Parameters:**
  - `catch_mode` : 0: Horizontal grasping, 1: Vertical grasping
  - `direction` : 0: Left arm, 1: Right arm
- **Function:** Display the camera screen and identify and print the stag code's **camera coordinates**

Usage Example

```
import numpy as np
from pymycobot import Mercury
# Import the vision recognition stag package
from camera_detect import camera_detect

if __name__ == "__main__":
    camera_params = np.load("camera_params.npz")  # Camera configuration file
    mtx, dist = camera_params["mtx"], camera_params["dist"]
    ml = Mercury("/dev/left_arm")  # Set left arm port
    mr = Mercury("/dev/right_arm")  # Set right arm port
    arm = 0         # 0: Left arm, 1: Right arm
    catch_mode = 0  # 0: Horizontal grasping, 1: Vertical grasping
    # Camera ID needs to be changed according to the actual situation
    left_camera_id = 3
    right_camera_id = 6
    stag_size = 32
    # Create vision recognition objects for left and right arms
    ml_obj = camera_detect(left_camera_id, stag_size, mtx, dist, arm)
    mr_obj = camera_detect(right_camera_id, stag_size, mtx, dist, arm)
    # Move the left and right arms to the horizontal observation position for grasping
    mr_obj.vision_trace(catch_mode, mr)
    ml_obj.vision_trace(catch_mode, ml)
```

Without Using the `camera_detect` Vision Recognition Package

```
from uvc_camera import UVCCamera
from marker_utils import *
import stag

# Gripper tool length
Tool_LEN = 175
# Distance from camera center to flange
Camera_LEN = 78
np.set_printoptions(suppress=True, formatter={'float_kind': '{:.2f}'.format})
# Camera configuration file
camera_params = np.load("src/camera_params.npz")
mtx, dist = camera_params["mtx"], camera_params["dist"]
# QR code size
MARKER_SIZE = 32
# Set left arm port
ml = Mercury("/dev/left_arm")

# Convert rotation matrix to Euler angles
def CvtRotationMatrixToEulerAngle(pdtRotationMatrix):
    pdtEulerAngle = np.zeros(3)
    pdtEulerAngle[2] = np.arctan2(pdtRotationMatrix[1, 0], pdtRotationMatrix[0, 0])
    fCosRoll = np.cos(pdtEulerAngle[2])
    fSinRoll = np.sin(pdtEulerAngle[2])
    pdtEulerAngle[1] = np.arctan2(-pdtRotationMatrix[2, 0],
                                  (fCosRoll * pdtRotationMatrix[0, 0]) + (fSinRoll * p
    pdtEulerAngle[0] = np.arctan2((fSinRoll * pdtRotationMatrix[0, 2]) - (fCosRoll * p
                                  (-fSinRoll * pdtRotationMatrix[0, 1]) + (fCosRoll *
    return pdtEulerAngle

# Convert Euler angles to rotation matrix
def CvtEulerAngleToRotationMatrix(ptrEulerAngle):
    ptrSinAngle = np.sin(ptrEulerAngle)
    ptrCosAngle = np.cos(ptrEulerAngle)
    ptrRotationMatrix = np.zeros((3, 3))
    ptrRotationMatrix[0, 0] = ptrCosAngle[2] * ptrCosAngle[1]
    ptrRotationMatrix[0, 1] = ptrCosAngle[2] * ptrSinAngle[1] * ptrSinAngle[0] - ptrSi
    ptrRotationMatrix[0, 2] = ptrCosAngle[2] * ptrSinAngle[1] * ptrCosAngle[0] + ptrSi
    ptrRotationMatrix[1, 0] = ptrSinAngle[2] * ptrCosAngle[1]
    ptrRotationMatrix[1, 1] = ptrSinAngle[2] * ptrSinAngle[1] * ptrSinAngle[0] + ptrCo
    ptrRotationMatrix[1, 2] = ptrSinAngle[2] * ptrSinAngle[1] * ptrCosAngle[0] - ptrCo
    ptrRotationMatrix[2, 0] = -ptrSinAngle[1]
    ptrRotationMatrix[2, 1] = ptrCosAngle[1] * ptrSinAngle[0]
    ptrRotationMatrix[2, 2] = ptrCosAngle[1] * ptrCosAngle[0]
    return ptrRotationMatrix

# Convert coordinates to homogeneous transformation matrix (x, y, z, rx, ry, rz) in ra
def Transformation_matrix(coord):
    position_robot = coord[:3]
```

```
    pose_robot = coord[3:]
    # Convert Euler angles to rotation matrix
    RBT = CvtEulerAngleToRotationMatrix(pose_robot)
    PBT = np.array([[position_robot[0]],
                    [position_robot[1]],
                    [position_robot[2]]])
    temp = np.concatenate((RBT, PBT), axis=1)
    array_1x4 = np.array([[0, 0, 0, 1]])
    # Concatenate the two arrays row-wise
    matrix = np.concatenate((temp, array_1x4), axis=0)
    return matrix

def Eyes_in_hand_left(coord, camera):
    # Camera coordinates
    Position_Camera = np.transpose(camera[:3])
    # Robotic arm coordinate matrix
    Matrix_BT = Transformation_matrix(coord)
    # Hand-eye matrix
    Matrix_TC = np.array([[0, -1, 0, Camera_LEN],
                          [1, 0, 0, 0],
                          [0, 0, 1, -Tool_LEN],
                          [0, 0, 0, 1]])
    # Object coordinates (camera system)
    Position_Camera = np.append(Position_Camera, 1)
    # Object coordinates (base coordinate system)
    Position_B = Matrix_BT @ Matrix_TC @ Position_Camera
    return Position_B

# Wait for the robotic arm to finish moving
def waitl():
    time.sleep(0.2)
    while (ml.is_moving()):
        time.sleep(0.03)

# Get object coordinates (camera system)
def calc_markers_base_position(corners: NDArray, ids: T.List, marker_size: int, mtx: N
    if len(corners) == 0:
        return []
    # Get object rotation and translation vectors through QR code corners
    rvecs, tvecs = solve_marker_pnp(corners, marker_size, mtx, dist)
    for i, tvec, rvec in zip(ids, tvecs, rvecs):
        tvec = tvec.squeeze().tolist()
        rvec = rvec.squeeze().tolist()
        rotvector = np.array([[rvec[0], rvec[1], rvec[2]]])
        # Convert rotation vector to rotation matrix
        Rotation = cv2.Rodrigues(rotvector)[0]
        # Convert rotation matrix to Euler angles
        Euler = CvtRotationMatrixToEulerAngle(Rotation)
```

```
        # Object coordinates (camera system)
        target_coords = np.array([tvec[0], tvec[1], tvec[2], Euler[0], Euler[1], Euler
    return target_coords

if __name__ == "__main__":
    # Set camera ID
    camera = UVCCamera(5, mtx, dist)
    # Open the camera
    camera.capture()
    # Set left arm observation point
    origin_anglesL = [-44.24, 15.56, 0.0, -102.59, 65.28, 52.06, 23.49]
    # Set gripper movement mode
    ml.set_gripper_mode(0)
    # Set tool coordinate system
    ml.set_tool_reference([0, 0, Tool_LEN, 0, 0, 0])
    # Set the end coordinate system as the tool
    ml.set_end_type(1)
    # Set movement speed
    sp = 40
    # Move to the observation point
    ml.send_angles(origin_anglesL, sp)
    # Wait for the robotic arm to finish moving
    waitl()
    # Refresh the camera interface
    camera.update_frame()
    # Get the current frame
    frame = camera.color_frame()
    # Get the angles and IDs of the QR codes in the frame
    (corners, ids, rejected_corners) = stag.detectMarkers(frame, 11)
    # Get the object's coordinates (camera system)
    marker_pos_pack = calc_markers_base_position(corners, ids, MARKER_SIZE, mtx, dist)
    # Get the current coordinates of the robotic arm
    cur_coords = np.array(ml.get_base_coords())
    # Convert angle values to radians
    cur_bcl = cur_coords.copy()
    cur_bcl[-3:] *= (np.pi / 180)
    # Convert the object's coordinates (camera system) to (base coordinate system) thr
    fact_bcl = Eyes_in_hand_left(cur_bcl, marker_pos_pack)
    target_coords = cur_coords.copy()
    target_coords[0] = fact_bcl[0]
    target_coords[1] = fact_bcl[1]
    target_coords[2] = fact_bcl[2] + 50
    # Move the robotic arm above the QR code
    ml.send_base_coords(target_coords, 30)
    # Wait for the robotic arm to finish moving
    waitl()
    # Open the gripper
    ml.set_gripper_value(100, 100)
```

```
    # Move the robotic arm down along the z-axis
    ml.send_base_coord(3, fact_bcl[2], 10)
    # Wait for the robotic arm to finish moving
    waitl()
    # Close the gripper
    ml.set_gripper_value(20, 100)

    # Wait for the gripper to close
    time.sleep(2)

    # Lift the gripper
    ml.send_base_coord(3, fact_bcl[2] + 50, 10)
```

← Previous Page | Next Page →

Spatial Movement

Note: The chassis battery must be above 21V to move.

Case video

Keyboard Control for Chassis Movement

Execute the following command in a new terminal:

```
roslaunch turn_on_mercury_robot turn_on_mercury_robot.launch
```

Then, in another new terminal, execute:

```
roslaunch mercury_x1_teleop keyboard_teleop.launch
```

At this point, you can control the chassis movement by pressing the specified keyboard keys as prompted by the console.

Controlling the Chassis Movement and Navigation via API

Execute the following command in a new terminal:

```
roslaunch turn_on_mercury_robot navigation.launch
```

```
from mercury_ros_api import MapNavigation

if __name__ == "__main__":
    # Navigate to the target point, this point needs to be mapped and set by yourself
    map_navigation = MapNavigation()
    flag_feed_goalReached = map_navigation.moveToGoal(1.8811798181533813, 1.2514267325
    # Move the chassis forward
    map_navigation.goStraight(0.25, 5)
    # Move the chassis backward
    map_navigation.goBack(0.25, 5)
    # Move the chassis to the left
    map_navigation.turnLeft(0.25, 5)
    # Move the chassis to the right
    map_navigation.turnRight(0.25, 5)
```

Mobile and Grasping

Case video

Using the `camera_detect` Vision Recognition Package

```python
import numpy as np
from pymycobot import Mercury
# Import the vision recognition stag package
from camera_detect import camera_detect

if __name__ == "__main__":
    # Navigate to the target point, this point needs to be mapped and set by yourself
    map_navigation = MapNavigation()
    flag_feed_goalReached = MapNavigation.moveToGoal(1.8811798181533813, 1.25142673254
    # Check if the target point is reached
    if flag_feed_goalReached:
        camera_params = np.load("camera_params.npz")  # Camera configuration file
        mtx, dist = camera_params["mtx"], camera_params["dist"]
        ml = Mercury("/dev/left_arm")  # Set left arm port
        mr = Mercury("/dev/right_arm")  # Set right arm port
        arm = 0         # 0: Left arm, 1: Right arm
        catch_mode = 0  # 0: Horizontal grasping, 1: Vertical grasping
        # Camera ID needs to be changed according to the actual situation
        left_camera_id = 3
        right_camera_id = 6
        stag_size = 32
        # Create vision recognition objects for left and right arms
        ml_obj = camera_detect(left_camera_id, stag_size, mtx, dist, arm)
        mr_obj = camera_detect(right_camera_id, stag_size, mtx, dist, arm)
        # Move the left and right arms to the horizontal observation position for gras
        mr_obj.vision_trace(catch_mode, mr)
        ml_obj.vision_trace(catch_mode, ml)
```

Without Using the `camera_detect` Vision Recognition Package

```
from uvc_camera import UVCCamera
from arm_control import *
from marker_utils import *
import stag
from mercury_ros_api import MapNavigation

# Gripper tool length
Tool_LEN = 175
# Distance from camera center to flange
Camera_LEN = 78
np.set_printoptions(suppress=True, formatter={'float_kind': '{:.2f}'.format})
# Camera configuration file
camera_params = np.load("src/camera_params.npz")
mtx, dist = camera_params["mtx"], camera_params["dist"]
# QR code size
MARKER_SIZE = 32
# Set left arm port
ml = Mercury("/dev/left_arm")

# Convert rotation matrix to Euler angles
def CvtRotationMatrixToEulerAngle(pdtRotationMatrix):
    pdtEulerAngle = np.zeros(3)
    pdtEulerAngle[2] = np.arctan2(pdtRotationMatrix[1, 0], pdtRotationMatrix[0, 0])
    fCosRoll = np.cos(pdtEulerAngle[2])
    fSinRoll = np.sin(pdtEulerAngle[2])
    pdtEulerAngle[1] = np.arctan2(-pdtRotationMatrix[2, 0],
                                  (fCosRoll * pdtRotationMatrix[0, 0]) + (fSinRoll * p
    pdtEulerAngle[0] = np.arctan2((fSinRoll * pdtRotationMatrix[0, 2]) - (fCosRoll * p
                                  (-fSinRoll * pdtRotationMatrix[0, 1]) + (fCosRoll *
    return pdtEulerAngle

# Convert Euler angles to rotation matrix
def CvtEulerAngleToRotationMatrix(ptrEulerAngle):
    ptrSinAngle = np.sin(ptrEulerAngle)
    ptrCosAngle = np.cos(ptrEulerAngle)
    ptrRotationMatrix = np.zeros((3, 3))
    ptrRotationMatrix[0, 0] = ptrCosAngle[2] * ptrCosAngle[1]
    ptrRotationMatrix[0, 1] = ptrCosAngle[2] * ptrSinAngle[1] * ptrSinAngle[0] - ptrSi
    ptrRotationMatrix[0, 2] = ptrCosAngle[2] * ptrSinAngle[1] * ptrCosAngle[0] + ptrSi
    ptrRotationMatrix[1, 0] = ptrSinAngle[2] * ptrCosAngle[1]
    ptrRotationMatrix[1, 1] = ptrSinAngle[2] * ptrSinAngle[1] * ptrSinAngle[0] + ptrCo
    ptrRotationMatrix[1, 2] = ptrSinAngle[2] * ptrSinAngle[1] * ptrCosAngle[0] - ptrCo
    ptrRotationMatrix[2, 0] = -ptrSinAngle[1]
    ptrRotationMatrix[2, 1] = ptrCosAngle[1] * ptrSinAngle[0]
    ptrRotationMatrix[2, 2] = ptrCosAngle[1] * ptrCosAngle[0]
    return ptrRotationMatrix

# Convert coordinates to homogeneous transformation matrix (x, y, z, rx, ry, rz) in ra
```

```
def Transformation_matrix(coord):
    position_robot = coord[:3]
    pose_robot = coord[3:]
    # Convert Euler angles to rotation matrix
    RBT = CvtEulerAngleToRotationMatrix(pose_robot)
    PBT = np.array([[position_robot[0]],
                    [position_robot[1]],
                    [position_robot[2]]])
    temp = np.concatenate((RBT, PBT), axis=1)
    array_1x4 = np.array([[0, 0, 0, 1]])
    # Concatenate the two arrays row-wise
    matrix = np.concatenate((temp, array_1x4), axis=0)
    return matrix

def Eyes_in_hand_left(coord, camera):
    # Camera coordinates
    Position_Camera = np.transpose(camera[:3])
    # Robotic arm coordinate matrix
    Matrix_BT = Transformation_matrix(coord)
    # Hand-eye matrix
    Matrix_TC = np.array([[0, -1, 0, Camera_LEN],
                          [1, 0, 0, 0],
                          [0, 0, 1, -Tool_LEN],
                          [0, 0, 0, 1]])
    # Object coordinates (camera system)
    Position_Camera = np.append(Position_Camera, 1)
    # Object coordinates (base coordinate system)
    Position_B = Matrix_BT @ Matrix_TC @ Position_Camera
    return Position_B

# Wait for the robotic arm to finish moving
def waitl():
    time.sleep(0.2)
    while (ml.is_moving()):
        time.sleep(0.03)

# Get object coordinates (camera system)
def calc_markers_base_position(corners: NDArray, ids: T.List, marker_size: int, mtx: N
    if len(corners) == 0:
        return []
    # Get object rotation and translation vectors through QR code corners
    rvecs, tvecs = solve_marker_pnp(corners, marker_size, mtx, dist)
    for i, tvec, rvec in zip(ids, tvecs, rvecs):
        tvec = tvec.squeeze().tolist()
        rvec = rvec.squeeze().tolist()
        rotvector = np.array([[rvec[0], rvec[1], rvec[2]]])
        # Convert rotation vector to rotation matrix
        Rotation = cv2.Rodrigues(rotvector)[0]
```

```
        # Convert rotation matrix to Euler angles
        Euler = CvtRotationMatrixToEulerAngle(Rotation)
        # Object coordinates (camera system)
        target_coords = np.array([tvec[0], tvec[1], tvec[2], Euler[0], Euler[1], Euler
    return target_coords

if __name__ == "__main__":
    # Navigate to the target point, this point needs to be mapped and set by yourself
    map_navigation = MapNavigation()
    flag_feed_goalReached = MapNavigation.moveToGoal(1.8811798181533813, 1.2514267325
    # Check if the target point is reached
    if flag_feed_goalReached:
        # Set camera ID
        camera = UVCCamera(5, mtx, dist)
        # Open the camera
        camera.capture()
        # Set left arm observation point
        origin_anglesL = [-44.24, 15.56, 0.0, -102.59, 65.28, 52.06, 23.49]
        # Set gripper movement mode
        ml.set_gripper_mode(0)
        # Set tool coordinate system
        ml.set_tool_reference([0, 0, Tool_LEN, 0, 0, 0])
        # Set the end coordinate system as the tool
        ml.set_end_type(1)
        # Set movement speed
        sp = 40
        # Move to the observation point
        ml.send_angles(origin_anglesL, sp)
        # Wait for the robotic arm to finish moving
        waitl()
        # Refresh the camera interface
        camera.update_frame()
        # Get the current frame
        frame = camera.color_frame()
        # Get the angles and IDs of the QR codes in the frame
        (corners, ids, rejected_corners) = stag.detectMarkers(frame, 11)
        # Get the object's coordinates (camera system)
        marker_pos_pack = calc_markers_base_position(corners, ids, MARKER_SIZE, mtx, d
        # Get the current coordinates of the robotic arm
        cur_coords = np.array(ml.get_base_coords())
        # Convert angle values to radians
        cur_bcl = cur_coords.copy()
        cur_bcl[-3:] *= (np.pi / 180)
        # Convert the object's coordinates (camera system) to (base coordinate system)
        fact_bcl = Eyes_in_hand_left(cur_bcl, marker_pos_pack)
        target_coords = cur_coords.copy()
        target_coords[0] = fact_bcl[0]
        target_coords[1] = fact_bcl[1]
```

```
        target_coords[2] = fact_bcl[2] + 50
        # Move the robotic arm above the QR code
        ml.send_base_coords(target_coords, 30)
        # Wait for the robotic arm to finish moving
        waitl()
        # Open the gripper
        ml.set_gripper_value(100, 100)
        # Move the robotic arm down along the z-axis
        ml.send_base_coord(3, fact_bcl[2], 10)
        # Wait for the robotic arm to finish moving
        waitl()
        # Close the gripper
        ml.set_gripper_value(20, 100)

        # Wait for the gripper to close
        time.sleep(2)

        # Lift the gripper
        ml.send_base_coord(3, fact_bcl[2] + 50, 10)
```

← Previous Page | Next Chapter →

Keyboard typing

Case video

Case code

```python
import json
import threading

from uvc_camera import UVCCamera
import cv2
import time
import numpy as np
from transformation import *
from pymycobot import Mercury
import typing as T
import cv2.aruco as aruco
from arm_control import *
from marker_utils import *
import stag
from matplotlib.path import Path



# Convert the rotation matrix to Euler angles
def CvtRotationMatrixToEulerAngle(pdtRotationMatrix):
    pdtEulerAngle = np.zeros(3)
    pdtEulerAngle[2] = np.arctan2(pdtRotationMatrix[1, 0], pdtRotationMatrix[0, 0])
    fCosRoll = np.cos(pdtEulerAngle[2])
    fSinRoll = np.sin(pdtEulerAngle[2])
    pdtEulerAngle[1] = np.arctan2(-pdtRotationMatrix[2, 0],
                                  (fCosRoll * pdtRotationMatrix[0, 0]) + (fSinRoll * p
    pdtEulerAngle[0] = np.arctan2((fSinRoll * pdtRotationMatrix[0, 2]) - (fCosRoll * p
                                  (-fSinRoll * pdtRotationMatrix[0, 1]) + (fCosRoll *
    return pdtEulerAngle



# Obtain object coordinates (camera system)
def calc_markers_base_position(corners: NDArray, ids: T.List, marker_size: int, mtx: N
    if len(corners) == 0:
        return []
    rvecs, tvecs = solve_marker_pnp(corners, marker_size, mtx, dist)
    res = []
    for i, tvec, rvec in zip(ids, tvecs, rvecs):
        tvec = tvec.squeeze().tolist()
        rvec = rvec.squeeze().tolist()
        rotvector = np.array([[rvec[0], rvec[1], rvec[2]]])
        Rotation = cv2.Rodrigues(rotvector)[0]
        Euler = CvtRotationMatrixToEulerAngle(Rotation)
        cam_coords = tvec + rvec
        target_coords = cam_coords
    return target_coords



# Convert Euler angles into rotation matrices
```

```
def CvtEulerAngleToRotationMatrix(ptrEulerAngle):
    ptrSinAngle = np.sin(ptrEulerAngle)
    ptrCosAngle = np.cos(ptrEulerAngle)
    ptrRotationMatrix = np.zeros((3, 3))
    ptrRotationMatrix[0, 0] = ptrCosAngle[2] * ptrCosAngle[1]
    ptrRotationMatrix[0, 1] = ptrCosAngle[2] * ptrSinAngle[1] * ptrSinAngle[0] - ptrSi
    ptrRotationMatrix[0, 2] = ptrCosAngle[2] * ptrSinAngle[1] * ptrCosAngle[0] + ptrSi
    ptrRotationMatrix[1, 0] = ptrSinAngle[2] * ptrCosAngle[1]
    ptrRotationMatrix[1, 1] = ptrSinAngle[2] * ptrSinAngle[1] * ptrSinAngle[0] + ptrCo
    ptrRotationMatrix[1, 2] = ptrSinAngle[2] * ptrSinAngle[1] * ptrCosAngle[0] - ptrCo
    ptrRotationMatrix[2, 0] = -ptrSinAngle[1]
    ptrRotationMatrix[2, 1] = ptrCosAngle[1] * ptrSinAngle[0]
    ptrRotationMatrix[2, 2] = ptrCosAngle[1] * ptrCosAngle[0]
    return ptrRotationMatrix


# Coordinate conversion to homogeneous transformation matrix, (x, y, z, rx, ry, rz) un
def Transformation_matrix(coord):
    position_robot = coord[:3]
    pose_robot = coord[3:]
    RBT = CvtEulerAngleToRotationMatrix(pose_robot)
    PBT = np.array([[position_robot[0]],
                    [position_robot[1]],
                    [position_robot[2]]])
    temp = np.concatenate((RBT, PBT), axis=1)
    array_1x4 = np.array([[0, 0, 0, 1]])
    # Concatenate two arrays by row
    matrix = np.concatenate((temp, array_1x4), axis=0)
    return matrix


def Eyes_in_hand(coord, camera, arm):
    # Camera coordinates
    Position_Camera = np.transpose(camera[:3])
    # Robot arm coordinate matrix
    Matrix_BT = Transformation_matrix(coord)
    # Hand eye matrix
    if arm == "left":
        Matrix_TC = np.array([[0, -1, 0, Camera_LEN],
                              [1, 0, 0, -3],
                              [0, 0, 1, -Tool_LEN],
                              [0, 0, 0, 1]])
    else:
        Matrix_TC = np.array([[0, 1, 0, Camera_LEN],
                              [-1, 0, 0, 28],
                              [0, 0, 1, -Tool_LEN],
                              [0, 0, 0, 1]])
    # Object coordinates (camera system)
```

```
    Position_Camera = np.append(Position_Camera, 1)
    Position_B = Matrix_BT @ Matrix_TC @ Position_Camera
    return Position_B


# Waiting for the end of the dual arm operation
def wait():
    time.sleep(0.2)
    while (ml.is_moving() or mr.is_moving()):
        time.sleep(0.03)


# Waiting for the left arm to finish running
def waitL():
    time.sleep(0.2)
    while (ml.is_moving()):
        time.sleep(0.03)


# Waiting for the right arm to finish running
def waitR():
    time.sleep(0.2)
    while (mr.is_moving()):
        time.sleep(0.03)


# Draw a keyboard frame and read the coordinates of each letter
def draw(frame, arm):
    global per_right_corners, per_left_corners
    # Grayscale the image
    imGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Obtain the corner points of the QR code in the image
    (corners, ids, rejected_corners) = stag.detectMarkers(imGray, 11)
    # Obtain the displacement and rotation vectors of the QR code relative to the came
    marker_pos_pack = calc_markers_base_position(corners, ids, marker_size, mtx, dist)
    if arm == "left":
        # Obtain the current end coordinates of the robotic arm
        stag_cur_coords = np.array(ml.get_base_coords())
        stag_cur_bcl = stag_cur_coords.copy()
        stag_cur_bcl[-3:] *= (np.pi / 180)
        # Obtain the coordinates of the QR code relative to the base of the robotic ar
        stag_fact_bcl = Eyes_in_hand(stag_cur_bcl, marker_pos_pack, "left")
        stag_coord = stag_cur_coords.copy()
        stag_coord[0] = stag_fact_bcl[0]
        stag_coord[1] = stag_fact_bcl[1]
        stag_coord[2] = stag_fact_bcl[2]
        # Store the three-dimensional coordinates of the QR code
        keyboard_coords[","] = stag_coord
```

```
    else:
        # Obtain the current end coordinates of the robotic arm
        stag_cur_coords = np.array(mr.get_base_coords())
        stag_cur_bcl = stag_cur_coords.copy()
        stag_cur_bcl[-3:] *= (np.pi / 180)
        # Obtain the coordinates of the QR code relative to the base of the robotic ar
        stag_fact_bcl = Eyes_in_hand(stag_cur_bcl, marker_pos_pack, "right")
        stag_coord = stag_cur_coords.copy()
        stag_coord[0] = stag_fact_bcl[0]
        stag_coord[1] = stag_fact_bcl[1]
        stag_coord[2] = stag_fact_bcl[2]
        # Store the three-dimensional coordinates of the QR code
        keyboard_coords["."] = stag_coord
    # Obtain the displacement and rotation vectors of the QR code relative to the came
    rvecs, tvecs = solve_marker_pnp(corners, marker_size, mtx, dist)
    # Draw the coordinate system
    cv2.drawFrameAxes(frame, mtx, dist, rvecs, tvecs, 50)
    draw_img = frame

    x1 = corners[0][0][0][0]
    y1 = corners[0][0][0][1]
    x2 = corners[0][0][1][0]
    y2 = corners[0][0][1][1]

    x = x1 - x2
    y = y1 - y2
    # Obtain the deflection angle based on the angle of the line connecting two inters
    r = np.arctan(y / x)
    r = abs(r)
    # Obtain the distance between two corner points
    size = abs(x / np.cos(r))
    # The distance between the x-axis of the two buttons on the left arm camera
    left_x_dis = size * 1.3
    # The distance between the y-axis of the two buttons on the left arm camera
    left_y_dis = size * 1.35
    # The distance between the x-axis of the two buttons on the right arm camera
    right_x_dis = size * 1.3
    # The distance between the y-axis of the two buttons on the right arm camera
    right_y_dis = size * 1.35
    # Radius of key box
    rad = int(size / 2)
    # The offset distance between the x-axis of the left arm camera and the first lett
    left_add_x = [size * 1.25]
    # The offset distance between the y-axis rows of the left arm camera and the first
    left_add_y = [-size * 2, -size * 2.3, -size * 3]
    # The offset distance between the x-axis of the right arm camera and the first let
    right_add_x = [size * 1.3]
    # The offset distance between the y-axis rows of the right arm camera and the firs
```

```
right_add_y = [size * 4.1,  size * 2.1, size * 1]
# Get the center point of the keyframe
tray_frame = Path(corners[0][0])
tray_frame_center_plot = tray_frame.vertices.mean(axis=0)

# Draw the button box
def draw_keyboard(arm):
    if arm == "left":
        # The offset distance between the camera x-axis and the first letter
        add_x = left_add_x
        # The offset distance between the y-axis rows of the left arm camera and t
        add_y = left_add_y
        # Left arm button layout
        keyboard_txt = left_keyboard_txt
        # The distance between the x-axis of the two buttons on the left arm camer
        x_dis = left_x_dis
        # The distance between the y-axis of the two buttons on the left arm camer
        y_dis = left_y_dis
    else:
        # The offset distance between the camera x-axis and the first letter
        add_x = right_add_x
        # The offset distance between each row of the camera y-axis and the first
        add_y = right_add_y
        # Right arm button layout
        keyboard_txt = right_keyboard_txt
        # The distance between the x-axis of the two buttons on the right arm came
        x_dis = right_x_dis
        # The distance between the y-axis of the two buttons on the right arm came
        y_dis = right_y_dis

    # Loop drawing the keyboxes for each row and column
    for j in range(len(keyboard_txt)):
        for i in range(len(keyboard_txt[j])):
            # Obtain the corresponding key letters
            txt = keyboard_txt[j][i]
            # Obtain the two-dimensional coordinates of the corresponding keys
            if arm == "left":
                x_key = int(tray_frame_center_plot[0] + add_x[0] + x_dis * j)
                y_key = int(tray_frame_center_plot[1] + add_y[j] - y_dis * i)
            else:
                x_key = int(tray_frame_center_plot[0] - x_dis * j)
                y_key = int(tray_frame_center_plot[1] + add_y[j] + y_dis * i)
            if y1 < y2:
                rr = -r
            else:
                rr = r
            row = draw_img.shape[0]
```

```
x_key = x_key
y_key = row - y_key
x_center = tray_frame_center_plot[0]
y_center = row - tray_frame_center_plot[1]
# Obtain the position of a point in the image after rotating around ar
x_end = (x_key - x_center) * np.cos(rr) - (y_key - y_center) * np.sin(
y_end = (x_key - x_center) * np.sin(rr) + (y_key - y_center) * np.cos(
x_end = int(x_end)
y_end = int(row - y_end)
# Calculate the corner points of this button
key_corners = (np.array(
    [[(x_end - size / 2, y_end - size / 2),
      (x_end + size / 2, y_end - size / 2),
      (x_end + size / 2, y_end + size / 2),
      (x_end - size / 2, y_end + size / 2)]],
    dtype=np.float32,
),)
# Obtain the coordinates of the button (camera system)
marker_pos_pack = calc_markers_base_position(key_corners, ids, marker_
if arm == "left":
    # Obtain the current coordinates of the robotic arm
    left_cur_coords = np.array(ml.get_base_coords())
    left_cur_bcl = left_cur_coords.copy()
    left_cur_bcl[-3:] *= (np.pi / 180)
    # Convert key coordinates (camera system) to (base coordinate syst
    left_fact_bcl = Eyes_in_hand(left_cur_bcl, marker_pos_pack, "left"
    left_end_coord = left_cur_coords.copy()
    left_end_coord[0] = left_fact_bcl[0]
    left_end_coord[1] = left_fact_bcl[1]
    left_end_coord[2] = stag_coord[2]
    # Save the button positions in the dictionary for easy access to s
    if txt in left_control:
        keyboard_coords[txt] = left_end_coord
else:
    # Obtain the current coordinates of the robotic arm
    right_cur_coords = np.array(mr.get_base_coords())
    right_cur_bcl = right_cur_coords.copy()
    right_cur_bcl[-3:] *= (np.pi / 180)
    # Convert key coordinates (camera system) to (base coordinate syst
    right_fact_bcl = Eyes_in_hand(right_cur_bcl, marker_pos_pack, "rig
    right_end_coord = right_cur_coords.copy()
    right_end_coord[0] = right_fact_bcl[0]
    right_end_coord[1] = right_fact_bcl[1]
    right_end_coord[2] = stag_coord[2]
    # Save the button positions in the dictionary for easy access to s
    if txt in right_control:
        keyboard_coords[txt] = right_end_coord
# Draw the frame of the button
```

```
                cv2.circle(draw_img, (x_end, y_end), rad, (0, 0, 255), 2)

    draw_keyboard(arm)
    return draw_img


# Fine adjustment of left and right walls
l_x = 3
l_y = 3
r_x = - 5
r_y = -10


def keyboard():
    while True:
        key_code = cv2.waitKey(1) & 0xFF
        if key_code != 255:
            key_code = chr(key_code)
            if key_code in left_control:
                # Read the three-dimensional coordinates of the corresponding keys
                coords = keyboard_coords[key_code].copy()
                coords[0] = coords[0] + l_x
                coords[1] = coords[1] + l_y
                coords[2] = 315
                # Control point press button
                ml.send_base_coords(coords, sp)
                waitL()
                ml.send_base_coord(3, 305, sp)
                waitL()
                ml.send_base_coords(coords, sp)
                waitL()
                ml.send_angles(ml_pos, sp)
                waitL()
                print(coords)
            elif key_code in right_control:
                # Read the three-dimensional coordinates of the corresponding keys
                coords = keyboard_coords[key_code].copy()
                coords[0] = coords[0] + r_x
                coords[1] = coords[1] + r_y
                coords[2] = 330
                # Control point press button
                mr.send_base_coords(coords, sp)
                waitR()
                mr.send_base_coord(3, 320, sp)
                waitR()
                mr.send_base_coords(coords, sp)
                waitR()
                mr.send_angles(mr_pos, sp)
```

```
                waitR()
                print(coords)
            print("==============================================")


def write_string(txt):
    per_key = None
    # Loop click button
    for key_code in txt:
        cv2.waitKey(1)
        if key_code in left_control:
            # Read which arm the previous button was on. If it is the same arm, click
            if per_key not in left_control:
                # Return to the starting point
                mr.send_angles(mr_pos, sp)
                waitR()
            per_key = key_code
            # Read the three-dimensional coordinates of the corresponding keys
            coords = keyboard_coords[key_code].copy()
            coords[0] = coords[0] + l_x
            coords[1] = coords[1] + l_y
            coords[2] = 315
            # Control the mechanical arm by clicking and pressing
            ml.send_base_coords(coords, sp)
            waitL()
            ml.send_base_coord(3, 305, sp)
            waitL()
            ml.send_base_coords(coords, sp)
            waitL()
            print(coords)
        elif key_code in right_control:
            # Read which arm the previous button was on. If it is the same arm, click
            if per_key not in right_control:
                # Return to the starting point
                ml.send_angles(ml_pos, sp)
                waitL()
            per_key = key_code
            # Read the three-dimensional coordinates of the corresponding keys
            coords = keyboard_coords[key_code].copy()
            coords[0] = coords[0] + r_x
            coords[1] = coords[1] + r_y
            coords[2] = 330
            # Control the mechanical arm by clicking and pressing
            mr.send_base_coords(coords, sp)
            waitR()
            mr.send_base_coord(3, 320, sp)
            waitR()
            mr.send_base_coords(coords, sp)
```

```
            waitR()
            print(coords)
        print("==============================================")


# Open the camera to read the image
def camera():
    per_frame_left = None
    per_frame_right = None
    while True:
        left_camera.update_frame()
        left_frame = left_camera.color_frame()
        try:
            left_frame = draw(left_frame, "left")
            per_frame_left = left_frame
        except Exception as e:
            pass
        right_camera.update_frame()
        right_frame = right_camera.color_frame()
        try:
            right_frame = draw(right_frame, "right")
            per_frame_right = right_frame
        except Exception as e:
            pass
        # rotate image
        left_turn_img = cv2.rotate(left_frame, cv2.ROTATE_90_CLOCKWISE)
        right_turn_img = cv2.rotate(right_frame, cv2.ROTATE_90_CLOCKWISE)
        # merge images
        imgs = np.hstack([left_turn_img, right_turn_img])
        # Show pictures
        cv2.imshow("calibed", imgs)
        key_code = cv2.waitKey(1) & 0xFF
        # When pressing the q key, capture a picture for operation
        if chr(key_code) == "q":
            left_turn_img = cv2.rotate(per_frame_left, cv2.ROTATE_90_CLOCKWISE)
            right_turn_img = cv2.rotate(per_frame_right, cv2.ROTATE_90_CLOCKWISE)
            imgs = np.hstack([left_turn_img, right_turn_img])
            cv2.imshow("calibed", imgs)
            break
    # Control both arms and click on the keyboard to hello world
    write_string("helloworld")


if __name__ == "__main__":
    # Set left and right wall ports
    mr = Mercury("/dev/ttyACM0")
    ml = Mercury("/dev/ttyTHS0")
    # Store the corner values of the QR code read by the left and right arms
```

```
per_left_corners = []
per_right_corners = []
Sp=60 # Robot arm movement speed
Tool_LEN=175 # Distance between Claw and Flange
CameraLEN=78 # Distance between camera and flange
# Set the initial angle of the left and right arms
ml_pos = [-31.92, -7.36, 0.0, -110.82, 81.75, 59.04, 44.25]
mr_pos = [27.12, 28.34, 0.0, -108.77, -69.42, 71.92, -9.52]
# Left arm keyboard layout
left_keyboard_txt = [
    ["q", "w", "e", "r", "t", "y", "u", "i", "o", "p"],
    ["a", "s", "d", "f", "g", "h", "j", "k", "l"],
    ["z", "x", "c", "v", "b", "n", "m"]
]
# Right arm keyboard layout
right_keyboard_txt = [
    ["m", "n", "b", "v", "c", "x", "z"],
    ["l", "k", "j", "h", "g", "f", "d", "s", "a"],
    ["p", "o", "i", "u", "y", "t", "r", "e", "w", "q"],
]
# Left arm clicks on letters
left_control = ["q", "w", "e", "r", "t",
                "a", "s", "d", "f", "g",
                "z", "x", "c", "v", ","]
# Right arm clicks on letters
right_control = ["y", "u", "i", "o", "p",
                 "h", "j", "k", "l",
                 "b", "n", "m", "."]
# Store the three-dimensional coordinates of all keyboard keys
keyboard_coords = {}

np.set_printoptions(suppress=True, formatter={'float_kind': '{:.2f}'.format})
# Camera configuration file
camera_params = np.load("src/camera_params.npz")
mtx, dist = camera_params["mtx"], camera_params["dist"]
# QR code size
marker_size = 15
# Set left and right arm camera IDs
left_camera = UVCCamera(3, mtx, dist)
left_camera.capture()
right_camera = UVCCamera(2, mtx, dist)
right_camera.capture()
# Move the left and right walls to the initial point
mr.send_angles(mr_pos, sp)
ml.send_angles(ml_pos, sp)
# Waiting for the end of the exercise
wait()
time.sleep(2)
```

```
    camera()
    # threading.Thread(target=camera).start()
    # while True:
    #     s = input("键盘字母: ")
    #     if s in left_control:
    #         coords = keyboard_coords[s]
    #         print(coords)
    #
    #     elif s in right_control:
    #         coords = keyboard_coords[s]
    #         print(coords)
```

← Previous Page | Next Chapter →

Elephant Robotics

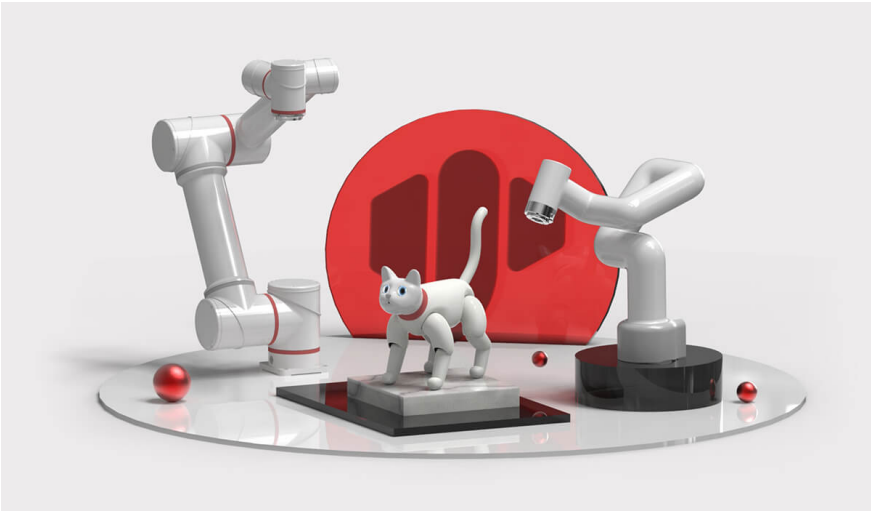
1. Company Introduction

Elephant Robotics, based in Shenzhen, China, is a high-tech enterprise focused on the research, design, and development of robots and automation solutions.

We are dedicated to providing highly flexible collaborative robots, easy-to-learn operating systems, and intelligent automation solutions for robot education and research institutions, commercial scenarios, and industrial production. Our product quality and intelligent solutions have been highly recognized and praised by several Fortune 500 companies from South Korea, Japan, the United States, Germany, Italy, Greece, and more.

Elephant Robotics upholds the vision of "Enjoy Robots World," advocating for human-robot collaboration, making robots good helpers in human work and life, helping people get liberated from simple, repetitive, and tedious work, fully leveraging the advantages of human-robot collaboration, thereby improving work efficiency and helping humans create a better new life.

In the future, Elephant Robotics hopes to promote the development of the robotics industry through new-generation cutting-edge technology and jointly open a new era of automation and intelligence with customers and partners.



2. Development History

- 2016.08 ----- Elephant Robotics Co., Ltd. was officially established
- 2016.08 ----- Entered HAX incubator and received SOSV seed round investment
- 2016.08 ----- Started developing Elephant S industrial collaborative robot
- 2017.01 ----- Awarded "CES China Top 10 Most Innovative Companies"
- 2017.04 ----- Attended Hannover Messe and Korea Automation Show
- 2017.07 ----- Two founders selected as "30 Under 30 Asia" by Forbes
- 2017.10 ----- Fifth generation single-arm industrial collaborative robot Elephant S launched
- 2018.04 ----- Received angel round investment from "Cloud Angel Fund"
- 2018.06 ----- First public appearance at Hannover Messe 2018
- 2018.06 ----- Awarded "Intelligent Manufacturing Entrepreneurship MBA Award" by Cheung Kong Graduate School of Business
- 2018.06 ----- Awarded "X-elerator Award" by Tsinghua SEM
- 2018.11 ----- Second place in the Shenzhen division of the Asia Smart Hardware Competition
- 2018.11 ----- Awarded "Most Investable Company Award" at the GG Gold Ball Awards
- 2019.03 ----- Awarded "Leader Award" at the GG Gold Ball Awards
- 2019.04 ----- Catbot awarded "Industrial Robot Innovation Award" in March 2019
- 2019.09 ----- Attended Huawei European Ecosystem Conference (HCE), officially becoming a Huawei ecosystem partner
- 2019.11 ----- Attended IROS International Conference on Intelligent Robots and Systems with Harbin Institute of Technology
- 2019.12 ----- "Intelligent Robot Joint Development Laboratory" with South China University of Technology officially unveiled
- 2019.12 ----- Awarded "Innovation Technology Award" at the GG Gold Ball Awards 2019
- 2019.12 ----- Awarded "Top 10 Fast-Growing Companies" at the GG Gold Ball Awards 2019

- 2019.12 ----- Awarded "New Star Company Award" in the industrial robot segment of Shenzhen Equipment Industry
- 2019.12 ----- World's first bionic robot cat MarsCat launched
- 2020.05 ----- Founder awarded "2019 Shenzhen Robotics New Star Award"
- 2020.10 ----- World's lightest and smallest six-axis collaborative robot myCobot launched
- 2021.03 ----- Smallest collaborative robot for research myCobotPro 320 launched
- 2021.05 ----- MarsCat featured in multiple media outlets including Xinhua Finance, China Daily, Nanjing Daily, and Harbin Daily
- 2021.07 ----- Launched the smallest composite robot chassis - myAGV
- 2021.09 ----- World's first fully enclosed four-axis robotic arm myPalletizer launched
- 2022.01 ----- Featured in 36Kr and GeekPark for lightweight consumer-grade robots
- 2022.02 ----- MarsCat and myCobot featured in CCTV Spring Festival Gala live broadcast and Shenzhen TV New Year special program
- 2022.05 ----- Launched the most compact small six-axis robotic arm mechArm, focusing on AI robot education
- 2022.06 ----- Collaborated with Unity engine to launch an introductory book on AI robot practice based on myCobot robot (international course)
- 2022.07 ----- Launched the simulation companion robot cat metaCat in the AI era
- 2022.07 ----- Launched the smallest dual-arm collaborative robot mybuddy
- 2022.08 ----- Awarded "Top 10 Non-Industrial Technology Innovation Award"
- 2022.08 ----- Founder awarded "2022 Shenzhen Robotics New Star Award"
- 2022.11 ----- Runner-up in the real-time engagement track of the iFLYTEK AI Developer Competition
- 2022.12 ----- Awarded "Best Robot Award" at the 2022 World Voice Expo 1024 Science and Technology Expo
- 2023 ----- Donated to Shanghai Putuo District Zhanyi Children's Intelligence Service Center, Shenzhen Futian District Tongban Time Special Children's Rehabilitation Center, and Renda Elderly Care Industry Group
- 2023.01 ----- Featured in CCTV News - Huaqiangbei Entrepreneur Interview "Towards Spring | Late Night Huaqiangbei"
- 2023.01 ----- Launched desktop high-precision robotic arm ultraArm
- 2023.02 ----- Microsoft successfully conducted ChatGPT for Robotics experiment using Elephant myCobot 280 robotic arm
- 2023.03 ----- Certified as a "Specialized and New" enterprise in Shenzhen
- 2023.06 ----- Selected as one of the first "Intelligent Robot Application Demonstration Cases" in Shenzhen
- 2023.07 ----- Featured in Shenzhen Business Daily "Collaborative Robots, Born for Companionship - A 'Big Elephant' Emerges from Huaqiangbei"
- 2023.07 ----- Fourth place in the South China division of the 15th China Shenzhen Innovation and Entrepreneurship Competition and the 9th China Hardware Innovation Maker Competition
- 2023.08 ----- Launched the world's smallest seven-degree-of-freedom collaborative robotic arm myArm

- 2023.09 ----- Selected for the "2023 NVIDIA Inception Showcase"
- 2023.09 ----- Received a letter of appreciation from the Shenzhen Municipal Organization Department
- 2023.10 ----- Second place in the finals of the 5th China Chip Competition
- 2023.10 ----- Awarded "Outstanding Enterprise in High-End Manufacturing" in the finals of the 12th China Innovation and Entrepreneurship Competition
- 2023.11 ----- Awarded "Export Commodity Brand Certificate" by the China Council for the Promotion of International Trade
- 2023.11 ----- Top three in the national finals of the 9th China Hard Innovation Competition
- 2023.11 ----- Hosted the Elephant Robotics & OpenCV AI Robot Practical Training Camp
- 2023.12 ----- Featured in the 137th issue of the official Raspberry Pi magazine MagPi with a review article on the Elephant Robotics myCobot 280 desktop six-axis collaborative robotic arm
- 2023.12 ----- Awarded "2023 Annual Scientific Research Contribution Award" and "Top 50 Most Investable Companies" by leaderobot
- 2023.12 ----- Second place in the finals of the first National Advanced Computing Technology Innovation Competition
- 2023.12 ----- Launched the Mercury series humanoid robot

3. Related Links

Purchase Links

- Taobao: https://shop504055678.taobao.com
- Shopify: https://shop.elephantrobotics.com/
- AliExpress: https://elephantrobotics.aliexpress.com/store/1101941423

  Other Information

  - Official Website: https://www.elephantrobotics.com
  - Videos
- Bilibili: https://space.bilibili.com/2126215657
- YouTube: https://www.youtube.com/c/Elephantrobotics
  - Facebook: https://www.facebook.com/mycobotcreator/
  - LinkedIn: https://www.linkedin.com/company/18319865
  - X (Twitter): https://twitter.com/CobotMy
  - Discord: https://discord.gg/2MAherp7nt
  - Hackster: https://www.hackster.io/elephant-robotics

4. Contact Us

> Our working hours are on Chinese working days, from 10 AM to 6 PM Beijing time.

- If you have any other questions, please contact us via the following email.
  E-mail :

```
support@elephantrobotics.com
```

- If you have purchase intentions or any parameter questions, please send an email to this address. [E-mail](sales@elephantrobotics.com) :

```
sales@elephantrobotics.com
```

- If the listed issues do not help you solve your problem and you have more after-sales questions, please send an email to this address.
  E-mail :

```
support@elephantrobotics.com
```

We will respond within 1-2 working days;

WeChat:
We provide one-on-one service through WeChat for users who purchase myArm M&C series products.

← Previous Chapter | Next Chapter →

Acknowledgments

We express our deep gratitude to everyone involved in the development, testing, and refinement of the Mercury series products. Every detail polished and every innovation made is a testament to the hard work and dedication of the team behind it.

Special Thanks:

R&D Team: Thank you for your innovative thinking and countless days and nights of effort, turning complex technology into user-friendly products.
Quality Assurance and Testing Team: Your meticulous attention to detail ensures the reliability of our products and the ultimate user experience.
Customer Support Team: Thank you for providing professional support to our users, helping them solve every issue during their usage.
Partners and Suppliers: Your support and services are crucial to the success of our products. Thank you for your high-quality raw materials and components, and your readiness to support us at all times.
Investors and Advisors: Without your trust and financial support, we could not bring these innovations to market. Your insights and guidance have always been our driving force.

User Appreciation:

We especially thank every user who has chosen and trusted the Mercury series products. Your feedback and suggestions are our motivation to keep moving forward and improving. We promise to continue listening to your voices and continuously optimize our products and services.

Future Outlook:

We look forward to continuing to explore and advance in the field of robotics technology with all stakeholders. Let's work together to create more possibilities and bring greater convenience and innovation to the world.

← Previous Chapter