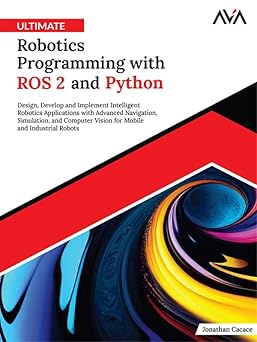
Ultimate Robotics Programming with ROS 2 and Python - 2024

*Orange Education Pvt*



# Preface

## Downloading the code bundles and colored images

<https://github.com/ava-orange-education/Ultimate-Robotics-Programming-with-ROS2-and-Python>

# 1. Introduction to Robot Operating System 2

## Introduction

## Structure

## Robot Programming

## Benefits of Using ROS 2

## ROS 2 Concepts

ROS 2 Nodes

Communication Models

Nodes Discovery

Packages and Workspace

ROS 2 Distributions and Licensing

## ROS 2 Internal Architecture

Data Distribution Service (DDS)

## Installing ROS 2

Installing ROS 2 Using Dockers

Installing Dockers

Creating the ROS 2 Image

Testing ROS 2 Installation

## Conclusion

# 2. Hands-on ROS 2 Programming Using Python

## Introduction

## Structure

## ROS 2 Command-Line Tools

Creating a ROS 2 Workspace

Creating a New ROS 2 Package

## Writing a First ROS 2 Node Using Python

## Implementing a Publisher Node Using Python

Publishing a Sinusoidal Signal

## Implementing a Subscriber Node Using Python

Receiving and Plotting a Sinusoidal Signal

## Implementing the Service Communication Model in ROS 2

Creating a Package for the Interfaces

Implementing the Service Node

Implementing the Client Node

## Implementing ROS 2 Action Client and Server

Creating an Action Package

Creating an Action Server Node

Creating an Action Client Node

## Creating Custom ROS 2 Messages

## Conclusion

# 3. Supplementary Tools for ROS 2

## Introduction

## Structure

## Getting Started with Turtlesim

Using Command-Line to Interact with ROS 2 Nodes

## Introducing ROS 2 Parameters

Using ROS 2 Parameters

## Starting Nodes Using Launch Files

Create your First Launch File

## ROS 2 Quality of Service (QoS)

Implement the pub-sub Protocol with QoS

## Record Topic Data Using Bagfiles

## Starting with RQT

## Conclusion

# 4. Robot Visualization and Simulation

## Introduction

## Structure

## The Importance of Simulation and Visualization

## Robot Modeling

Getting Started with URDF and Xacro

## Writing your First Robot Model

## Publishing Robot Model

TF System

The robot\_state\_publisher Node

## Starting with Rviz2

Visualizing the Robot Model

## Starting with Ignition Gazebo

Launching Gazebo

Spawn Robot Models in Gazebo

## Conclusion

# 5. Writing Tests Using Pytest for ROS 2 Nodes

## Introduction

## Structure

## Getting Started with Code Testing

Code Testing in Robotics

## Advances in Workspace Compilation

Automatize ROS 2 Initialization

Clean ROS 2 WorkspaceCompiling the Workspace

Navigate the Workspace

Compile Only One Package

## Introduction to Pytest Framework

Write Your First Basic Pytest Code

Check Test Results

## Using Pytest to Perform Integration Test

Creating the Custom Messages Package

Creating the ROS 2 Calculator Package

Creating Advanced Pytest File

## Conclusion

# 6. Controlling an Inverted Pendulum with a PID Controller

## Introduction

## Structure

## The Low-level Control Problem

## The ros2\_control Package

Simulate Robot Controllers

## Integrate Controllers into Gazebo Simulations

## Getting Started with the Controller Manager

Use the ros2 control Command

## Implement a PID Controller in ROS 2

## Conclusion

# 7. Laser-based Obstacle Avoidance with a Wheeled Mobile Robot

## Introduction

## Structure

## The Problem of Mobile Robot Navigation

Differential Drive Locomotion

Other Locomotion Systems

## Simulate a Differential Drive Robot Using Ignition Gazebo

Add Sensors and Controllers to Your Robot

## Navigate the Environment Using the APF

Test the Controller

## Conclusion

# 8. ROS 2 Behaviour Trees Using Python

## Introduction

## Structure

## Introduction to Behaviour Trees

BTs and FSMs

Behaviour Tree Elements

## Getting Started with py\_trees

Using py\_trees Sequence Composite

Using py\_trees Selector Composite

Interact with the py\_trees Blackboard

## Integrate ROS 2 with py\_trees

## Conclusion

# 9. Surveillance System Using Behaviour Trees

## Introduction

## Structure

## Introducing the TurtleBot Platform

Simulate the TurtleBot Robot Using Gazebo

Use Custom Models in Gazebo

## Executing Long-Term Operations with BTs

Integrate ROS 2 Actions into BTs

## Implementing a Surveillance Robot Using ROS 2 and BTs

## Conclusion

# 10. Robot Navigation Using ROS 2 Navigation Stack (Nav2)

## Introduction

## Structure

## Nav2 Architecture Overview

## Setting up Nav2 for Your Mobile Robot

Define Robot Model and Transforms

Understanding the Tfs

Understand Robot Odometry

Simulate Odometry Calculation Using Gazebo

Mapping and Localization

## Starting with Map Generation Using Nav2

Getting Started with Map Localization

## Getting Start with Robot Navigation in Nav2

## Implement a ROS 2 Client for Nav2

## Conclusion

# 11. Robot Arm Control Using MoveIt 2

## Introduction

## Structure

## Understanding the Robot Manipulation Problem

Robot Manipulator Control Modes

## Introducing MoveIt 2

Integration of MoveIt with ROS 2

## Getting Started with MoveIt 2

Using the MoveIt Setup Assistant

Integrate MoveIt Files in the URDF Model

Using Rviz2 to Plan Robot Motions

## Python Scripting and MoveIt 2

Plan Trajectories in Joint Space

Plan Trajectories in Cartesian Space

## Conclusion

# 12. Programming Aerial Robots Using ROS 2

## Introduction

## Structure

## Introduction to Aerial Robotics

Key Components of UAVs

The PixHawk FCU and the PX4 Control Stack

## Simulating Aerial Robot Using ROS 2 and Gazebo

PX4 Software-in-the-Loop

Starting with QGroundControl

PX4-ROS 2 Integration

Controlling a UAV Using ROS 2

## Simulating Drones with Additional Sensors

## Teleoperating a UAV Using the Keyboard

## Conclusion

# 13. Computer Vision Using ROS 2

## Introduction

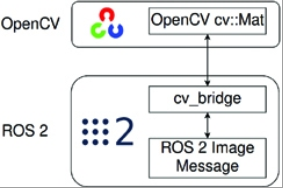
## Structure

## Robotics and Computer Vision

### Introducing OpenCV

### Understanding ROS 2: OpenCV Integration

OpenCV and ROS 2 integration is enabled by a particular package called **cv\_bridge**. Specifically, this package provides an interface to convert between ROS 2 image messages (sensor\_msgs/msg/Image) and OpenCV image objects, defined as *NumPy* arrays in Python. After making this conversion, each ROS 2 node can perform image elaboration using standard OpenCV functions. The opposite conversion can be operated to share the image over the ROS 2 network.



The **cv\_bridge** is part of the ROS 2 **vision\_opencv** toolkit that contains a set of packages to interface ROS 2 and OpenCV.

### Interfacing Cameras with ROS 2

Let's integrate the video stream provided by the laptop camera or an external USB camera with a ROS 2 node.

Use the following command to list the video devices.

$ ls /dev/video\*

$ /dev/video0 /dev/video1 /dev/video2 /dev/video3

Typically, the USB camera will show up as /dev/videoX, where the X is a number from 0 to the number of devices attached to your computer.

Typically, a camera can be associated with multiple devices (for example, if you only have the laptop camera, you will have /dev/video0 and /dev/video1). As you can see from the result of the previous command, the different video devices are listed. However, only one of those devicesstreams the video data.

Now that we have identified the device, we can write our first ROS 2 node interfaced with OpenCV. Let us create a new package storing the ROS 2 nodes.

$ ros2 pkg create --build-type ament\_python usb\_camera\_reader -- dependencies rclpy sensor\_msgs

In this first example, we test both the bridge that converts OpenCV data into ROS 2 topics and vice-versa and some functions of the OpenCV library. The main steps of the Python script are detailed as follows:

1. At the start, we read the data from a camera device. The device id is specified using a ROS 2 parameter.
2. The device is read by using the OpenCV function to get the stream from a device.
3. The image is resized and converted to a grayscale image. Then, it is shown in a window.
4. On the original image, some information is overprinted, such as the image resolution.
5. The new image so-shaped is published on a ROS 2 topic.

Let us create a new script called **camera\_publisher.py**.

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge

import cv2

class **CameraPublisher**(Node):

def **\_\_init\_\_**(self):

super().\_\_init\_\_('usb\_camera\_publisher')

The parameter to define the device id is defined. By default, it is assigned to 0 (this means that it is assigned to the /dev/video0 device).

self.declare\_parameter('camera\_id', 0)

camera\_id = self.get\_parameter('camera\_id').value

We then initialize the name of the topic on which the image is published and the device associated. These data are used in the following to be overwritten on the final image.

self.topic\_name = '/usb\_camera/image\_raw'

self.camera\_id = '/dev/video' + str(camera\_id)

The publisher is initialized to publish data of type Image.

self.publisher\_ = self.create\_publisher(Image, self.topic\_name, 10)

A timer is started to extract the image frame from the device. It runs at 10 Hz. Similarly, another timer is defined to elaborate the image taken from the device.

self.get\_frame\_timer = self.create\_timer(0.1, self.get\_image\_frame)

self.image\_elaboration\_timer = self.create\_timer(0.1, self.image\_elaboration)

The CvBridge object is initialized.

self.bridge = CvBridge()

self.frame = []

self.img\_ready = False

Finally, the VideoCapture function of OpenCV is used to initialize video capture from a camera device.

self.cap = cv2.VideoCapture(camera\_id)

if not self.cap.isOpened():

self.get\_logger().error(f"Could not open video device with ID {camera\_id}")

raise SystemExit

In the get\_image\_frame function, **the read function on the capture object of OpenCV is used to obtain the current frame from the device.** This function returns two elements. A boolean value, assessing if the image has been received or not and the image matrix. If the image is received correctly, we retrieve the image size (height and width) and use them to resize the original image three times. Finally, we convert it into a grayscale image.

def **get\_image\_frame**(self):

ret, self.frame = self.cap.read()

if ret:

height, width = self.frame.shape[:2]

new\_size = (width // 3, height // 3)

resized\_frame = cv2.resize(self.frame, new\_size)

gray\_frame = cv2.cvtColor(resized\_frame, cv2.COLOR\_BGR2GRAY)

The *imshow* function opens a new window, displaying the current frame. Note that the resultant image is resized and converted to grayscale.

cv2.imshow('frame', gray\_frame)

cv2.waitKey(1)

self.img\_ready = True

In the **image\_elaboration** timer, we retrieve the original OpenCV image to write additional information on it.

def **image\_elaboration**(self):

if (self.img\_ready == True):

The OpenCV frame is saved with a local one. Again, the resolution of the image is retrieved. Then, some lines of code are initialized to print the text in the top-left part of the image.

frame = self.frame

height, width, \_ = frame.shape

font = cv2.FONT\_HERSHEY\_SIMPLEX

font\_scale = 0.6

font\_color = (255, 255, 255)

font\_thickness = 1

resolution\_text = f"Resolution: {width}x{height}"

The positions where to print the three data are initialized in the following. Remember that the first pixel usable to print the data is (0, 0).

position\_device = (10, 30)

position\_topic = (10, 60)

position\_res = (10, 90)

Three text elements are put on the image. The device name, the topic on which the image is published, and the resolution of the image.

cv2.putText(frame, "Device: " + self.camera\_id,

position\_device, font, font\_scale, font\_color, font\_thickness)

cv2.putText(frame, "Topic: " + self.topic\_name,

position\_topic, font, font\_scale, font\_color, font\_thickness)

cv2.putText(frame, resolution\_text, position\_res, font,

font\_scale, font\_color, font\_thickness)

**The cv2\_to\_imgmsg function is used to directly translate the OpenCV image into a ROS 2 image datatype.** Then, this data is published.

image\_msg = self.bridge.cv2\_to\_imgmsg(frame, encoding="bgr8")

self.publisher\_.publish(image\_msg)

def **main**(args=None):

rclpy.init(args=args)

node = CameraPublisher()

rclpy.spin(node)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Change the setup.py file to add the new executable:

entry\_points={

'console\_scripts': [

'usb\_camera\_reader = usb\_camera\_reader.camera\_publisher:main',

],

},

Now, compile the workspace and execute this new node.

$ colcon build --synlink-install

$ ros2 run usb\_camera\_reader camera\_publisher --ros-args -p

camera\_id:=0

After starting this node, the image of the camera should be shown.

To view the content of the image published on ROS 2, with the text overlayed, you can use rqt\_image\_view, selecting the proper topic (/usb\_camera/image\_raw).

$ ros2 run rqt\_image\_view

#### usb\_cam package

In the previous example, we wrote our own procedure to read the data from the camera. However, ROS 2 has a package to implement the same procedure (read the data from the camera). This package is called usb\_cam and can be installed using the following command:

$ sudo apt-get install ros-humble-usb-cam

The usb\_cam package implements a node called usb\_cam\_node\_exe that streams the image data flow on a topic.

Let us create a package containing a launch file for this node. Execute this command in the src directory of your workspace.

$ ros2 pkg create usb\_cam\_launch

$ cd usb\_cam\_launch

$ mkdir launch && touch launch/usb\_cam.launch.py

$ mkdir config && touch config/cam\_parameters.yaml

The usb\_cam node must be configured properly. For this reason, we use a proper yaml configuration file. We will inspect the content of the configuration file. For now, just retrieve the path to the file.

def **generate\_launch\_description**():

usb\_cam\_params\_file = os.path.join(

get\_package\_share\_directory('usb\_cam\_launch'),

'config', 'cam\_params.yaml')

Then we create the only node executed in this launch file. We define two additional elements along with the name of the package and executable: the aforementioned parameter file and the remappings of the topic published by this node.

return LaunchDescription([

Node(

package='usb\_cam',

executable='usb\_cam\_node\_exe',

name='usb\_camera',

output='screen',

parameters=[usb\_cam\_params\_file],

As you can see, there are two topics remapped, that is, the image\_raw, defining the image, and the camera\_info. On this topic, the camera calibration is published.

remappings=[

('/image\_raw', '/camera/image\_raw'), ('/camera\_info',

'/camera/camera\_info'),]),])

Following is an example of the configuration file.

/\*\*:

ros\_\_parameters:

camera\_name: 'camera'

image\_width: 1280

image\_height: 720

framerate: 30.0

pixel\_format: 'yuyv' # Example, use your camera's format

io\_method: 'mmap' # Options: mmap, userptr, read, dma

video\_device: '/dev/video0' # The camera device path

### Starting with Camera Calibration

Essentially, to perform the projection of a pixel location into a 2D image plan, we must use additional information from the camera: the camera calibration. In particular, when we consider a sensor, we can have two types of calibration:

* **Extrinsic Calibration**: Refers to the external information about the sensor. In a few words, the rotation and translation of the camera with respect to the base frame of the robot.
* **Intrinsic Calibration**: Refers to the internal characteristics of the sensor. These are the settings that specify how the camera creates an image on its sensor.

In this section, we will only take into account the intrinsic calibration, simply referring to camera calibration. The goal of camera calibration is to map pixel coordinates to corresponding points in the real world.

There are different tools to perform camera calibration. Among them, ROS 2 directly wraps the OpenCV functions to perform the calibration. To do this, first, you must install the **camera-calibration** package.

$ sudo apt-get install ros-humble-camera-calibration

Then, after launching the node that streams the images of your camera, you must run the **cameracalibrator** with some parameters, using the following command:

$ ros2 run camera\_calibration cameracalibrator --size 10x7 -- square 0.02 --ros-args -r image:=/usb\_camera/image\_raw -p camera:=/usb\_camera

Here, the node accepts the following parameters:

1. **--size**: Specifies the number of inner corners on the chessboard pattern
2. **--square**: Specifies the square size of the chessboard in meters.
3. **--image**: The topic on which the image is published.
4. **--camera\_name**: A string name for your camera (you can choose the name of the camera).

The tool will display a GUI showing the camera feed, and you will move the chessboard or the camera around to capture calibration frames. Once enough good images are captured, the tool computes the calibration.

## Understanding Depth Sensors

depth sensors give 3D information, allowing robots to perceive and interact with their surroundings more effectively.

In this chapter, we will use the RealSense D435 integrated with ROS 2. Let us start installing RealSense in our system. We can install it directly using the ROS 2 humble packages.

$ sudo apt-get install ros-humble-realsense2-\* -y

Let us start the depth camera with the following command:

$ ros2 launch realsense2\_camera rs\_launch.py

### Starting Programming with Depth Sensors

In this application, we want to develop a sort of meter stick that provides the distance from the sensor and the object detected in the central pixel of the image. This distance, after being retrieved from the depth matrix, is printed on the RGB image and published on a ROS 2 topic. Create a package to store this node.

$ ros2 pkg create --build-type ament\_python distance\_calculator -- dependencies cv\_brdige rclpy sensor\_msgs

As dependencies, we must include the **cv\_bridge** to allow the conversion of the ROS 2 image (received on a topic) into an OpenCV image matrix. Create a Python node called **distance\_from\_depth.py**.

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge

import numpy as np

import cv2

In the main class of the application, link the node to its inputs: the RGB and Depth images. The topic names are the default ones published from the RealSense package publisher; you must change them accordingly to your topics.

class **DepthDistanceCalculator**(Node):

def **\_\_init\_\_**(self):

super().\_\_init\_\_('depth\_distance\_calculator')

self.subscription\_image = self.create\_subscription(Image,

'/camera/camera/color/image\_raw', self.image\_callback, 10)

self.subscription\_depth = self.create\_subscription(Image,

'/camera/camera/depth/image\_rect\_raw', self.depth\_callback, 10)

self.bridge = CvBridge()

self.color\_image = None

self.depth\_image = None

self.dist\_coeffs = None

self.distance = None

Finally, a publisher is initialized. We want to publish a **sensor\_msgs/msg/Image**, containing the image with the distance value printed on it.

self.publisher\_ = self.create\_publisher(Image, "/annotated\_image", 10)

In the image callback function, we retrieve the image streamed by the depth sensor. If a distance measurement has already been initialized and the image received from the sensor is valid, we can publish the output image using the **display\_annotated\_image** function.

def **image\_callback**(self, msg):

self.color\_image = self.bridge.imgmsg\_to\_cv2(msg, "bgr8")

if self.color\_image is not None and self.distance is not None:

self.display\_annotated\_image()

Similarly, in the **depth\_callback** function, we save the image in a class variable.

If the image converted from the ROS 2 message to OpenCV format is valid, we can start with the calculation of the distance.

def **depth\_callback**(self, msg):

self.depth\_image = self.bridge.imgmsg\_to\_cv2(msg, "32FC1")

if self.depth\_image is not None:

self.calculate\_distance()

def **calculate\_distance**(self):

height, width = self.depth\_image.shape

center\_x = width // 2

center\_y = height // 2

Then, we extract the depth value at the central point of the image. This data is encoded in millimeters. So, if an object is 1 meter away, the distance value is 1000. To have a better visualization, we convert it to meters.

self.distance = self.depth\_image[center\_y, center\_x]\*0.001

An important check to do when we work with the distance provided by the depth sensors is to check that the distance is valid. In some conditions, a not a number (NaN) value can be read from the distance due to bad light conditions or if the objects in the image are too far.

if not np.isfinite(self.distance):

self.get\_logger().warning('Invalid depth value at center point.')

self.distance = None

We are now ready to generate the image to publish on the ROS 2 network. First, we create a copy of the original image. Also, in this case, we retrieve the central pixel of the image and draw a circle on it to identify the central point.

def **display\_annotated\_image**(self):

annotated\_image = self.color\_image.copy()

height, width, \_ = annotated\_image.shape

center\_x = width // 2

center\_y = height // 2

cv2.circle(annotated\_image, (center\_x, center\_y), 5, (0, 0, 255), -1)

If the distance value is a valid value, we can print it on the image, as already shown in the previous example of this chapter.

if self.distance is not None and np.isfinite(self.distance):

distance\_text = f"{self.distance:.2f} m"font = cv2.FONT\_HERSHEY\_SIMPLEX

font\_scale = 0.6

color = (255, 255, 255)

thickness = 2

text\_position = (center\_x - 50, center\_y - 10)

cv2.putText(annotated\_image, distance\_text, text\_position, font,

font\_scale, color, thickness)

Finally, we can publish the image.

image\_msg = self.bridge.cv2\_to\_imgmsg(annotated\_image, encoding="bgr8")

self.publisher\_.publish(image\_msg)

#### depth\_image explanation from Claude

**What the depth\_image actually is:** Think of the depth\_image as a special photograph where instead of storing colors (red, green, blue) at each pixel, it stores distance values. It's still a 2D array with the same width and height as a regular camera image, but each "pixel" contains a depth measurement rather than color information. When you look at this data structure, you're seeing depth\_image[row, column] = distance\_value.

**How this differs from a point cloud:** A point cloud is a completely different data structure. Instead of organizing data in rows and columns like an image, a point cloud is a collection of individual 3D points, where each point has explicit X, Y, and Z coordinates in 3D space. You might think of it as a list of points: [(x1, y1, z1), (x2, y2, z2), (x3, y3, z3), ...].

**The relationship between them:** Here's where it gets interesting - you can convert a depth image into a point cloud, but they're not the same thing. To make this conversion, you need additional information called camera intrinsic parameters (things like focal length and optical center). The process involves taking each pixel's depth value and, using the camera parameters, calculating where that pixel would exist as a 3D point in real-world coordinates.

**Why use one format over the other:** The depth image format is more compact and easier to process when you only need specific measurements (like the center distance your code is calculating). Point clouds are better when you need to work with the full 3D geometry of a scene or perform operations like 3D object recognition or mapping.

#### distance\_from\_depth.py - source code

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge

import numpy as np

import cv2

class DepthDistanceCalculator(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('depth\_distance\_calculator')

# Create subscriptions to color image, depth image, and camera calibration

self.subscription\_image = self.create\_subscription(

Image, '/camera/camera/color/image\_raw', self.image\_callback, 10)

self.subscription\_depth = self.create\_subscription(

Image, '/camera/camera/depth/image\_rect\_raw', self.depth\_callback, 10)

self.bridge = CvBridge()

self.color\_image = None

self.depth\_image = None

self.dist\_coeffs = None

self.distance = None

self.publisher\_ = self.create\_publisher(Image, "/annotated\_image", 10)

def image\_callback(self, msg):

self.color\_image = self.bridge.imgmsg\_to\_cv2(msg, "bgr8")

if self.color\_image is not None and self.distance is not None:

self.display\_annotated\_image()

def depth\_callback(self, msg):

self.depth\_image = self.bridge.imgmsg\_to\_cv2(msg, "32FC1")

if self.depth\_image is not None:

self.calculate\_distance()

def calculate\_distance(self):

if self.depth\_image is None:

self.get\_logger().warning('Depth image not available.')

return

# Get the image height and width

height, width = self.depth\_image.shape

# Calculate the central point of the depth image

center\_x = width // 2

center\_y = height // 2

# Extract the depth value at the central point

self.distance = self.depth\_image[center\_y, center\_x]\*0.001

# Check if the distance value is valid

if not np.isfinite(self.distance):

self.get\_logger().warning('Invalid depth value at center point.')

self.distance = None

def display\_annotated\_image(self):

# Make a copy of the color image to annotate

annotated\_image = self.color\_image.copy()

# Get the image dimensions

height, width, \_ = annotated\_image.shape

# Calculate the central point

center\_x = width // 2

center\_y = height // 2

# Draw a red dot at the central point

cv2.circle(annotated\_image, (center\_x, center\_y), 5, (0, 0, 255), -1)

# Add the distance text near the central point

if self.distance is not None and np.isfinite(self.distance):

distance\_text = f"{self.distance:.2f} m"

font = cv2.FONT\_HERSHEY\_SIMPLEX

font\_scale = 0.6

color = (255, 255, 255) # White color

thickness = 2

text\_position = (center\_x - 50, center\_y - 10)

cv2.putText(annotated\_image, distance\_text, text\_position, font, font\_scale, color, thickness)

# Display the annotated image in a window

image\_msg = self.bridge.cv2\_to\_imgmsg(annotated\_image, encoding="bgr8")

self.publisher\_.publish(image\_msg)

def main(args=None):

rclpy.init(args=args)

node = DepthDistanceCalculator()

rclpy.spin(node)

node.destroy\_node()

rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

main()

### Getting Started with Point Cloud Data

**Point Cloud**. This data is a collection of points in 3D space, representing the surface of objects or an environment. Each point in a point cloud typically contains 3D coordinates (x, y, z), which describe its position relative to a reference frame, often along with additional attributes such as color (RGB) or intensity.

The Point Cloud data structure is described as follows:

* **Points**: Each point in the cloud has 3D coordinates that describe its position. In some cases, each point may also include extra data, such as:
  + **Color (RGB)**: Indicates the color of the point, useful for visualizing or identifying objects.
  + **Intensity**: Reflects how strongly the sensor detected the point, often used with LiDAR sensors.
* **Density**: Point clouds can be **dense** (many points per unit area) or **sparse** (fewer points per area), depending on the sensor resolution and scene complexity.

#### Point Cloud data according to Claude

**The core data in point clouds:** At its foundation, every point in a point cloud contains three essential coordinates: X, Y, and Z positions in 3D space. These coordinates tell you exactly where each point exists in the real world, typically measured in meters from some reference point (often the sensor's location). So yes, distance information is definitely contained in point clouds, but it's represented differently than in depth images.

**How distance works in point clouds:** Rather than storing distance as a single value like in depth images, the distance information in point clouds is embedded in the 3D coordinates themselves. If you want to know how far a point is from the sensor, you calculate it using the Pythagorean theorem in 3D space: distance = sqrt(x² + y² + z²). This gives you the straight-line distance from the origin (usually the sensor) to that specific point.

**Additional data that point clouds often contain:** Here's where point clouds become really interesting. Beyond just position, each point can carry additional attributes that make the data much more informative. Color information is very common - each point might have red, green, and blue values, essentially giving you a 3D colored representation of the scene. Imagine being able to walk around inside a photograph, and you'll get the idea.

**Intensity and reflectance values:** Many point clouds, especially those from LiDAR sensors, include intensity values. These represent how strongly the laser light reflected off each surface. Shiny metal surfaces might have high intensity values, while dark, absorptive materials might have low values. This information helps distinguish between different types of materials even when they're the same color.

**Normal vectors for surface understanding:** More advanced point clouds might include normal vectors for each point. These are mathematical descriptions of which direction a surface is facing at that specific location. Think of it like having tiny arrows pointing perpendicular to every surface in your 3D scene. This information is incredibly valuable for understanding the geometry and orientation of objects.

**Timestamps and motion data:** In dynamic environments, point clouds might include timestamp information for each point, allowing you to understand not just where things are, but when they were observed. Some systems even include velocity information, helping track how objects are moving through space.

*End of Claude info*

In ROS 2, point clouds are managed through the **sensor\_msgs/PointCloud2** message type, which is widely used for exchanging 3D point cloud data between nodes. We can easily visualize the Point Cloud generated from the depth sensors using Rviz2. Let us do it. Start the RealSense camera.

$ ros2 launch realsense2\_camera rs\_launch.py

Open and configure Rviz2.

$ rviz2

To configure RViz2 for depth visualization, add the **DepthCloud** plugin and set it to display the data from the **/camera/camera/depth/image\_rect\_raw** topic. As always, make sure to set the **Fixed Frame** correctly. The **rs\_launch.py** file will also publish the necessary transforms (TFs) for the sensor’s optical elements. This allows you to visualize everything relative to the **camera\_link** frame, making it possible to view the point cloud as if you were seeing the scene from the camera’s perspective.

we have a library that can be used to manipulate this data, that is, the **Point Cloud Library** (PCL). This is an open-source library specifically designed for processing and analyzing 3D point cloud data. It provides a comprehensive set of tools for tasks such as filtering, feature extraction, segmentation, registration, and visualization of point clouds. PCL is widely used in robotics, computer vision, and 3D perception for applications such as object recognition, 3D mapping, and Simultaneous Localization and Mapping (SLAM).

Links PCL:

<https://pointclouds.org/>

<https://pcl.readthedocs.io/projects/tutorials/en/master/>

A good alternative for Python is **Open3D**.

<https://www.open3d.org/>

Or PyTorch3D

<https://pytorch3d.org/>

PCL supports various sensor formats, such as LiDAR and RGB-D cameras, and integrates well with ROS for real-time processing. It includes modules for:

* **Filtering**: Reducing noise or down sampling point clouds.
* **Segmentation**: Identifying distinct objects or surfaces.
* **Registration**: Aligning multiple point clouds.
* **Feature extraction**: Identifying key features such as edges or corners

PCL can be mainly programmed in C++. Some unofficial wrappers for PCL exist in Python but they are not actively maintained. However, also in this case, ROS 2 packages that need Point Cloud data to work take such data directly from the topic.

## Solving Object Pose Reconstruction Using ARTags and ROS 2

the use of additional image elaboration tools based on the fiducial markers—also referred to as Augmented Reality Markers (**ARMarkers**)—is common in robotics. A **fiducial marker** is a predefined visual pattern used to provide a reference point for cameras or sensors to detect and interpret within a scene. These markers are often used in computer vision, robotics, and augmented reality to estimate the position, orientation, or scale of objects relative to the marker. A common type of fiducial markers is known as **ARMarkers**,



Each tag encodes a unique ID, allowing a system to differentiate between multiple tags. When placed in the environment or attached to objects, ARTags can be detected by a camera and used for tasks such as augmented reality applications, robot localization, and pose estimation. In robotics, ARTags are widely used for:

* **Localization and Navigation**
* **Object Tracking**:
* **Pose Estimation**: By detecting the 3D pose (position and orientation) of ARTags, robots can adjust their movement or
* interaction with objects accordingly.
* **Vision-Based Control**: ARTags provide visual feedback that robots use for precise control during tasks such as pick-and-place operations.

### Getting Started with Aruco Using ROS 2

The Aruco library is already installed in the OpenCV libraries.

### Performing Object Classification Using Machine Learning

Let us start integrating YOLO in our ROS 2 Python node. For this example, we will use a version of YOLO developed by the Ultralytics company that mainly contributed to the development of **YOLOv8**. Before starting programming YOLO, install the needed Python module.

$ sudo apt-get install python3-pip

$ pip3 install ultralytics

Create the package storing the YOLO ROS 2 node.

$ ros2 pkg create yolo\_object\_detector --build-type ament-python - -dependencies cv\_bridge sensor\_msgs rclpy

Add the content of the Python script in the **yolo.py** source

Among the different modules, we must include the YOLO module of the **ultralytics** library.

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge

import cv2

from ultralytics import YOLO

import numpy as np

from std\_msgs.msg import String

**Note**: ultralytics requires the version of numpy lower than 2, so the execution of this script could generate an execution error. To solve this problem, install the correct version of the library using the following command:

$ pip install numpy<2

In the ROS 2 node definition, let us subscribe to the **image\_raw** message. In this case, we do not need the camera calibration.

class **YoloNode**(Node):

def **\_\_init\_\_**(self):

super().\_\_init\_\_('yolo\_node')

self.subscription = self.create\_subscription(Image, '/camera/image\_raw',

self.image\_callback, 10)

We want to publish the result of the detections as a string. All the classes of the detected objects are spaced with a comma.

self.detections\_publisher = self.create\_publisher(String, '/yolo/detections', 10)

self.bridge = CvBridge()

Before using it, we must initialize the YOLO object, defining the weights of the neural network used internally by YOLO. We can rely on the default weight provided by the ultralytics library, using the **yolov8n.pt**.

self.model = YOLO('yolov8n.pt')

The detection happens in the **image\_callback**. We must just convert the input image into an OpenCV image and pass this image to the YOLO model.

def **image\_callback**(self, msg):

frame = self.bridge.imgmsg\_to\_cv2(msg, "bgr8")

results = self.model(frame)

The **display\_results** function writes on the image the result of the classification, while the **format\_detections** returns all the labels of the detected objects.

self.display\_results(frame, results)

detections = self.format\_detections(results)

detection\_msg = String()

detection\_msg.data = detections

self.detections\_publisher.publish(detection\_msg)

In the **display\_results** function, we just elaborate the output of the YOLO classifier to print the labels and their bounding boxes on the image.

def **display\_results**(self, frame, results):

for result in results:

boxes = result.boxes

for box in boxes:

First, for each object, let us extract the bounding box coordinates.

x1, y1, x2, y2 = map(int, box.xyxy[0])

Extract the class ID and the confidence score of the class.

class\_id = int(box.cls[0])

confidence = box.conf[0]

Finally, get the label of the class ID. Later, the bounding boxes are directly printed on the image.

label = self.model.names[class\_id]

cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

cv2.putText(frame, f'{label} {confidence:.2f}', (x1, y1 - 10),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 2)

cv2.imshow('YOLOv8 Detections', frame)

cv2.waitKey(1)

As already said, the **format\_detections** is used to collect the output of YOLO and translate it into a sequence of labels spaced with a comma.

def format\_detections(self, results):

# Format the detection results as a string for publishing

detections = []

for result in results:

boxes = result.boxes

for box in boxes:

class\_id = int(box.cls[0])

label = self.model.names[class\_id]

detections.append(label)

return ', '.join(detections) if detections else 'No objects detected'

In the **main** function, we must initialize the Yolo node.

def main(args=None):

rclpy.init(args=args)

node = YoloNode()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Modify the **setup.py** file, adding the **entry\_points**.

entry\_points = {

'console\_scripts': [

'yolo\_object\_detector = yolo\_object\_detector.yolo:main'

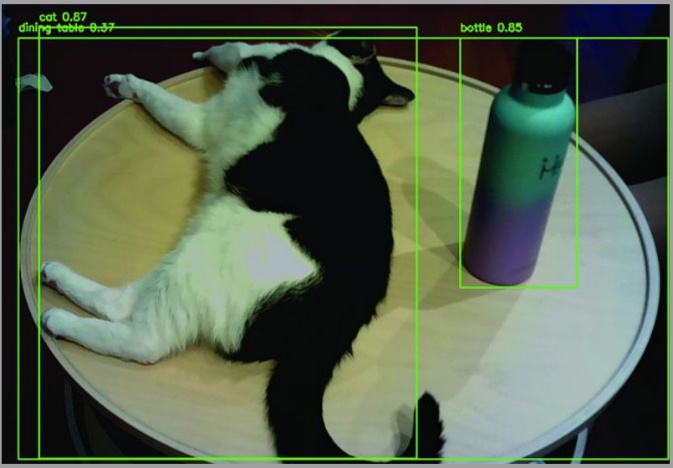
Now, compile the workspace and source it.

To run the YOLO, start the camera and the **yolo\_object\_detector** node. Of course, the USB camera must be connected to your computer.

$ ros2 launch usb\_cam\_launch usb\_cam.launch.py

$ ros2 run yolo\_object\_detector

The image generated by YOLO is depicted in Figure 13.13. As you can see, three objects are detected with different confidences.



***Figure 13.13:*** *Result of YOLO detector*

#### yolo.py – source code

import rclpy

from rclpy.node import Node

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge

import cv2

from ultralytics import YOLO

import numpy as np

from std\_msgs.msg import String

class YoloNode(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('yolo\_node')

# Create subscriber to video stream

self.subscription = self.create\_subscription(

Image,

'/camera/image\_raw', # The topic where the video stream is published

self.image\_callback,

10)

# Create publisher for the detection results

self.detections\_publisher = self.create\_publisher(String, '/yolo/detections', 10)

# Initialize CvBridge to convert between ROS Image messages and OpenCV images

self.bridge = CvBridge()

# Load YOLOv8 model (pre-trained on COCO dataset)

self.model = YOLO('yolov8n.pt') # You can replace with custom weights if needed

self.get\_logger().info('YOLO Node has been started.')

def image\_callback(self, msg):

# Convert ROS Image message to OpenCV image

frame = self.bridge.imgmsg\_to\_cv2(msg, "bgr8")

# Use YOLO to make predictions

results = self.model(frame)

# Draw bounding boxes and labels on the frame

self.display\_results(frame, results)

# Publish the detected object classes as a string

print("results: ", results)

detections = self.format\_detections(results)

detection\_msg = String()

detection\_msg.data = detections

self.detections\_publisher.publish(detection\_msg)

def display\_results(self, frame, results):

# Draw boxes and labels

for result in results:

boxes = result.boxes

for box in boxes:

# Extract bounding box coordinates

x1, y1, x2, y2 = map(int, box.xyxy[0])

# Extract the class ID and confidence score

class\_id = int(box.cls[0])

confidence = box.conf[0]

# Get label name for the class

label = self.model.names[class\_id]

# Draw bounding box and label on the image

cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

cv2.putText(frame, f'{label} {confidence:.2f}', (x1, y1 - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 2)

# Show the frame (for debugging)

cv2.imshow('YOLOv8 Detections', frame)

cv2.waitKey(1)

def format\_detections(self, results):

# Format the detection results as a string for publishing

detections = []

for result in results:

boxes = result.boxes

for box in boxes:

class\_id = int(box.cls[0])

label = self.model.names[class\_id]

detections.append(label)

return ', '.join(detections) if detections else 'No objects detected'

def main(args=None):

rclpy.init(args=args)

node = YoloNode()

try:

rclpy.spin(node)

except KeyboardInterrupt:

node.get\_logger().info("Shutting down YOLO node")

finally:

node.destroy\_node()

rclpy.shutdown()

cv2.destroyAllWindows()

if \_\_name\_\_ == '\_\_main\_\_':

main()

## Conclusion

# 14. Object Detection Using ROS 2

## Introduction

## Structure

## Introducing the Object Detection Problem

let us detail the problems addressed in this chapter:

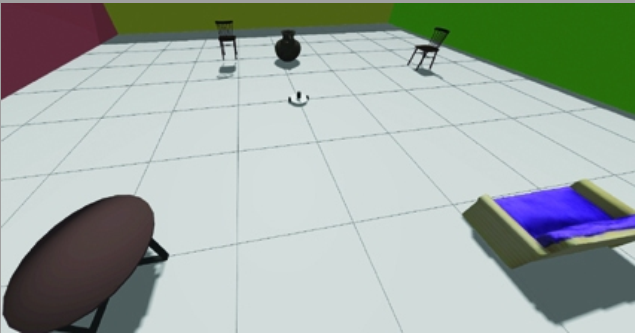
1. **Create a Custom World in Gazebo**: So far, we have been using basic geometric shapes, such as cubes and cylinders, to populate Gazebo. However, to effectively use object detection, we need more realistic objects. In this chapter, we will learn how to create a custom world with more complex and detailed objects.

2. **Simulate Depth Sensors**: Integrate depth sensors into Gazebo simulations by equipping a wheeled mobile robot with these sensors.

3. **Transform Points from Pixel Camera to 3D Plane Coordinates**: Using the depth data from the depth camera and the camera’s intrinsic calibration, transform a pixel point into a 3D coordinate that can be used to control the robot.

4. **Control the Robot based on the Camera Data**: In each iteration, the data processed by the camera is used to generate the appropriate control command for the robot’s base. In robotics, this type of control is known as **visual servoing**. In this chapter, we will explore a simplified version of this control method.

## Using YOLO in a Simulation Environment



*Figure 14.2: Gazebo simulation world to perform object detection*

Let us start creating a package to implement the simulation world.

$ ros2 pkg create seek\_and\_go\_world

We need a world directory to store the world file, a models directory to store some of the object model files, and a launch directory for the world launch file.

$ mkdir seek\_and\_go\_world/world

$ mkdir seek\_and\_go\_world/launch

$ mkdir seek\_and\_go\_world/models

Before defining the world file, let us quickly recap how simulation worlds are described in **Gazebo**. Gazebo uses **Simulation Description Format** (**SDF**), an XML-based format that allows you to define the properties of a simulation environment. With SDF, you can specify objects, robots, sensors, lights, and various environmental settings. Every world starts with the world tag. We will create the SDF file in the world directory of the seek\_and\_go\_world package.

$ touch seek\_and\_go\_world/world/seek\_and\_go.sdf

A world file consists of several elements, such as physics, the ground plane, and something similar.

Realistic models consist of 3D shapes (CAD models) with textures. New models can be included mainly in two ways. The simplest one is by including the link of the model from the Gazebo model website, which can be reached at this link: <https://app.gazebosim.org/fuel>.

In the **seek\_and\_go** world file, we included two models following this method, for example, the chair. The link of this model is as follows: <https://app.gazebosim.org/OpenRobotics/fuel/models/WoodenChair>

Each webpage of the model contains a panel to download the model locally (this is useful if you do not have an internet connection when you start the simulation) or to copy SDF block code to directly paste the code in the world file.

<include>

<uri>

https://fuel.gazebosim.org/1.0/OpenRobotics/models/WoodenChair

</uri>

</include>

This line creates the object, placing it in the (0, 0, 0) location with a default name. If you want to assign a different location and a different name, you can add the **name** and **pose** tags in the SDF snippet.

<include>

<name>Chair</name>

<pose>0.9 -3 0 0 0 1.57 </pose>

<uri>

https://fuel.gazebosim.org/1.0/OpenRobotics/models/WoodenChair

</uri>

</include>

If you want to avoid recreating the model when the simulator starts, you should save the model in a local folder. Use the download button on the model page to download an archive folder containing all necessary files to import your model into the simulation. Extract the folder in the **seek\_and\_go\_world** package into the models subfolder. In the provided example, we downloaded the **3D\_Dollhouse\_Sofa** model.

To include a local model in the simulation environment, two things are needed:

* Define the **IGN\_GAZEBO\_RESOURCE\_PATH** environment variable. We can do this directly in the launch file.
* Import the model in the SDF file. Like the online models, we need to use the include tag:

<include>

<pose>-2.3 -1.5 0 0 0 1.55 </pose>

<uri>

model://3D\_Dollhouse\_Sofa

</uri>

</include>

As you can see from the last code snippet, it points to a local model repository. In the launch file, we will see how to set this variable. To start the simulation, we must create a proper launch file. Let us create the **seek\_and\_go\_world.launch.py** script

Before compiling the workspace to add the resources developed in this package to your ROS 2 system, remember to modify the **CMakeLists.txt** to save the models, the launch, and the world files in your install subdirectory of the ROS 2 workspace.

install(DIRECTORY launch world models DESTINATION

share/${PROJECT\_NAME})

To finish the setup of our simulation environment, we must add a new sensor to the mobile robot developed in the **nav2\_mobile\_robot** package, the depth sensor.

### Adding the Depth Sensor to a Simulated Robot

To perform object detection using YOLO, a colored camera is enough. However, to perform the navigation task, retrieving the 3D coordinates of the objects detected by YOLO, we must use the information generated by the depth image of the depth sensor. For this reason, let us modify the mobile robot developed into the **nav2\_mobile\_robot** package.

We are going to put the sensor directly into the **nav2\_mobile\_robot.xacro** file. To include the depth sensor, we use another model file, implemented into the **realsense2\_description** package. This package gives us a structure for the model definition, such as where the optical cameras and colored cameras are placed with respect to the sensor frame. To install this package, use the following command:

$ sudo apt-get install ros-humble-realsense2-description

To finally add the model, follow the next steps.

1. In the robot mode file, we must create the link associated with the sensor. Then, we must fix the link to the base structure of the robot using a fixed joint.

<link name="depth\_link">

…

</link>

<joint name='depth\_link\_joint' type='fixed'>

<origin xyz="0 0 0.1" rpy="0 0 0"/>

<parent link="base\_link"/>

<child link="depth\_link"/>

</joint>

1. As already done with the other sensors of this robot, we must use the following tags. First, we inform the model that we have a Gazebo section, referring to a specific link: the just added **depth\_link**. In this code section, we include the xacro file from the **realsense2\_description** package and add the mechanical part of the sensor.

<gazebo reference="depth\_link">

<xacro:include filename="$(find

realsense2\_description)/urdf/\_d435.urdf.xacro"/>

<xacro:sensor\_d435 parent="depth\_link"

use\_nominal\_extrinsics="true" add\_plug="false"

use\_mesh="true">

<origin xyz="0 0 0.5" rpy="0 0 0"/>

</xacro:sensor\_d435>

1. In the gazebo sensor tag, we add the two sensors; one type is the **camera**, and the other is the **depth\_camera**. The name of the sensor is free, while the type of the sensor defines how the sensor acts in the scene. Inside the two sensor sections, we must include some parameters, such as the field of view, the resolution, and something similar.

<sensor name="camera" type="camera">

…

</sensor>

<sensor name='d435\_depth' type='depth\_camera'>

…

</sensor>

</gazebo>

### Implementing YOLO Detector Messages

we want to generate a list of detected objects, where each object includes both its class (category) and the pixel coordinates of the center of its bounding box. This structured data will allow other nodes to not only know which objects are detected at any given moment but also where they are located in the image.

To achieve this, we will define new message types in the **yolo\_msgs** package.

$ ros2 pkg create yolo\_msgs --dependencies std\_msgs rosidl\_default\_generators

$ mkdir yolo\_msgs/msg

Two message files are defined, one to define the single element of the list and another to define the list itself.

$ touch yolo\_msgs/msg/DetectedObject.msg

$ touch yolo\_msgs/msg/DetectedObjectList.msg

Each element of the list contains four fields: the belonging class, the detection confidence (how YOLO trusts its detection), and the center pixel of the bounding box.

string class\_name

float32 confidence

int32 center\_x

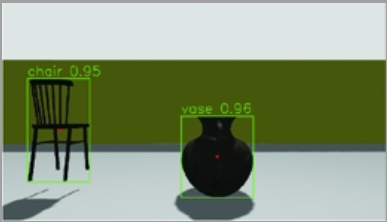
int32 center\_y

As for the **DetectedObjectList**, it is just a dynamic vector containing a number of **DetectedObject**.

DetectedObject[] objects # Array of detected objects

We can now implement the YOLO detector, using this message type to inform the ROS 2 network of all the detected objects and their position in the environment.

### Performing Image Elaboration Using YOLO



*Figure 14.4: Output of YOLO detector on the simulated scene*

Considering this image, the object list published as output will have the following shape:

$ ros2 topic echo /yolo/detections/list

---

objects:

- class\_name: vase

confidence: 0.9564287662506104

center\_x: 714

center\_y: 508

- class\_name: chair

confidence: 0.9526203274726868

center\_x: 186

center\_y: 421

## Using YOLO and Depth Sensor to Reach a Desired Object with a Mobile Robot

7. The depth image is also saved in the variables of the class, along with the detection list generated by the YOLO ROS 2 node, described in the previous section.

def depth\_image\_callback(self, msg):

try:

self.depth\_image = self.bridge.imgmsg\_to\_cv2(msg, desired\_encoding='32FC1')

self.first\_depth = True

except Exception as e:

self.get\_logger().error(f"Failed to convert depth image: {str(e)}")

def yolo\_detections( self, msg ):

self.obj\_list = msg

self.first\_yolo = True

## seek\_and\_go.py – source code

import rclpy

from rclpy.node import Node

from std\_msgs.msg import String

from rclpy.duration import Duration

from rclpy.node import Node

from tf2\_ros import TransformListener, Buffer

from scipy.spatial.transform import Rotation as R

from geometry\_msgs.msg import Twist

from sensor\_msgs.msg import CameraInfo, Image

from cv\_bridge import CvBridge

import math

import time

from threading import Thread

from yolo\_msgs.msg import DetectedObject, DetectedObjectList

class SeekAndGo(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('seek\_and\_go')

self.declare\_parameter('kx', 0.2)

self.declare\_parameter('kyaw', 0.3)

self.kx = self.get\_parameter('kx').get\_parameter\_value().double\_value

self.kyaw = self.get\_parameter('kyaw').get\_parameter\_value().double\_value

self.get\_logger().info("kx: {}, kyaw: {}".format(self.kx, self.kyaw))

self.cmd\_vel\_pub = self.create\_publisher(Twist, '/cmd\_vel', 10)

self.obj\_input\_sub = self.create\_subscription(String, '/object\_to\_seek', self.object\_to\_seek\_input, 1)

self.yolo\_input\_sub = self.create\_subscription(DetectedObjectList, '/yolo/detections/list', self.yolo\_detections, 10)

self.subscription = self.create\_subscription(Image, '/depth/image\_raw', self.depth\_image\_callback, 10 )

self.camera\_info\_sub = self.create\_subscription(CameraInfo, '/camera/camera\_info', self.camera\_info\_callback, 10)

self.timer = self.create\_timer(0.5, self.get\_transform)

self.object\_to\_seek = None

self.new\_object\_to\_seek = False

self.obj\_list = DetectedObjectList()

self.bridge = CvBridge()

self.depth\_image = None

self.tf\_buffer = Buffer()

self.tf\_listener = TransformListener(self.tf\_buffer, self)

self.yaw = 0

self.first\_tf\_data = False

self.first\_cam\_info = False

self.first\_depth = False

self.first\_yolo = False

self.reaching\_obj = False

self.obj\_reached = False

def camera\_info\_callback(self, msg):

camera\_matrix = msg.k

self.cx = camera\_matrix[2]

self.cy = camera\_matrix[5]

self.fx\_inv = 1.0 / camera\_matrix[0]

self.fy\_inv = 1.0 / camera\_matrix[4]

self.first\_cam\_info = True

def depth\_image\_callback(self, msg):

try:

self.depth\_image = self.bridge.imgmsg\_to\_cv2(msg, desired\_encoding='32FC1')

self.first\_depth = True

except Exception as e:

self.get\_logger().error(f"Failed to convert depth image: {str(e)}")

def yolo\_detections( self, msg ):

#self.get\_logger().info("Info")

self.obj\_list = msg

self.first\_yolo = True

def get\_transform(self):

try:

transform = self.tf\_buffer.lookup\_transform('map', 'base\_link', rclpy.time.Time(), Duration(seconds=0.1))

rotation = transform.transform.rotation

quaternion = [rotation.x, rotation.y, rotation.z, rotation.w]

rotation = R.from\_quat(quaternion)

euler\_angles = rotation.as\_euler('xyz', degrees=False)

self.yaw = euler\_angles[2]

self.first\_tf\_data = True

except Exception as e:

self.get\_logger().warn(f"Could not get transform: {e}")

def retrieve\_obj\_from\_list( self, obj ):

det\_obj = DetectedObject()

found = False

i = 0

obj\_list = self.obj\_list

while( found == False and i < len(obj\_list.objects) ):

if( obj.lower() == obj\_list.objects[i].class\_name.lower()):

det\_obj = obj\_list.objects[i]

found = True

i = i+1

return found, det\_obj

def seek\_object( self ):

self.obj\_list.objects.clear()

done = False

found = False

rate = self.create\_rate(10)

v = Twist()

v.angular.z = 0.3

total\_yaw = 0

prev\_yaw = self.yaw

laps = 0

while ( done == False and found == False ):

found, \_ = self.retrieve\_obj\_from\_list( self.object\_to\_seek )

self.cmd\_vel\_pub.publish( v )

total\_yaw = total\_yaw + math.fabs( (math.fabs( self.yaw ) - math.fabs( prev\_yaw )) )

prev\_yaw = self.yaw

if (total\_yaw > 6.2 ):

laps = laps + 1

total\_yaw = 0.0

if( laps > 1 ):

done = True

rate.sleep()

v.angular.z = 0.0

self.cmd\_vel\_pub.publish( v )

return found

def goto\_object( self ):

rate = self.create\_rate(10)

reached = False

done = False

vel\_cmd = Twist()

while( not done and not reached ):

found, det\_obj = self.retrieve\_obj\_from\_list( self.object\_to\_seek )

if( found ):

u = det\_obj.center\_x

v = det\_obj.center\_y

c\_z = self.depth\_image[v, u]

c\_x = c\_z \* ( (u - self.cx ) \* self.fx\_inv)

c\_y = c\_z \* ( (v - self.cy ) \* self.fy\_inv)

x = c\_z

y = -c\_x

e\_x = math.fabs( x )

e\_y = math.fabs( y )

vel\_cmd.linear.x = self.kx\*e\_x

if (e\_x < 0.8 ):

vel\_cmd.linear.x = 0.0

if( e\_y < 0.3 ):

self.obj\_reached = True

self.get\_logger().info("Object reached")

reached = True

done = True

dir = 1

if ( y < 0.0 ):

dir = -1

vel\_cmd.angular.z = dir\*self.kyaw\*e\_y

self.cmd\_vel\_pub.publish( vel\_cmd )

rate.sleep()

else:

self.get\_logger().info("Object lost!")

done = True # Exit from the loop

vel\_cmd.linear.x = 0.0

vel\_cmd.angular.z = 0.0

self.cmd\_vel\_pub.publish( vel\_cmd )

return reached

def main\_loop(self):

while ( not self.first\_tf\_data and

not self.first\_cam\_info and not self.first\_depth and not self.first\_yolo):

time.sleep(0.1)

rate = self.create\_rate(2)

obj\_found = False

v = Twist()

v.angular.z = 0.0

self.cmd\_vel\_pub.publish( v )

while rclpy.ok():

if self.new\_object\_to\_seek:

self.reaching\_obj = True

obj\_found = self.seek\_object()

if( obj\_found == True):

self.get\_logger().info("Requested object seen in the scene, navigate to it")

reached = self.goto\_object()

if( reached ):

self.new\_object\_to\_seek = False

else:

print("Object not found... Try with another object")

self.reaching\_obj = False

rate.sleep()

def object\_to\_seek\_input(self, msg):

self.get\_logger().info("New object to seek: {}".format(msg.data))

self.object\_to\_seek = msg.data

self.new\_object\_to\_seek = True

self.obj\_reached = False

def run( self ):

main\_loop\_thread = Thread(target = self.main\_loop, args = ())

main\_loop\_thread.start()

rclpy.spin(self)

def main(args=None):

rclpy.init(args=args)

node = SeekAndGo()

node.run()

if \_\_name\_\_ == '\_\_main\_\_':

main()

## Conclusion

# 15. Using Large Language Models with ROS 2

## Introduction

## Structure

## Introducing Large-Language Models

### LLM and Robotics

### LLM Basic Components

### LLM Models

## Starting a Local LLM Server Using ROS 2

Installing Ollama on a Linux computer is enough to use the following command:

$ curl -fsSL https://ollama.com/install.sh | sh

When the installation is completed, you will have new commands that can be used to configure and start Ollama.

## Interfacing ROS 2 with Ollama

We additionally need to install the Python API for Ollama.

$ pip3 install ollama

3. To forward a new question to the LLM, the function chat of **ollama** is used. This function requires specifying the model (**llama3.2**), and submitting the request to the model.

def request\_cb(self, msg):

self.get\_logger().info(f'Received request: "{msg.data}"')

response = ollama.chat( model='llama3.2', messages=[{'role': 'user', 'content': msg.data}])

output = String()

4. The response of the LLM is a JSON-style message with different fields. The response of the model is contained in the message and content fields.

output.data = response['message']['content']

self.resp\_pub.publish( output )

def main(args=None):

rclpy.init(args=args)

node = OllamaInterface()

rclpy.spin(node)

if \_\_name\_\_ == '\_\_main\_\_':

main()

### Using Function Calling with ROS 2

### Controlling a Mobile Robot Using LLMs

## Conclusion

# 16. Deep Reinforcement Learning Using ROS 2

## Introduction

**deep reinforcement learning** (DRL) has emerged as a powerful approach, especially for controlling robots with many degrees of freedom, such as humanoids.

Two key tools for deep reinforcement learning: **gymnasium** and **stable\_baselines3**.

## Structure

## Introducing Deep Reinforcement Learning (DRL)

### DRL and Robotics

### Controlling a Robot Using Torque Values

In the previous examples contained in this book, we mainly controlled robotic structures in position or velocity. Another control mode that can be encountered in robotics is the **torque** (or **effort**) control mode.

### Introducing Gymnasium Framework

**Gymnasium** is a Python toolkit designed for developing Reinforcement Learning algorithms. It is the successor of the famous toolkit called **Gym**. The core idea behind Gymnasium is to offer a consistent interface that simplifies the interaction between the robot (the agent) and an environment (the task or problem to be solved).

## Setting the Cart-Pole Simulation Scene

## Integrating Gymnasium and ROS 2

o

### Training a Robot Using Gymnasium and stable\_baselines3

To simplify the training step, we will use another library, typically connected with Gymnasium, that is, the **Stable-Baselines3 (SB3)**.

### Controlling a Robot Using Deep Reinforcement Learning

## Conclusion