

# ROS 2

## Sensor sampling and image processing

Roberto Masocco  
[roberto.masocco@uniroma2.it](mailto:roberto.masocco@uniroma2.it)

University of Rome Tor Vergata  
Department of Civil Engineering and Computer Science Engineering

June 7, 2023



**TOR VERGATA**  
UNIVERSITY OF ROME

School of Engineering

# Recap

**ROS 2** offers a common framework for the development of **robotics software**, providing services for:

- **organizing** and **building** a **distributed** software architecture;
- establishing mostly self-configured **inter-process communication** among modules;
- modules **configuration** and **management**.

The last two lectures will show **real applications** of these tools.

**New code examples are available.**

This lecture is [here](#).

# Roadmap

- 1 Sensor sampling
- 2 Image processing
- 3 Software tools of the trade
- 4 Examples: a target detection pipeline

# Roadmap

- 1 Sensor sampling
- 2 Image processing
- 3 Software tools of the trade
- 4 Examples: a target detection pipeline

# Sensor sampling basics

From theory to practice

**Sampling** a sensor consists of reading **measurements** from it, to be fed to a control loop or some other subsystem.

It requires:

- the definition of a **sampling frequency**;
- the implementation of an **encoding**;
- the application of **post-processing** steps (e.g., **filtering**).

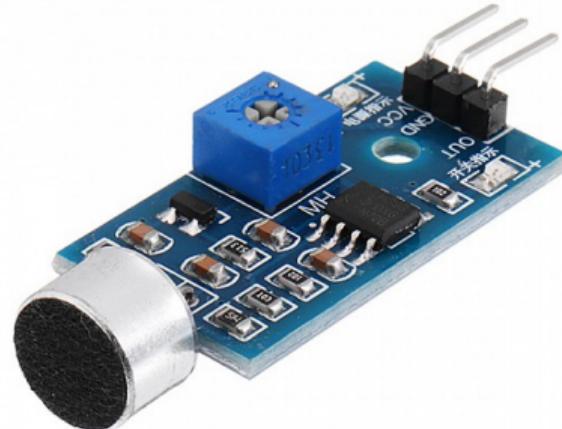


Figure 1: Analog sound sensor.

# Sensor sampling with ROS 2

## Driver modules

Sampling is generally handled by **microcontrollers**, but when the sampling frequency is not too high, e.g., down to some ms, it can be carried out by a higher-level device.

To implement a ROS 2 sensor sampling module, one has to develop a **driver node**, i.e., an application that:

- configures the **sensor hardware** to run as required;
- ensures **stable sampling frequency** and **low jitter**;
- outputs data with a **standard interface** and **low latency**.

The achievement of the first goal depends on the **sensor**, the second on the **system** (hardware and software!), while the third one is solved by ROS 2 (**messages**, **QoS**).

**Must take the best of both worlds: robotics and system programming!**

# Anatomy of a driver node

## Guidelines and best practices

In essence, a **driver node** always consists of:

- an **enable service**, to be called to start or stop the sampling;
- a **hardware configuration** routine, to be run at startup or when enabled;
- a **sampling loop**, to be run at a fixed frequency in a separate **thread**;
- a **publisher** using a common message type and an appropriate QoS policy;
- a set of **parameters** to configure the sensor and the sampling loop;
- **launch files** and **configuration files**, to configure remapping rules and node behaviour.

# Roadmap

- 1 Sensor sampling
- 2 Image processing
- 3 Software tools of the trade
- 4 Examples: a target detection pipeline

# Vision in robotics

## Sensors characteristics

**Visual sensors**, commonly referred to as **cameras**, are sensors that provide **images** as output, encoded in **frames**, i.e., **matrices** of data points. They are usually made of:

- one or more **optical lenses**, to focus light on the sensor;
- a **sensor**, to convert light into electrical signals;
- a **processing unit**, to convert electrical signals into images, optionally applying **post-processing** steps.



Figure 2: RGB USB camera.

# Vision in robotics

## Sensors characteristics

**Light might not belong to the visible band of the spectrum.**



Figure 3: Intel RealSense D435i depth camera: RGB and IR sensors.

# Vision in robotics

## Sensors characteristics

**Cameras** are usually characterized by:

- **resolution**, i.e., the number of pixels in the image;
- **field of view**, i.e., the angular extension of the scene (horizontal and vertical);
- **frame rate**, i.e., the number of frames per second;
- **dynamic range**, i.e., the ratio between the maximum and minimum measurable light intensity.



Figure 4: Intel RealSense T265 tracking camera: two fisheye sensors.

# Vision in robotics

## Sensors characteristics

**Cameras, and image processing algorithms in general, are usually characterized by a trade-off between resolution and frame rate.**



Figure 5: ZED Mini stereo tracking camera.

# Vision in robotics

## Main use cases

Visual sensors are usually employed in robotics for:

- **object detection**, *i.e.*, identifying objects in the scene;
- **object tracking**, *i.e.*, following objects in the scene;
- **localization**, *i.e.*, estimating the robot's position in the environment;
- **mapping**, *i.e.*, building a model of the environment.

Using the sensor is not enough: **algorithms** are needed to perform these tasks.  
Such algorithms usually run in separate modules.

# Types of frames

## RGB frame



Figure 6: RGB frame: each pixel contains at least the intensities of the red, green and blue components of the corresponding point in the scene.

# Types of frames

## IR frame



Figure 7: IR frame: each pixel contains the intensity of the corresponding point in the scene.

# Types of frames

## Depth map frame



Figure 8: Depth map frame: each pixel contains the distance of the corresponding point from the camera.

# Lens distortion

## Camera calibration and rectification

Pinhole cameras generally introduce radial distortion in the images they produce, i.e., straight lines appear curved. This is due to how the light enters the camera through the lens.

A camera calibration procedure is needed to estimate the parameters of the distortion model, so that a rectification map can then be applied to each frame.



Figure 9: Checkerboard pattern used for camera calibration in the presence of radial distortion.

# Lens distortion

## Camera calibration and rectification

The calibration procedure usually involves moving a **checkerboard** of known dimensions in front of the camera, and taking several pictures of it from different angles.

An **estimation model** can then be used to estimate the parameters of the distortion model, which can then be used to build the rectification map.

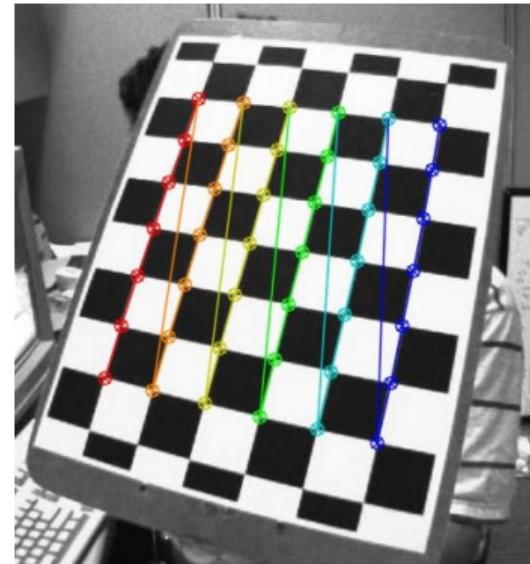


Figure 10: A frame from a checkerboard calibration procedure.

# Lens distortion

## Camera calibration and rectification

The **rectification map** is a **lookup table** that associates each pixel in the original frame with a pixel in the rectified frame, **interpolating** when necessary.

It is usually stored in **configuration files**, and must be applied to each frame before it can be used by other algorithms.

ROS 2 offers the [cameracalibrator](#) tool in the `camera_calibration` package to perform camera calibration.



Figure 11: Checkerboard pattern after rectification.

# Roadmap

- 1 Sensor sampling
- 2 Image processing
- 3 Software tools of the trade
- 4 Examples: a target detection pipeline

# OpenCV

The de-facto standard library for computer vision

OpenCV is the largest open-source library for real-time **computer vision** and **image processing**. It has been developed initially by Intel, and evolved in a **cross-platform** library supporting **C++** and **Python**.

It ships with hundreds of **APIs** and **algorithms** to perform a wide variety of image processing and computer vision tasks and computations.



Figure 12: OpenCV logo.

# OpenCV

The de-facto standard library for computer vision

In Linux, it is available as **binary packages**, but the full range of features can be enabled only by **configuring a source build**.

Its **elementary data type** is a **matrix**:  
`cv::Mat`.

It is **heavily optimized** to run on:

- **parallel CPUs**;
- **GPUs** (e.g., `cv::cuda`, `cv::ocl`);
- **embedded devices** (e.g., Nvidia Jetson).



Figure 12: OpenCV logo.

# ROS tools for image processing

## A quick overview

ROS 2 provides a wide range of **tools** to acquire, process, and transmit images. The most important ones are:

- `sensor_msgs/Image`: the standard ROS **message type** to transmit **images** over the DDS layer;
- `sensor_msgs/CameraInfo`: the standard ROS **message type** to transmit and parse **camera calibration parameters**;
- `image_transport`: a ROS package to **transmit images** over the DDS layer;
- `camera_info_manager`: a library to parse, use, and store **camera calibration parameters** from configuration files and messages inside ROS nodes.

# ROS tools for image processing

## The Image message

---

```
1 std_msgs/Header header
2
3 uint32 height
4 uint32 width
5
6 # This can be RGB, BGR, RGBA, BGRA, YUV, Bayer, etc.
7 string encoding
8
9 uint8 is_bigendian
10 uint32 step
11 uint8[] data
```

---

**Listing 1:** Definition of the sensor\_msgs/msg/Image message.

# ROS tools for image processing

## Sending images over the DDS

Using images and video streams in general over the DDS layer is **not trivial**, since:

- ① we would like some kind of **specialized transport strategy** to optimize throughput and latency;
- ② the DDS specification is **not optimized** to transmit **large** and **variable-size** data chunks over potentially **lossy** networks, e.g., WiFi.

`image_transport` is a ROS 2 library that provides **wrappers** for **topic publishers** and **subscribers**, allowing to transmit images using different **transports**:

- `raw`: the default transport, which uses the `sensor_msgs/Image` message;
- `compressed`: uses the `sensor_msgs/CompressedImage` message, automatically converting frames to JPEG or PNG format;
- whatever you want to implement as a **plugin**.

# ROS tools for image processing

## A note on topic statistics

### Beware!

When measuring image topics publishing rates with `ros2 topic hz`, especially **over a lossy network**, you may get **wrong results**. This is because of:

- the **Python DDS implementation**;
- the **hz** Python tool buffering messages into a window-based filter.

To check the real sampling and receiving rates, **place publishers of ordinary messages** in your code on both ends, and inspect those instead!

Suggestion: `std_msgs/Empty`.

# Roadmap

- 1 Sensor sampling
- 2 Image processing
- 3 Software tools of the trade
- 4 Examples: a target detection pipeline

# ArUco detection pipeline

A real example from LDC22

Examples for this lecture are in the [cpp/image\\_processing](#) directory.

There are **three nodes**:

- ① `ros2_usb_camera`: acquires images from a USB camera;
- ② `aruco_detector`: detects ArUco markers in a video stream;
- ③ `rqt_image_view`: forked version of the official ROS 2 video stream visualizer.

These three nodes form a **pipeline** to detect **ArUco markers** in a video stream, and show them in a **GUI**. They are **completely configurable**, and **optimized** to run on **GPU** and similar hardware. They make use of **all the features we have seen so far**, plus [composition](#): a way to include **multiple nodes** in a **single running process**, loading and unloading them dynamically, scheduling their work, and benefitting of **zero-copy** data transfer and shared address space.

**Multiple instances of these nodes were active during the Leonardo Drone Contest 2022!**