# Localization and mapping

## From EKF to SLAM

Roberto Masocco
roberto.masocco@uniroma2.it

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

June 7, 2024

TOR VERGATA
UNIVERSITY OF ROME

School of Engineering

# Roadmap

1. The perception problem
2. The mapping problem
3. Simultaneous Localization And Mapping
4. Common interfaces
5. The tf2 library

# Roadmap

# The perception problem

## Definition

To be able to operate autonomously, a robot must continuously answer the following questions:

- **Where am I?**
- **What is this place?**

Thus, it must be able to **perceive** the environment, gathering information useful to:

- **localize itself**, *i.e.*, continuously estimate its **pose** as **both position and orientation** in 3D space;
- **map the environment**, *i.e.*, build a **representation** of the environment useful to **navigate** within it.

# The perception problem

## Challenges

The perception problem is challenging because:

- the environment may be **partially observable**, *i.e.*, the robot can only perceive a **subset** of it, and need to update its information in real time;
- the environment may be **dynamic**, *i.e.*, it can change over time;
- measurements are always subject to **noise**.

The perception problem is usually solved by **sensor fusion**, *i.e.*, combining information from **multiple sensors** to obtain a more **accurate** and **reliable** estimate of the environment, possibly accounting for **sensor faults**.

# The perception problem

## Tools for the job

The tools that robots use to gather **measurements** from the environment are called **sensors**.

They can be classified as:

- **proprioceptive**, *i.e.*, measuring robotic interaction with the environment (*e.g.*, **encoders**, **GPS**, **IMUs**);
- **exteroceptive**, *i.e.*, measuring the environment itself (*e.g.*, **cameras**, **LiDARs**, **radars**);
- **interoceptive**, *i.e.*, measuring the robot's internal state.

# The perception problem

## Tools for the job

As any other measurement tool, sensors are based on **physical principles** and **energy exchanges**, translating the information they gather into **electrical signals** that can be acquired and/or processed by a computer.

They are usually characterized by at least:

- a **digital** or **analog encoding** of the measurement;
- a **frame of reference** in which the measurement is expressed;
- **accuracy** and **uncertainty** parameters.

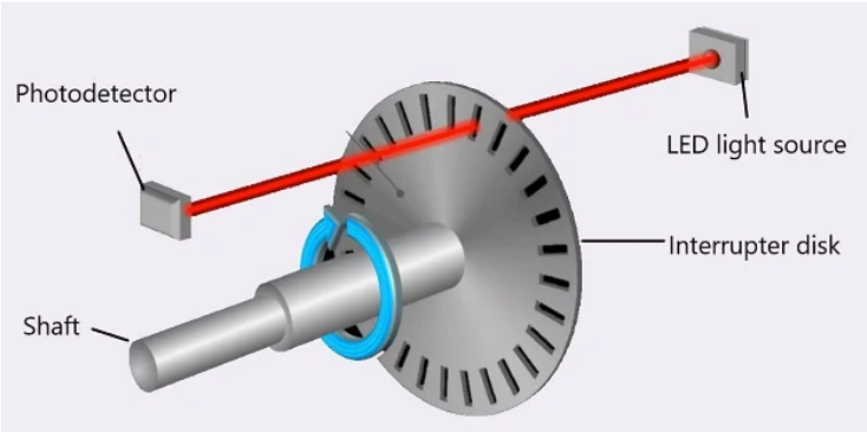


Figure 1: Rotary encoder working principle.


Figure 2: GPS module for drones.

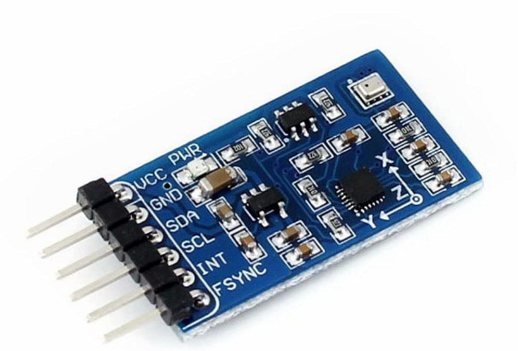


Figure 3: Inertial Measurement Unit (IMU).

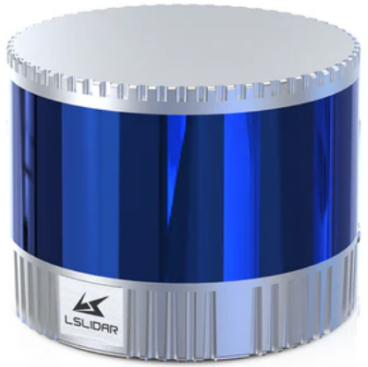


Figure 4: Light Detection and Ranging (LiDAR) sensor.


Figure 5: ZED 2i stereo camera.

# Roadmap

# The mapping problem

## Definition

Using **exteroceptive sensors**, a robot can gather information about the environment, which can be used to build a **map** of it.

A **map** is a **representation** of the environment, in a format that the robot can **understand**, **parse**, and **store**.

The utimate goal of mapping is twofold:

- to enable the robot to **localize itself** within the environment;
- to enable **safe navigation** of the robot within the environment.

# The mapping problem

## Challenges

Given the utility requirement of a map, the mapping problem must be **continuously solved in real time**.

Thus, it is challenging because:

- routines must be **efficient**, and run at a sufficiently **high rate**;
- the map must be **accurate**, and **reliable**;
- the map must be in a format that is as much **easy to load and parse** as possible, taking up as little **memory** as possible;
- the map must stay **up-to-date**, and **consistent** with the environment.

# The mapping problem

## Tools for the job

The most important tool for the mapping problem is the **occupancy grid**, a representation of the environment as a **grid** of **cells**, each of which is **occupied** or **free**.

The occupancy grid is a **probabilistic** representation, where each cell is associated with a **probability** of being occupied or free.

The occupancy grid is usually built using **LiDAR** or **camera** depth data, and is updated in real time as the robot moves.

The occupancy grid is the most common representation for **local** and **global** maps.

To efficiently store an occupancy grid, **tree-like data structures** are often employed (*e.g.*, **octrees**).

The second most important class of tools are **navigation algorithms**, which use the map to plan a **safe** and **efficient** path for the robot to follow.

The definition of such algorithms involves **geometry**, as well as **optimization** and **search** techniques.

They usually rely on two mathematical subjects:

- **topology**, to define the **connectivity** of the map (*e.g.*, Voronoi tessellation);
- **graph theory**, to define the **best way** of moving from one free cell to another (*e.g.*, Dijkstra's, A$^\star$ algorithms).

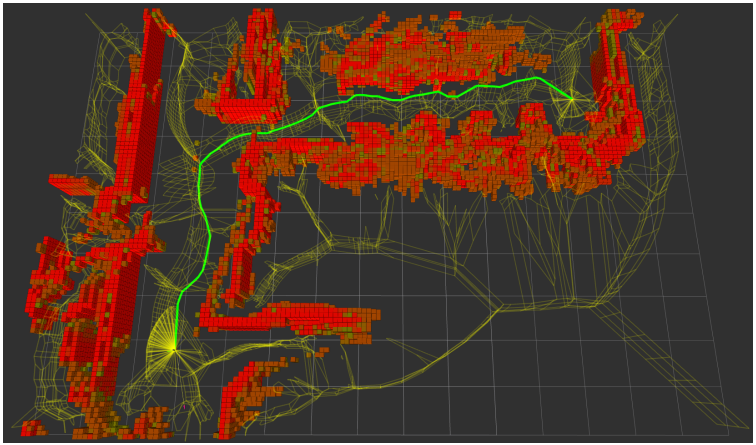

Figure 6: Mapping and navigation algorithms execution.

# Roadmap

# Simultaneous Localization And Mapping

## Definition

The **SLAM** problem is a **chicken-and-egg** problem: a robot must **localize itself** within an environment, while **mapping** it.

The ultimate goal of SLAM is to enable the robot to **navigate** within the environment, while **updating** the map as it moves.

The robot must be able to **efficiently** and **accurately** build a map of the environment, while **localizing itself** within it, with respect to either the origin of the map or the starting point of the robot itself (*i.e.*, with either some or none **prior notion of the environment**).

To solve this problem, **sensor fusion** techniques are often employed, mixing data coming from **heterogeneous sensors** and accounting for **sensor faults**.

Typically, SLAM algorithms rely on recognizable **features** of the environment to build the map and detect motion.

While a SLAM system builds a **map** of the environment, it can detect whether it is exploring a zone that it has **already visited**.

When this happens, the system can **close the loop** by **matching** the current "view" with a **previous one**, thus **correcting** the map and the robot's pose.

This process is called **loop closure** and is a key feature of SLAM algorithms.

Loop **detection** and **closing** must also be performed in real time, as well as the subsequent **corrections**; efficient optimization algorithms and data structures are crucial.

Figure 7: Loop closure of the ORB-SLAM2 algorithm.

Sensors:

- **LIDARs** for direct environment mapping through depth information;
- **Cameras** to infer the environment structure through image processing;
- **IMUs** to account for the robot's motion and correct sampled data;
- **GNSS** to have a slow, but reliable global position estimate.

Plus all the **algorithms** and the **mathematical tools** we discussed in the context of **mapping**.

**Cartographer** is a **2D** SLAM algorithm developed by Google.

Meant for **offline** floor plan generation, it has been ported in **ROS** for mobile robot navigation.

It used **LiDAR** laser scans, corrected by **IMU** data, to build 2D **submaps** of the **surrounding** environment.

Such maps are then used to **build and update** a **global map**, gluing submaps together and optimizing the result.

Features are particular **environment structures** that are used to detect loop closures and estimate the robot's trajectory.

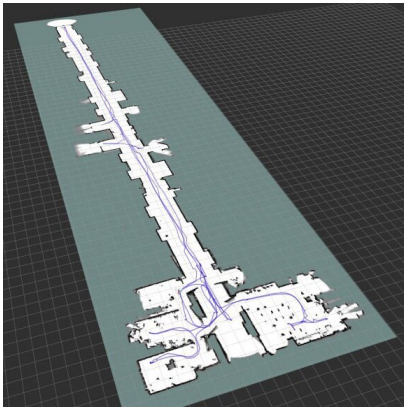

Figure 8: Execution of the Cartographer SLAM algorithm.

**ORB-SLAM** is a **visual** SLAM algorithm that uses **monocular** or **stereo** cameras to build a map of the environment.

It relies on **binary ORB features** to detect and match points in the environment, building a **covisibility graph** of **keyframes**: relevant views for pose estimation.

The graph, and thus, the map and the camera pose estimate, are optimized with **bundle adjustment**.

Latest versions also include **IMU** samples in the bundle adjustment cost function.

Figure 9: ORB-SLAM2 algorithm execution on a ROS 2 dataset (bag).

# Roadmap

# Common interfaces

A standard ROS 2 installation offers many **interface packages** (*i.e.*, messages), to provide **standard data types** to communicate sensor measurements and related data.

The most important are:

- `sensor_msgs`, for **sensor measurements**;
- `geometry_msgs`, for **geometric data**;
- `nav_msgs`, for **navigation data**.

It is suggested to **always use these message types**, plus common best practices, to ensure full interoperability between sensor drivers and localization and mapping algorithms.

Try to `ros2 interface show` these messages to understand their structure!

# sensor_msgs

## Common interfaces for sensors

- `Imu`
- `JointState`
- `CameraInfo` and `Image`
- `LaserScan`
- `PointCloud2`
- `Temperature`
- `NavSatFix`
- `Illuminance`
- ...

# geometry_msgs

## Common interfaces for algebraic data

- `Vector3Stamped`
- `QuaternionStamped`
- `PoseWithCovarianceStamped`
- `TwistWithCovarianceStamped`
- `TransformStamped` (used by `tf2`!)
- `AccelWithCovarianceStamped`
- ...

# nav_msgs

## Common interfaces for navigation data

- `Odometry`
- `Path`
- `OccupancyGrid`
- ...

# Roadmap

# Rigid transformations

When a robot moves in space, it is important to keep track of its **position** and **orientation** with respect to a **reference frame**.

Sensors measuring this information, as well as many more, are **mounted** on the robot, in fixed positions and orientations.

To process these measurements, they must first be **transformed** from the **body frame** into a common reference frame, usually called:

- **world frame** (world origin), in the case of **global localization**;
- **local frame**, or **odom frame** (robot starting point), in the case of **local localization**.

Such **rigid transformations** are **isometries**. They must be applied to almost every sensor measurement, and are usually **composable**.

We would like the middleware to provide tools to do this almost automatically...

# The tf2 library

`tf2` is the **standard ROS 2 library** to handle rigid transformations.

It allows to:

- **broadcast** and **listen** to transformations, thanks to appropriate buffering subscribers and publishers;
- **transform** any kind of sensor data from one frame to another, making efficient computations in C++ code relying on the `Eigen` mathematical library;
- broadcast **robot descriptions** from `URDF` files, listing links and joints and how they are connected, resulting in a **tree-like structure**;
- **command-line tools** to inspect the current tree status, and broadcast custom transformations.

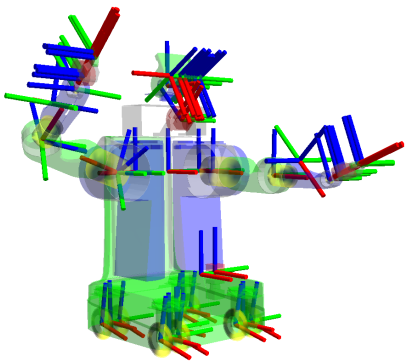

Figure 10: Example of robot description with tf2.

# The tf2 library

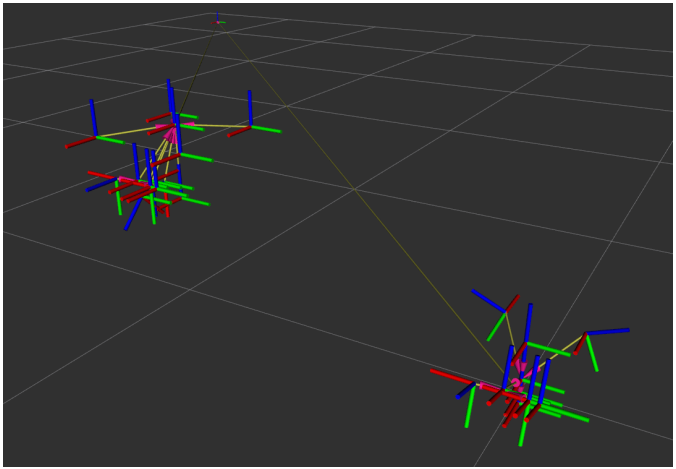

Figure 11: Broadcasted robot descriptions, plus real-time transformations given by localization systems.