Lab04

Localization with EKF

Objectives

- Develop a ROS 2 package to run EKF on the robot.
- Localize your robot using landmark measurements.
- Fuse measurements from different sensors (odometry, IMU) to improve localization accuracy.

Requirements

- The Python script explained during lectures (motion model, sensor model, ekf), present in the portale della didattica (materiale/lecture_notebooks/..)
- An updated version of `turtlebot3_simulations`
- An updated version of `turtelbot3_perception`

Update your repositories

In order to update a Github repository:
1. Open a terminal and change directory to the repository
2. Run the command `git pull`
3. Perform this operation for every repository that you need to update

Once you updated all the repositories, perform the following in your ROS 2 workspace:
1. Remove the folders `build`, `log` and `install`
2. Install the dependencies with the command
```
rosdep install --from-path src --ignore-src -y -r
```
3. Build the workspace using the command `colcon build`

Exercise

Main objective: realize a ROS 2 package for EKF localization. The package will be validated in Gazebo simulation and then deployed on the robot.
The lab exercise is divided into 3 Tasks. Tasks 1 and 2 can be prepared before the experimental session in the lab using the Gazebo simulation.
Suggestion: if you are running out of time in the lab, skip Task 2 and try your solution (Task 1) on the real robot (Task 3).

Task 1 [3 points]

Starting from the EKF scripts that were presented during lectures, implement inside a ROS 2 node an EKF for tracking the robot position given landmarks measurements.

State of the filter: the state of the filter is the one reported below, where x, y, θ represent the position of the robot in the global reference frame.

$$\mu = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Prediction: perform the prediction step at a **fixed rate** of 20 Hz using the **velocity motion model**. Use as **command** the most recent v and ω taken from the `/odom` **topic**. Data is used in this way because the velocity reported by the robot is closer to the real velocity than the command sent on `/cmd_vel` topic. **Create a timer in your node to perform the prediction operation of EKF.**

Update: measurements are provided as **range and bearing** of **landmarks** in the field of view of the robot. Measures are published on **topic** `/landmarks` (or `/camera/landmarks` on the real robot) inside a message of type `landmark_msgs/msg/LandmarkArray`.
Perform the update operation of the EKF inside the subscription callback for each landmark listed in the message.
Once all the updates are done, **publish** the estimated state in a message of type `nav_msgs/msg/Odometry` on topic `/ekf`. Make sure to **fill** the `header.stamp` field with the current time taken from `self.get_clock().now().to_msg()`.

Landmarks coordinates are provided in a yaml file inside the `turtlebot3_perception` package (`turtlebot3_perception/turtlebot3_perception/config/landmarks.yaml`) in the following format:

```yaml
landmarks:
  id: [id0, id1, ..., idN]
  x: [x1, x2, ..., xN]
  y: [y1, y2, ..., yN]
  z: [z1, z2, ..., zN]
```

Task 2 [1 point]

Extend the state of the filter to also include linear and angular velocity. **The new state is the one reported below.**

$$\mu = \begin{bmatrix} x \\ y \\ \theta \\ v \\ \omega \end{bmatrix}$$

Prediction: create **new functions to predict** the state and to compute **its Jacobians G** and **V**. You can write them by hand or you can modify the sympy based functions that you find in the `probabilistic_models.py` module (`sample_velocity_motion_model` and `velocity_mm_simpy`). It is reported here the function $g(u, x)$ and its Jacobian with respect to the new state $G(u,\ x)$. You can notice that the part of the Jacobian in the red box is the Jacobian matrix you were using in the first version of the filter.

$$\begin{bmatrix} x' \\ y' \\ \theta' \\ v' \\ \omega' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ v \\ \omega \end{bmatrix} + \begin{bmatrix} -\frac{v_t}{\omega_t}\sin\theta + \frac{v_t}{\omega_t}\sin(\theta + \omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\theta - \frac{v_t}{\omega_t}\cos(\theta + \omega_t\Delta t) \\ \omega_t\Delta t \\ 0 \\ 0 \end{bmatrix} + \mathcal{N}(0, R_t)$$

$$\left[\begin{array}{ccc|cc} 1 & 0 & -\frac{v\cos(\theta)}{w} + \frac{v\cos(dtw+\theta)}{w} & -\frac{\sin(\theta)}{w} + \frac{\sin(dtw+\theta)}{w} & \frac{dtv\cos(dtw+\theta)}{w} + \frac{v\sin(\theta)}{w^2} - \frac{v\sin(dtw+\theta)}{w^2} \\ 0 & 1 & -\frac{v\sin(\theta)}{w} + \frac{v\sin(dtw+\theta)}{w} & \frac{\cos(\theta)}{w} - \frac{\cos(dtw+\theta)}{w} & \frac{dtv\sin(dtw+\theta)}{w} - \frac{v\cos(\theta)}{w^2} + \frac{v\cos(dtw+\theta)}{w^2} \\ 0 & 0 & 1 & 0 & dt \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

Update: beside landmarks detection model `ht_landmark`, **consider wheel encoder** data and **IMU data** to correct the v, ω components of the state.
Use the wheel encoder data (in `/odom`) to update both v and ω with a function `ht_odom`, and the IMU to only update ω with a function `ht_imu`.
Measurement functions in matrix form are reported below, compute the associate Jacobian `Ht_odom`, `Ht_imu` with respect to the new state, necessary for the correction step of the filter. Also, update the Jacobian of the landmark measurement function `Ht_landmark` to consider the new state. Based on the expected confidence in the measurements, define the uncertainty parameters `std_odom` and `std_imu` to define the covariance matrix **Q** and to sample from the new measurement models.

$$h_{odom}(\bar{\mu}_t) = \begin{bmatrix} v_t \\ \omega_t \end{bmatrix} + \mathcal{N}(0, Q_{odom})\ , \quad h_{imu}(\bar{\mu}_t) = \begin{bmatrix} \omega_t \end{bmatrix} + \mathcal{N}(0, Q_{imu})$$

Task 3 [1 point]

Run the ROS 2 package for EKF localization on the real robot. You can choose between the implementation used in Task 1 and the one developed for Task 2.

Report requirements

- Provide a concise description and comments on the main structure of your ROS 2 program (do not copy your code in the report, just highlights the main elements and the workflow: nodes, publishers/subscribers to relevant topics and parameters)

- **[Task 1]**: **Run your filter in the simulation environment.** Record the `/ground_truth`, `/odom` and `/ekf` topics. Then:
  - **Make plots** comparing the position and orientation reported in the three topics
    - Make a plot for each state (x, y, θ) vs. time and compare the different topics. You shall put the state reported by all the three topics in the same plot. (Examples provided in python-crash-intro repository Module3)
    - Plot the (x, y) trajectory on the 2D plane using the robot's pose data from the three topics.
    - Comment the results
  - **Compute the following metrics** with respect to the ground truth: Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) Further instructions will be provided on this point in a separate file.

- **[Task 2]**: perform the same experiments and obtain the same plots and performance results of Task 1 for the extended EKF state with sensor fusion described in Task 2. Make some significant comparison comment on the different results obtained.

- **[Task 3]**: **Run your filter on the real robot.** You can choose between the implementation of the EKF you developed for Task 1 or Task 2. Record the `/odom` and `/ekf` topics. Then:
  - Make plots comparing the position and orientation reported in the two topics. For `/odom` you can subtract the first pose to all the other poses to align to the origin. Then:
    - Make a plot for each state component over the time length of the experiments, and compare the different topics.
    - Plot the (x, y) trajectory using the data collected by the two topics.
    - Comment on the results.

How to test your algorithms

Simulation Environment

The following operations shall be executed on your PC.

A simulation is provided to ease the development and testing of your code. To launch the simulation run the following command in a workspace with both `turtlebot3_simulations` and `turtelbot3_perception` packages. (Mac users should run the `ignition` version of the command in the same way adopted for Lab02)

```
ros2 launch turtlebot3_gazebo lab04.launch.py
```

The simulation environment is the one shown in Figure 1, with the **white columns** acting as **landmarks**. The orange labels are the landmarks IDs.

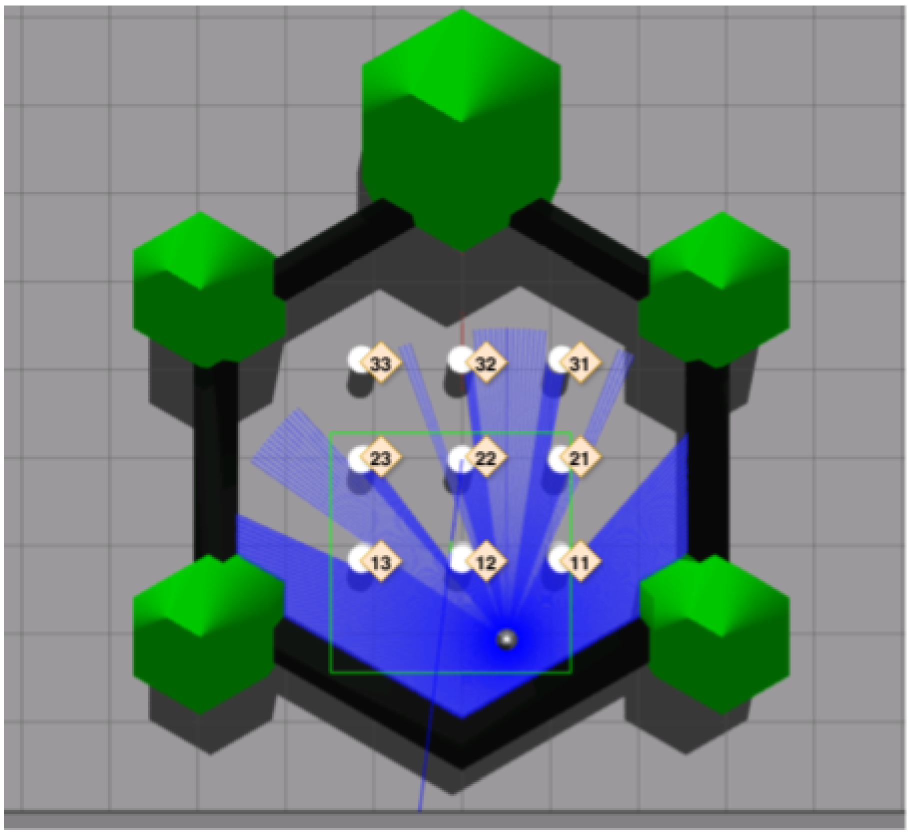

Figure 1. Simulation environment with labelled landmarks

Real Robot

The following operation shall be performed on the robot.

Install the camera

1. Mount the camera on the robot using the spacers and bracket as shown in Figure 2. Secure the cable to the side of the robot using the cable ties provided in the camera box.

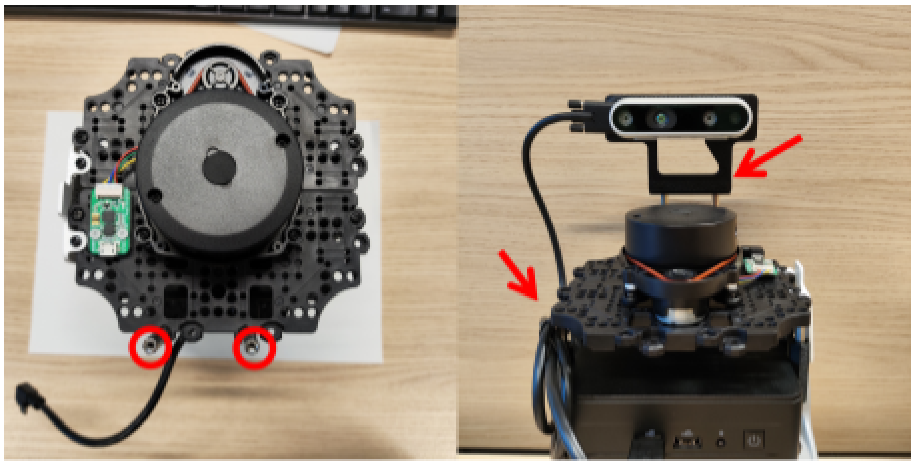
Figure 2. Realsense d435 camera mounted on TB3. ***Left****: red circle highlight the position of the spacers.* ***Right****: red arrows highlight the correct position of the bracket and the coiled cable.*

2. In two different terminals start the camera driver and the landmark detector using the following commands

   ```
   # Terminal 1
   ros2 launch turtlebot3_perception camera.launch.py

   # Terminal 2
   ros2 launch turtlebot3_perception apriltag.launch.py
   ```

3. If landmarks are present, they will be published on topic `/camera/landmarks` at approximately 6 Hz.