Politecnico di Torino

Department of Electronics and Telecommunications

Master’s degree in Mechatronic Engineering

2025/2026

Sensors, Embedded Systems and Algorithms for Service Robotics

Report 2

22/12/2025

Group n° 207

Marco Desogus – S346971

Paolo Ragusa – S344185

Siro Interlandi –S343898

Marco Conte – S346753

1 INTRODUCTION

This report presents the implementation and validation of an Extended Kalman Filter (EKF) in a ROS 2 framework for mobile-robot state estimation. The laboratory activity was structured into:

- experimental analysis of the motion and sensor models under noise
- the development of the EKF node and its execution in a Gazebo simulation, and the subsequent evaluation on a real robot.
- Particular attention is devoted to comparing a 3-state formulation $[x, y, \theta]$with an extended 5-state formulation $[x, y, \theta, v, \omega]$, highlighting the impact of model assumptions, Jacobian-based linearization, and landmark availability on estimation performance.

2 MOTION AND SENSOR MODELS

The motion model adopted is a velocity-based model. As the name implies, it relies on the system velocities to compute the next state. The model inputs consist of the current state x, the control command u, defined by a translational velocity v and an angular velocity w, and the noise parameters. The uncertainty is modelled as Gaussian noise, characterized by the standards deviations associated with both the linear and rotational velocities.

The model is executed 500 times using the same control and state input in two separate experiments. The difference between the two runs lies in the uncertainty configuration: in the first experiment, a high uncertainty is applied to the linear velocity, whereas in the second experiment a high uncertainty is applied to the angular velocity.

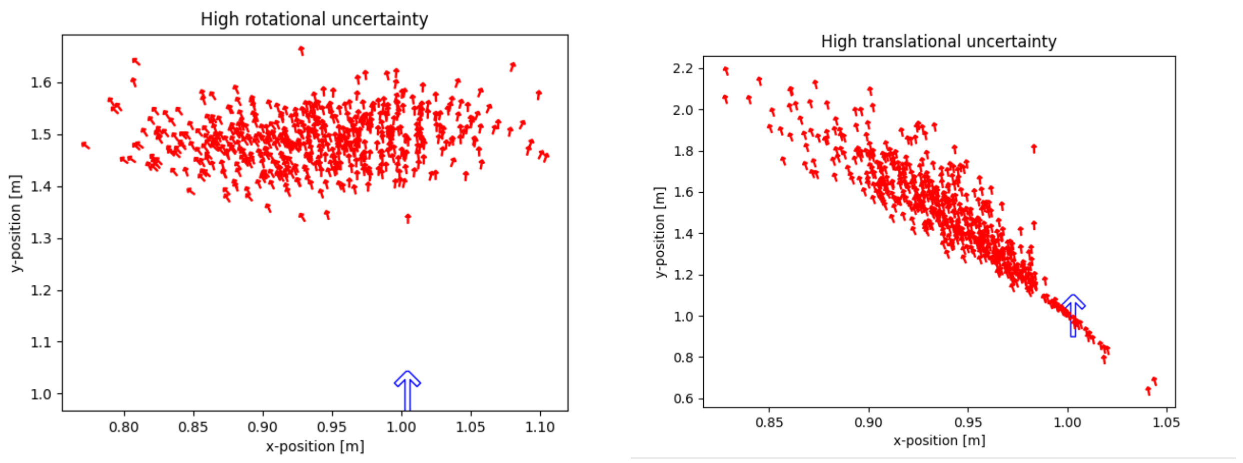


Figure 1: How the model uncertainty affects the state computation

The noise parameters of the model play a crucial role in determining the accuracy of the state prediction. As shown in Fig.1, a high rotational uncertainty causes the predicted states to spread along a curved pattern, while maintaining relatively higher accuracy in the linear displacement. Conversely, when a high level of noise affects the translational motion, the predicted states are scattered primarily along a straight line. The initial state x is represented in blue.

The sensor model adopted is a landmark-based model, whose objective is to compute a new possible robot state from the current state and the measurements obtained from a landmark. Each measurement consists of two quantities: the range, defined as the relative distance between landmark and robot, and the bearing, defined as the relative orientation angle with respect to the landmark. As in the motion model, noise influences the results; however, an additional key factor is the number of landmarks visible to the robot at a given time. In this experiment, the robot is simulated to observe only a single landmark. This limitation results in the estimated states being distributed around the landmark along a circular pattern, due to the ambiguity in the robot's actual position. Figure 2 illustrates the outcome of the experiment: the initial state is shown in blue, while the 1000 newly computed possible states are shown in red. In this case, the noise levels associated with range and bearing are equal, resulting in no noticeable differences in the linear or angular uncertainties.

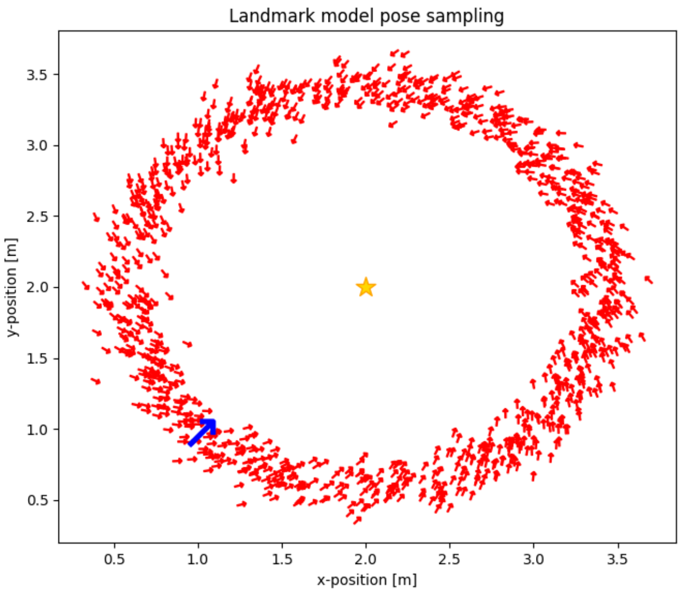


Figure 2:Landmark model experiment result

Unfortunately, both the motion and sensor models are nonlinear functions, which prevents the direct application of the standard Kalman Filter to a robotic system. To address this limitation, an Extended Kalman Filter (EKF) is employed. The EKF overcomes this issue by locally linearizing the nonlinear models, thereby allowing the Kalman Filter framework to be applied. Local linearization is achieved through a first-order Taylor expansion of the nonlinear functions, resulting in the computation of the corresponding Jacobian matrices. In particular, the velocity-based motion model is linearized to obtain the Jacobian with respect to the state, denoted as Gt, and the Jacobian with respect to the state, denoted as Vt. Similarly, the sensor model is linearized with respect to the state, yielding the Jacobian matrix Ht.

The motion model will be used in the prediction step of the EKF and its equation is:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = g(u_t, x_{t-1}) + \mathcal{N}(0, R_t) = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\theta + \frac{v_t}{\omega_t}\sin(\theta + \omega_t \Delta t) \\ \frac{v_t}{\omega_t}\cos\theta - \frac{v_t}{\omega_t}\cos(\theta + \omega_t \Delta t) \\ \omega_t \Delta t \end{pmatrix} + \mathcal{N}(0, R_t)$$

Figure 3: Motion model non-linear equation

By using the SymPy library in Python, it is possible to compute the symbolic expressions of the Jacobian matrices. In this case the resulting expressions are:

$$\begin{bmatrix} -\frac{\sin(\theta)}{w} + \frac{\sin(dtw+\theta)}{w} & \frac{dtv\cos(dtw+\theta)}{w} + \frac{v\sin(\theta)}{w^2} - \frac{v\sin(dtw+\theta)}{w^2} \\ \frac{\cos(\theta)}{w} - \frac{\cos(dtw+\theta)}{w} & \frac{dtv\sin(dtw+\theta)}{w} - \frac{v\cos(\theta)}{w^2} + \frac{v\cos(dtw+\theta)}{w^2} \\ 0 & dt \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & -\frac{v\cos(\theta)}{w} + \frac{v\cos(dtw+\theta)}{w} \\ 0 & 1 & -\frac{v\sin(\theta)}{w} + \frac{v\sin(dtw+\theta)}{w} \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 4: Jacobians of the motion model with respect to the input (left) and with respect to the state (right)

The same thing can be done for the sensor model, and its linearized function will be used in the update step of the EKF:

$$\begin{bmatrix} \frac{-m_x+x}{\sqrt{(m_x-x)^2+(m_y-y)^2}} & \frac{-m_y+y}{\sqrt{(m_x-x)^2+(m_y-y)^2}} & 0 \\ -\frac{-m_y+y}{(m_x-x)^2+(m_y-y)^2} & -\frac{m_x-x}{(m_x-x)^2+(m_y-y)^2} & -1 \end{bmatrix}$$

Figure 5: Jacobian of the sensor model with respect to the state

3 ROS 2 IMPLEMENTATION OF THE EXTENDED KALMAN FILTER

Task1 implementation

In general, the objective of an Extended Kalman Filter is to estimate the state of the robot. This estimation is performed through two main steps: a prediction step, in which the next state is predicted using the motion model, and an update step, in which the predicted state is corrected using the measurements obtained from the environment through the sensor model. Since the EKF operates within a Gaussian probability framework, the robot state is fully characterized by its mean μ and covariance Σ. In this setting, the filter maintains a unimodal Gaussian belief over the state: μ represents the nominal estimate of the state variables, while Σ represents the uncertainty and cross-correlations among them. This formulation is particularly convenient because it allows the system to model, predict, and correct the robot pose by updating only μ and Σ at each iteration. During the prediction stage, starting from a prior Gaussian $(\mu_{t-1}, \Sigma_{t-1})$and given the input velocities, the nonlinear motion model $g(\cdot)$produces the predicted mean μ_t^-. The covariance is propagated through linearization as

$$\Sigma_t^- = G_t\,\Sigma_{t-1}\,G_t^\top + V_t\,R_t\,V_t^\top,$$

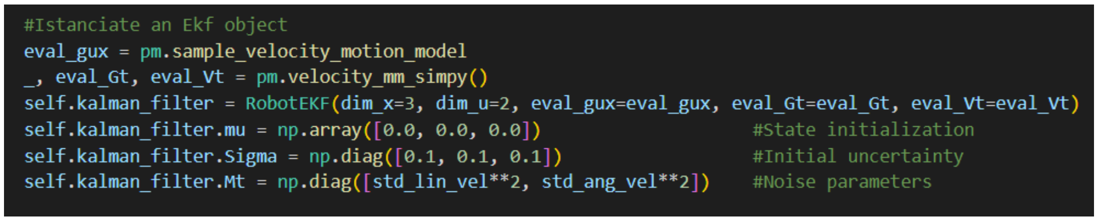
where $G_t = \frac{\partial g}{\partial x}$is the Jacobian of the motion model with respect to the state (capturing how small perturbations of the state affect the predicted dynamics), and $V_t = \frac{\partial g}{\partial \varepsilon}$(or, depending on the adopted formulation, with respect to the control/noise) maps process uncertainty into the state space. $R_t$is the covariance of the motion/process noise.

The implementation is carried out within the ROS 2 environment; therefore, the EKF must conform to its programming paradigms. A node object is first initialized, and within its constructor the required publishers and subscribers are created, together with the initialization of the EKF, as described below.

• Subscribers:
Two subscribers are created. The first subscribes to the /odom topic and extracts the robot's linear and angular velocities from the published Odometry messages; these values are used as input velocity command for the EKF prediction step. The second subscriber listens to the /landmarks topic and receives LandmarkArray messages, which contain the landmarks currently within the field of view of the robot's camera, together with their associated id and range and bearing measurements.

• Publisher:
Only one publisher is created, which is responsible for publishing the results of the overall EKF computation to a dedicated topic, in this case /ekf. The published message is of type Odometry

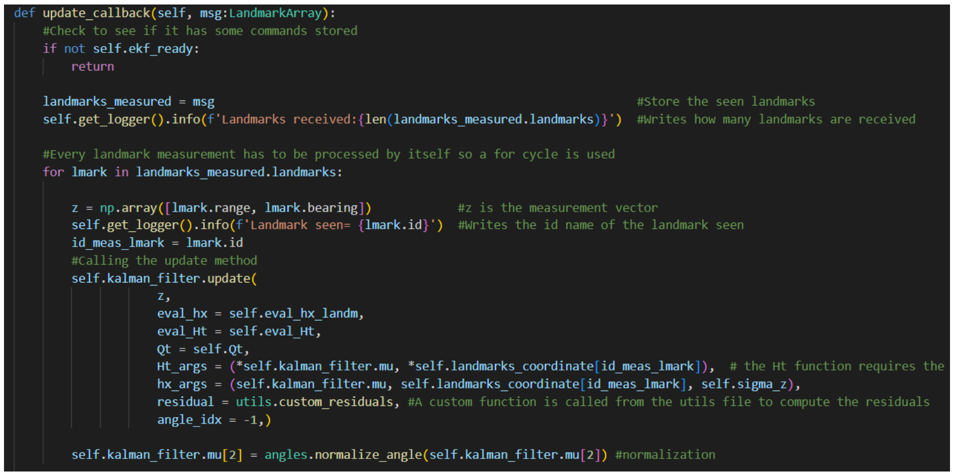
• EKF instantiation:
To enable the filter operation, an EKF object is instantiated within the node constructor. The object, which was given already prepared and available for use, requires as input the state and control dimensions, as well as the functions needed for the prediction step, namely the nonlinear motion model $g(\cdot)$and the corresponding linearization terms (Jacobians) required for the covariance propagation (e.g., $G_t$and V_t, together with the process noise model R_t). The EKF provides two predefined methods: one for the prediction step and one for the update step. These methods are general in nature and can be suitably tailored to the specific application.



```
#Istanciate an Ekf object
eval_gux = pm.sample_velocity_motion_model
_, eval_Gt, eval_Vt = pm.velocity_mm_simpy()
self.kalman_filter = RobotEKF(dim_x=3, dim_u=2, eval_gux=eval_gux, eval_Gt=eval_Gt, eval_Vt=eval_Vt)
self.kalman_filter.mu = np.array([0.0, 0.0, 0.0])                #State initialization
self.kalman_filter.Sigma = np.diag([0.1, 0.1, 0.1])              #Initial uncertainty
self.kalman_filter.Mt = np.diag([std_lin_vel**2, std_ang_vel**2])    #Noise parameters
```

Figure 6: EKF instantiation

The prediction method is invoked by a timer operating at a frequency of 20Hz. Its output consists of a predicted state mean and covariance. The update step is instead executed within the update_callback method, where the state update and the publication of the estimated state to the /ekf topic are performed. The update method is called inside a for loop that iterates over the set of currently observed landmarks (information taken by the subscriber in the /landmarks topic). The inputs to the update method include the landmark measurement model and its associated Jacobian, the measurement noise and parameters, and a residual function. In this phase, the correction is driven by the innovation term, i.e., the difference between the actual measurement and the expected measurement computed from the predicted state; this innovation is used to compute the Kalman gain and update the Gaussian belief, producing a corrected mean $\mu_t$and an updated covariance Σ_t, typically reducing uncertainty with respect to the prediction. The residual function is a custom function provided within the EKF node package and is responsible for both linear and nonlinear quantities, the latter corresponding in this case to the bearing measurement, ensuring consistent handling of angular residuals (wrapping/normalization).



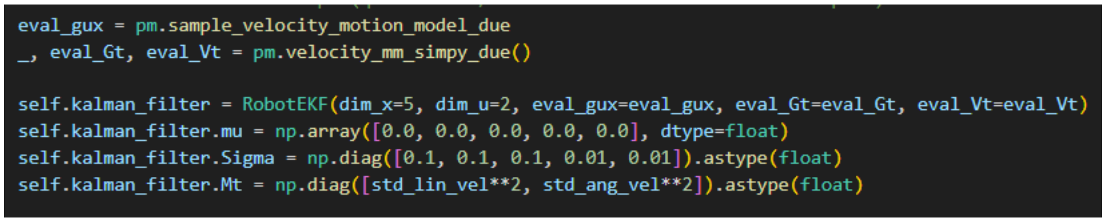
```python
def update_callback(self, msg:LandmarkArray):
    #Check to see if it has some commands stored
    if not self.ekf_ready:
        return

    landmarks_measured = msg                                                       #Store the seen landmarks
    self.get_logger().info(f'Landmarks received:{len(landmarks_measured.landmarks)}')  #Writes how many landmarks are received

    #Every landmark measurement has to be processed by itself so a for cycle is used
    for lmark in landmarks_measured.landmarks:

        z = np.array([lmark.range, lmark.bearing])            #z is the measurement vector
        self.get_logger().info(f'Landmark seen= {lmark.id}')  #Writes the id name of the landmark seen
        id_meas_lmark = lmark.id
        #Calling the update method
        self.kalman_filter.update(
                    z,
                    eval_hx = self.eval_hx_landm,
                    eval_Ht = self.eval_Ht,
                    Qt = self.Qt,
                    Ht_args = (*self.kalman_filter.mu, *self.landmarks_coordinate[id_meas_lmark]),  # the Ht function requires the
                    hx_args = (self.kalman_filter.mu, self.landmarks_coordinate[id_meas_lmark], self.sigma_z),
                    residual = utils.custom_residuals, #A custom function is called from the utils file to compute the residuals
                    angle_idx = -1,)

        self.kalman_filter.mu[2] = angles.normalize_angle(self.kalman_filter.mu[2]) #normalization
```

Figure 7: Update method inside the node

Task 2 implementation

In the second laboratory task, we were asked to extend the filter state vector by including the robot's linear velocity and angular velocity, in addition to its planar pose. Increasing the state dimension required a consistent revision of the entire Extended Kalman Filter (EKF) pipeline, affecting both the prediction and the update steps, in order to preserve dimensional consistency across vectors, matrices, and Jacobians.

Starting from the original 3-state filter $[x, y, \theta]$, we extended it to a 5-state formulation $[x, y, \theta, v, \omega]$. This required updating the initialization, motion model, linearization, and measurement models accordingly.



```python
eval_gux = pm.sample_velocity_motion_model_due
_, eval_Gt, eval_Vt = pm.velocity_mm_simpy_due()

self.kalman_filter = RobotEKF(dim_x=5, dim_u=2, eval_gux=eval_gux, eval_Gt=eval_Gt, eval_Vt=eval_Vt)
self.kalman_filter.mu = np.array([0.0, 0.0, 0.0, 0.0, 0.0], dtype=float)
self.kalman_filter.Sigma = np.diag([0.1, 0.1, 0.1, 0.01, 0.01]).astype(float)
self.kalman_filter.Mt = np.diag([std_lin_vel**2, std_ang_vel**2]).astype(float)
```

Figure 8: EKF instantiation

Beginning from the 3-state filter implementation, the initialization was updated by extending:

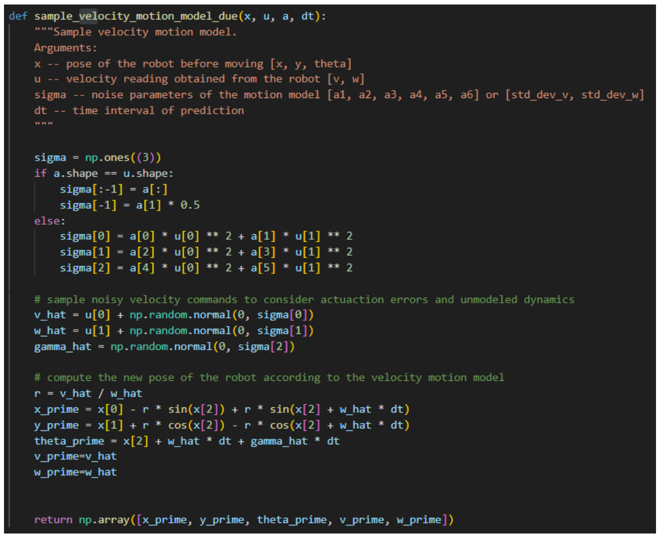
- the mean state vector μ, adding two initially null components corresponding to linear velocity v and angular velocity ω in kalman_filter.mu;
- the covariance matrix Σ, including the additional variances associated with the new states in kalman_filter.Sigma.

Specifically, while the first three pose-related states kept an initial variance of 0.1 (uncertainty on position and orientation), the two velocity states were initialized with a smaller variance of 0.01, since the modeled uncertainty concerns kinematic quantities (velocities) rather than position. This choice reflects a higher initial confidence in the velocity estimates, coherent with the nature of the commanded/measured inputs used by the model.

For the prediction step, it was necessary to update the terms $g(u_t, x_{t-1})$ (state transition model), G_t(Jacobian with respect to the state), and V_t(Jacobian with respect to control/noise), by introducing two new functions:

- sample_velocity_motion_model_due()
- velocity_mm_simpy_due()

Their purpose is to model the system consistently within a 5-dimensional state space, avoiding dimension mismatches in EKF operations (mean propagation and covariance propagation).



```python
def sample_velocity_motion_model_due(x, u, a, dt):
    """Sample velocity motion model.
    Arguments:
    x -- pose of the robot before moving [x, y, theta]
    u -- velocity reading obtained from the robot [v, w]
    sigma -- noise parameters of the motion model [a1, a2, a3, a4, a5, a6] or [std_dev_v, std_dev_w]
    dt -- time interval of prediction
    """

    sigma = np.ones((3))
    if a.shape == u.shape:
        sigma[:-1] = a[:]
        sigma[-1] = a[1] * 0.5
    else:
        sigma[0] = a[0] * u[0] ** 2 + a[1] * u[1] ** 2
        sigma[1] = a[2] * u[0] ** 2 + a[3] * u[1] ** 2
        sigma[2] = a[4] * u[0] ** 2 + a[5] * u[1] ** 2

    # sample noisy velocity commands to consider actuaction errors and unmodeled dynamics
    v_hat = u[0] + np.random.normal(0, sigma[0])
    w_hat = u[1] + np.random.normal(0, sigma[1])
    gamma_hat = np.random.normal(0, sigma[2])

    # compute the new pose of the robot according to the velocity motion model
    r = v_hat / w_hat
    x_prime = x[0] - r * sin(x[2]) + r * sin(x[2] + w_hat * dt)
    y_prime = x[1] + r * cos(x[2]) - r * cos(x[2] + w_hat * dt)
    theta_prime = x[2] + w_hat * dt + gamma_hat * dt
    v_prime=v_hat
    w_prime=w_hat


    return np.array([x_prime, y_prime, theta_prime, v_prime, w_prime])
```

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

Figure 9: five states Velocity Motion Model

1) sample_velocity_motion_model_due()

In this function, the extension required adding $v'$and ω'as new components of the output state vector. According to the provided model, the propagated velocities are defined as:

- output velocity = input velocity + zero-mean Gaussian noise
- standard deviation given by R_t

This function therefore supports mean prediction, since it estimates the a priori state evolution through nonlinear motion equations driven by inputs and affected by uncertainty.

2) velocity_mm_simpy_due()

In the second function, in addition to explicitly including the two states $v$and ωin the model equations, we recomputed:

- $G_t$as the Jacobian of the motion model with respect to all 5 states

This is essential to ensure that the covariance propagation equation:

$$\Sigma_t^- = G_t \Sigma_{t-1} G_t^\top + V_t R_t V_t^\top$$

remains dimensionally valid and physically meaningful in the extended state space. In other words, the linearization was updated to account for the effect of the velocity components on the overall dynamics (and vice versa), making the prediction step fully consistent with the new formulation.

For the update step, besides adapting the existing landmark-based update function from the 3-state filter, we introduced additional updates exploiting inertial and odometric data.

Numerical robustness of the motion model (handling $\omega \approx 0$). During the design phase, we noticed that in some segments where the angular velocity was zero, the filter tended to "blow up" in terms of approximation, showing a behavior typical of numerically unstable values (often related to divisions by zero or very small numbers). In particular, the motion model includes the term $\frac{v}{\omega}$, which, as $\omega \to 0$, can produce values close to ∞ and compromise the propagation of both the mean and the covariance.
To address this issue, before running the motion model we introduced an additional condition: if $|\omega| \leq 10^{-6}$, we approximate $\omega$to 10^{-6}. In this way, the ratio $\frac{v}{\omega}$remains numerically bounded, and the entire prediction phase preserves stability and consistency.

Landmark-based update (modified existing function)

The landmark update function was extended by including the velocity states as variables in the differentiation process, so that the resulting:

- $H_t$matrix is consistent with a 5-dimensional state vector,

which is required for a correct computation of the innovation covariance:

$$S_t = H_t \Sigma_t^- H_t^\top + Q_t$$

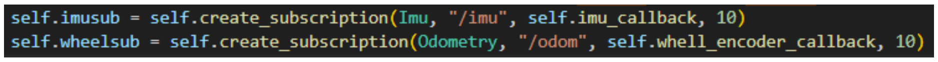
In practice, even if the landmark measurement does not explicitly depend on $v$and ω, the Jacobian must still have consistent dimensions (e.g., by adding additional zero columns) to correctly perform EKF matrix operations.

Two additional update functions were implemented, each linked to a specific sensor stream, by adding two subscribers:

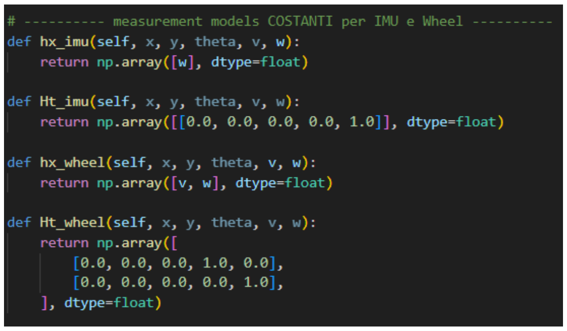
- **Subscriber on /imu**: performs an update specifically on angular velocity ω, correcting the state each time inertial data are received. This update leverages the angular rate measurement typically provided by the gyroscope.
- **Subscriber on /odom**: performs an update on both linear velocity v and angular velocity ωwhenever an odometry message is received (wheel encoder / odometric estimate). In this case, the measurement directly constrains both kinematic components of the state.

In both cases, the update was implemented following the standard EKF structure, defining the measurement model $h(x)$, the Jacobian H, and the measurement covariance Q, in order to fuse prediction and sensor observations in a statistically consistent manner.

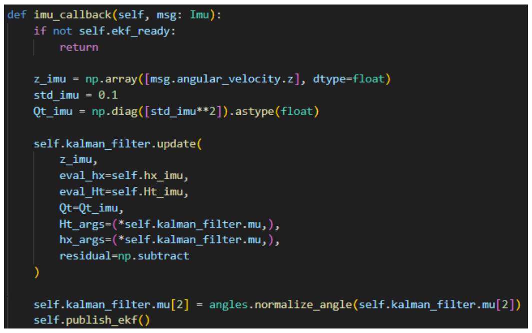
The corresponding functions are reported below (as required in the assignment).



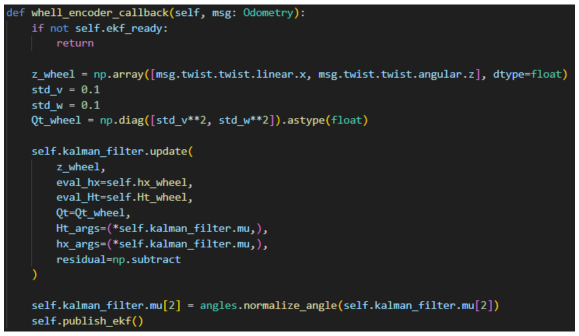
```python
self.imusub = self.create_subscription(Imu, "/imu", self.imu_callback, 10)
self.wheelsub = self.create_subscription(Odometry, "/odom", self.whell_encoder_callback, 10)
```



```python
# ---------- measurement models COSTANTI per IMU e Wheel ----------
def hx_imu(self, x, y, theta, v, w):
    return np.array([w], dtype=float)

def Ht_imu(self, x, y, theta, v, w):
    return np.array([[0.0, 0.0, 0.0, 0.0, 1.0]], dtype=float)

def hx_wheel(self, x, y, theta, v, w):
    return np.array([v, w], dtype=float)

def Ht_wheel(self, x, y, theta, v, w):
    return np.array([
        [0.0, 0.0, 0.0, 1.0, 0.0],
        [0.0, 0.0, 0.0, 0.0, 1.0],
    ], dtype=float)
```



```python
def imu_callback(self, msg: Imu):
    if not self.ekf_ready:
        return

    z_imu = np.array([msg.angular_velocity.z], dtype=float)
    std_imu = 0.1
    Qt_imu = np.diag([std_imu**2]).astype(float)

    self.kalman_filter.update(
        z_imu,
        eval_hx=self.hx_imu,
        eval_Ht=self.Ht_imu,
        Qt=Qt_imu,
        Ht_args=(*self.kalman_filter.mu,),
        hx_args=(*self.kalman_filter.mu,),
        residual=np.subtract
    )

    self.kalman_filter.mu[2] = angles.normalize_angle(self.kalman_filter.mu[2])
    self.publish_ekf()
```



```python
def whell_encoder_callback(self, msg: Odometry):
    if not self.ekf_ready:
        return

    z_wheel = np.array([msg.twist.twist.linear.x, msg.twist.twist.angular.z], dtype=float)
    std_v = 0.1
    std_w = 0.1
    Qt_wheel = np.diag([std_v**2, std_w**2]).astype(float)

    self.kalman_filter.update(
        z_wheel,
        eval_hx=self.hx_wheel,
        eval_Ht=self.Ht_wheel,
        Qt=Qt_wheel,
        Ht_args=(*self.kalman_filter.mu,),
        hx_args=(*self.kalman_filter.mu,),
        residual=np.subtract
    )

    self.kalman_filter.mu[2] = angles.normalize_angle(self.kalman_filter.mu[2])
    self.publish_ekf()
```

Figure 10: Update IMU and wheels encoder models.

4 Simulated and real robot results

The node is tested in a simulated environment using Gazebo. The results are recorded in a rosbag file and subsequently post-processed and plotted. The environment map is defined as a set of landmarks provided in a separate file. These landmarks are loaded through the load_landmarks function, implemented in the utils module, which reads the file and supplies the node with a dictionary structure. In this dictionary, each landmark id serves as a key, while the corresponding x and y coordinates represent the associated values.

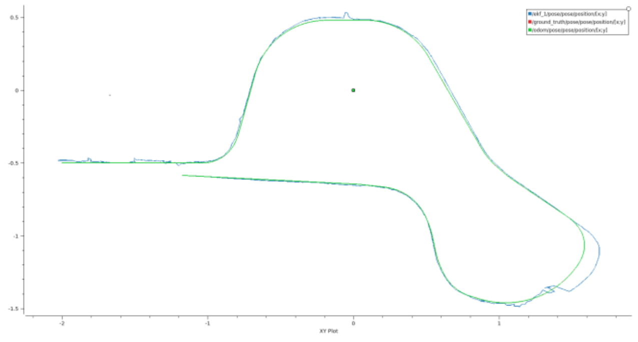


Figure 8: X-Y trajectory comparison

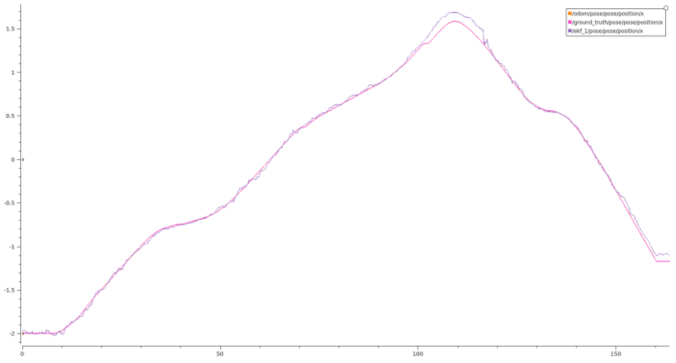


Figure 9: X-time trajectory comparison

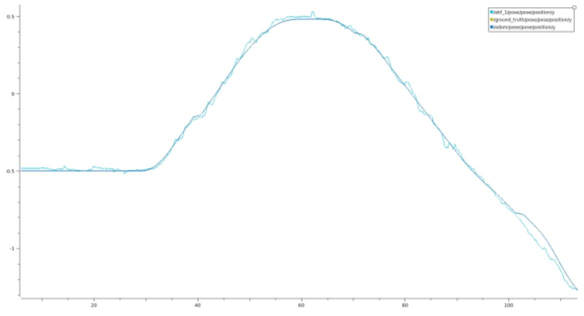


Figure 10: Y-time trajectory comparison

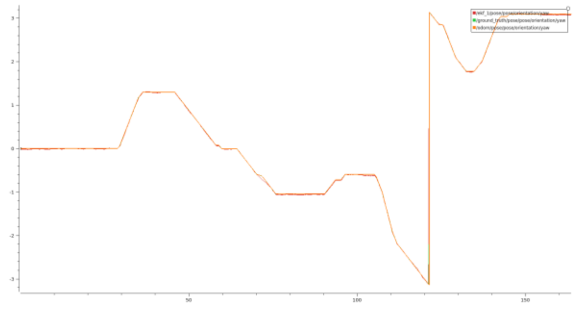
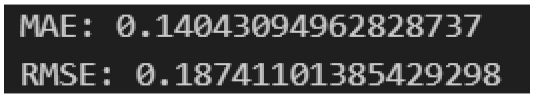


Figure 11: Yaw-time trajectory comparison

The results indicate that the Extended Kalman Filter (EKF) achieves satisfactory performance in the simulated environment. In particular, the estimated trajectory closely overlaps the ground-truth trajectory, demonstrating the filter's ability to reconstruct the robot state throughout its motion. This qualitative evidence is consistent with the computed quantitative metrics, namely the Root Mean Square Error (RMSE) and the Mean Absolute Error (MAE), reported in Figure 10. The good performance can also be attributed to the high availability of observable landmarks during navigation; conversely, with fewer reference features, an increase in estimation error would reasonably be expected.
The only significant deviation observable in the trajectory (for instance, in the x–y plot at the bottom right) is attributable to an intentional change in the simulation conditions. To evaluate the system's behavior in the absence of landmarks, the robot was manually repositioned in a region of the map devoid of reference

features. In this scenario, the lack of informative measurements reduces the effectiveness of the update step and leads to a progressive growth in the discrepancy between the true position and the filter estimate. Subsequently, once the robot was moved back into an area characterized by the presence of landmarks, the EKF was able to exploit the available observations to correct the state estimate through the update step, thereby reducing the error and realigning the estimated trajectory with the ground-truth one.



```
MAE: 0.14043094962828737
RMSE: 0.18741101385429298
```

Figure 12: RMSE and MAE computed from the simulated data (Task 1).

The real-world experimental data confirm this phenomenon. As observed in the y-coordinate and in the x-y trajectory, a slight divergence in the estimation error appears. This behavior is due to the robot entering a blind spot in which an insufficient number of landmarks were visible at that time.

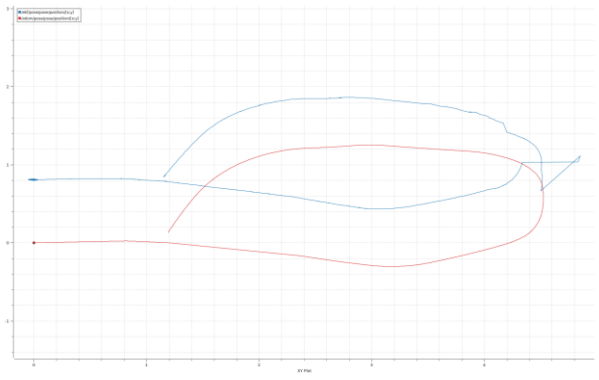


Figure 13: X-Y trajectory (real data)

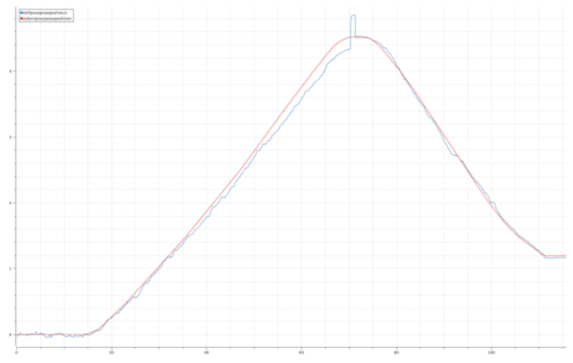


Figure 14: X-Time trajectory (real data)

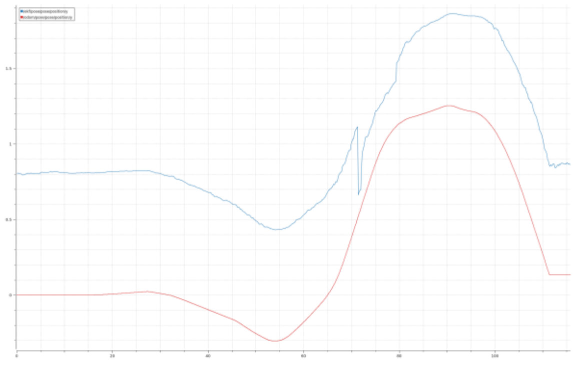


Figure 15: Y-time trajectory (real data)

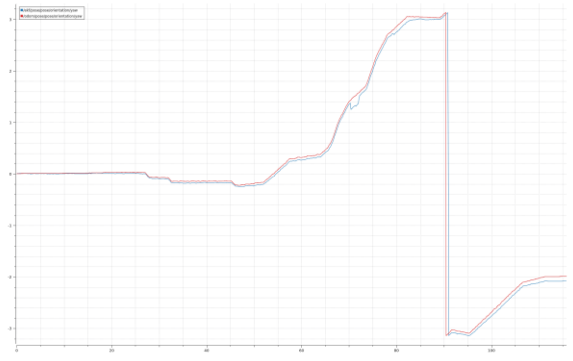


Figure 16: Yaw-time trajectory (real data)

The translation between the two plots comes from the fact that the origin of the two was shifted on the y axis. In all the plots, the red curve is the /odom data, the blue one is the /ekf data.

Task 2

The following plots compare the EKF estimates against ground truth and odometry. Since the experiment is performed in simulation, /odom and /ground_truth almost perfectly overlap; therefore, the main performance indicator is the deviation of /ekf from the reference trajectory.

x(t) and y(t) trends

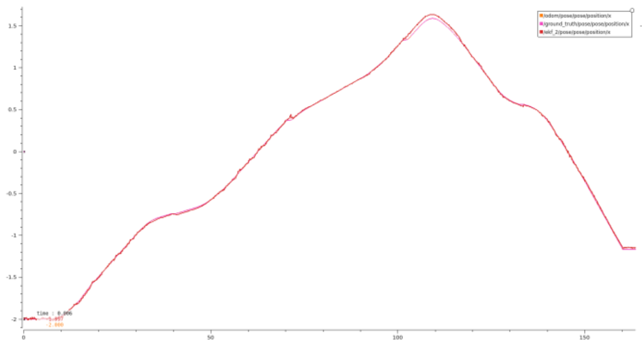


Figure 17: X-time trajectory comparison

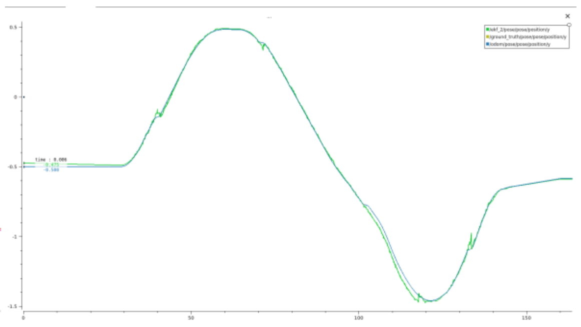


Figure 18: Y-time trajectory comparison

In the x-time and y-time plots, the EKF estimate remains well aligned with the reference along the entire trajectory. Any differences are small and localized, typically occurring in segments with higher curvature or during more abrupt changes in motion. This behaviour is consistent with the typical EKF trade-off between model-based prediction and measurement-based correction, where small transients can appear during dynamic variations.

ψ(t) (yaw) trend and discontinuities

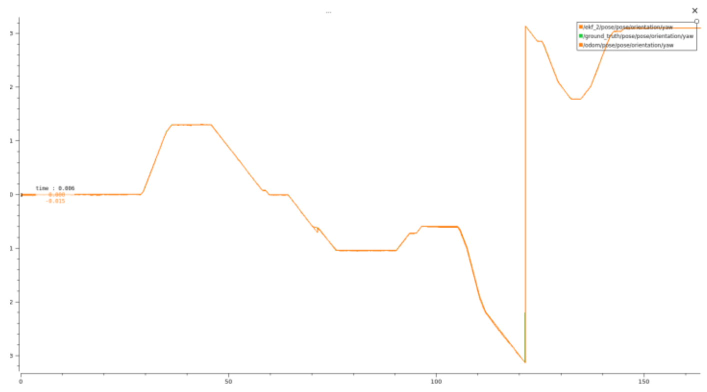


Figure 19: Yaw-time trajectory comparison

The yaw-time plot confirms good tracking of the robot heading. The "vertical jump" visible around $\psi \approx \pm\pi$ is not a physical discontinuity, but the standard angle wrapping effect (yaw representation in the $[-\pi, \pi]$ interval). When the heading crosses the boundary, the plotted value switches from $-\pi$ to $+\pi$ (or vice versa), producing an apparent instantaneous jump even though the orientation evolves continuously. Aside from this representation artifact, the EKF follows the reference heading with small deviations, which become more noticeable during fast rotations.

x–y trajectory comparison

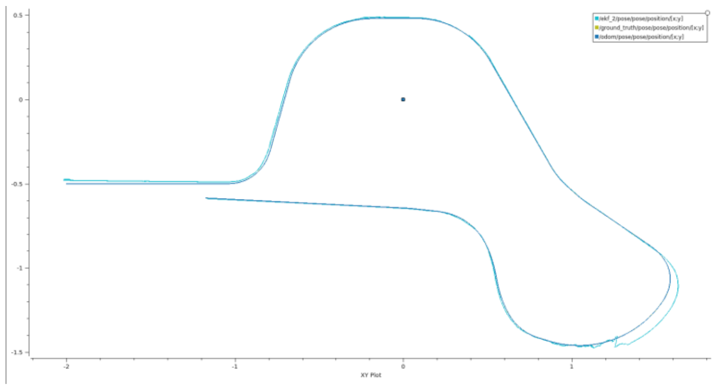


Figure 20: X-Y trajectory comparison

The planar trajectory plot shows a close overlap between the EKF estimate and the reference path for most of the run. The main deviation, especially visible along a curved segment, is due to the robot entering a map region where no landmarks are observable. In the absence of informative measurements, the update step becomes ineffective and the estimate is driven mainly by the prediction step (model/odometry), which leads to a progressive increase in pose error. Once the robot returns to an area with available landmarks, the EKF can exploit the observations to correct the state and realign the estimated trajectory with the reference, as discussed in the report.

Linear velocity $v_x$vs time

In the plot of the linear velocity along x, the EKF estimate appears strongly dominated by high-frequency components, effectively resembling Gaussian noise around its mean value. This behaviour is consistent with a scenario in which the simulated velocity dynamics (i.e., the actual command/motion variation) are comparable to, or smaller than, the amplitude of the process and/or measurement noise. Under these conditions, the signal-to-noise ratio is unfavourable and it becomes difficult to clearly appreciate the velocity trend, especially when $v_x$remains nearly constant over long intervals.

In other words, to make the noise negligible in the visualization (and thus highlight the filter tracking), the velocity variation imposed in simulation should be significantly larger than the overall standard deviation associated with v. In our case, since $v_x$is almost constant or only weakly varying, the plot mainly shows the noisy component.

Additionally, as discussed in the report, a direct comparison with the ground-truth velocity may be less immediate if it is expressed in a different reference frame (e.g., a fixed/world frame). For this reason, the report discusses the velocity using the norm of the velocity vector, which is more directly comparable to odometry.

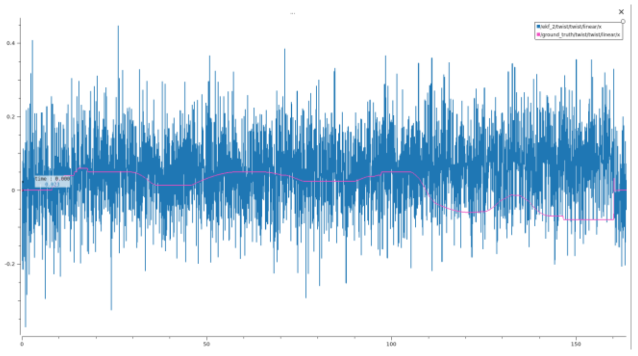


Figure 21: v_x-time plot

Angular velocity $\omega_z$vs time

In the angular velocity plot, the more pronounced changes (often step-like) make the filter behaviour easier to observe: the EKF is able to follow the $\omega_z$trend, producing an estimate that is overall consistent with the reference, with differences that become more visible during fast transitions. This matches the report's observation that the error tends to be larger during sharp direction changes, where the filter may exhibit a small delay or overshoot due to the combination of model assumptions, linearization, and the tuning of the covariance matrices Q/R.

16_Report01

In summary, the angular velocity contains more dynamic content than the noise (at least partially), making the EKF estimate more readable. To obtain a similarly clear behaviour for the linear velocity, Δvshould be much larger than the noise amplitude; in our results, this condition is only partially satisfied, mainly in the ω_z plot.

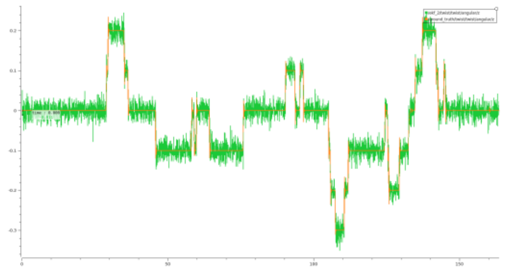
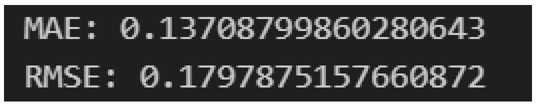


Figure 22: ω_z-time plot

Quantitative metrics (MAE / RMSE)

The computed metrics (MAE ≈ 0.137 and RMSE ≈ 0.180) confirm the qualitative evidence: the EKF estimate is on average close to the reference. The slightly higher RMSE compared to the MAE indicates the presence of a few larger transient deviations, typically associated with more complex trajectory segments and/or intervals with reduced measurement information (e.g., lack of visible landmarks).



```
MAE: 0.13708799860280643
RMSE: 0.1797875157660872
```

Figure 23: RMSE and MAE computed from the simulated data (Task 2).

Comparison

In the comparison between the Task 1 and Task 2 simulations, the plots show that in Task 1 the EKF trajectory generally overlaps well with the ground truth (with /odom and /ground_truth almost coincident), but small deviations appear mainly in higher-dynamics segments, while the presence of landmarks enables frequent updates that keep the pose error limited. In Task 2, for the same set of plots, the overlap with the reference is on average tighter and more stable, with further reduced deviations that are mostly confined to transients and more complex maneuvers, consistently with the increased robustness obtained by also including the velocity states. A localized deviation in the x-y plot can be attributed to the robot passing through a landmark-free region, where the update becomes poorly informative and the error grows until measurements become available again, allowing the filter to realign. Similarly, the x-time, y-time, and yaw-time plots highlight closer tracking in Task 2, while yaw jumps near $\pm\pi$ are due to angle wrapping. Overall, these qualitative observations are consistent with the metrics: in Task 1 a few larger transients affect the RMSE more than the MAE, whereas in Task 2 the better trajectory adherence and faster error convergence yield overall superior performance.

5 Conclusion

The results show that the EKF provides accurate trajectory reconstruction in simulation, with estimates generally overlapping the reference and deviations mainly associated with intervals of low measurement information (absence of landmarks) or more dynamic maneuvers. The state extension in Task 2 enables velocity estimation and overall leads to a more robust and stable filter, while also indicating that the readability/quality of velocity estimates depends on the signal-to-noise ratio (i.e., the motion variation relative to noise amplitude). Real-robot experiments confirm the same trends observed in simulation, with larger discrepancies when the robot traverses regions with poor landmark observability.