Electronic and Telecommunication Engineering
University of Moratuwa

EN4562: Autonomous Systems
Extended Kalman Filter Implementation

Madhushan R.M.S
200363R

1 Introduction

State estimation is a fundamental challenge in robotics, particularly for mobile robots operating in dynamic environments. The Kalman Filter (KF) is a widely used method for optimal state estimation in linear systems with Gaussian noise. However, its applicability is limited where it involves non-linear dynamics, as linear approximations often introduce significant errors.

To address this limitation, the Extended Kalman Filter (EKF) was introduced. The Extended Kalman Filter improve the capabilities of the standard Kalman Filter by linearizing the non-linear system dynamics around the best current estimate up to that particular time. This approach enables the EKF to estimate states in systems where the process and measurement models are non-linear.

This report focuses on the implementation of an Extended Kalman Fitler for an outdoor differential drive robot to estimate its position (x, y) and orientation θ. The Extended Kalman Filter utilizes odometry data from wheel encoders and position data from GPS to fuse information and produce a robust estimate of the robot's state. By fusing the encoder and the GPS data together we can minimize the limitations of each sensor such as errors in wheel encoders due to slippage, and noise errors in the GPS readings.

The following sections will discuss the theoretical foundation of the Extended Kalman Filter, the methodology employed for implementation, and an analysis of its performance. Key mathematical formulations, algorithmic steps, and results are presented to demonstrate the efficacy of the proposed approach in achieving accurate state estimation for outdoor differential drive robots.

2 Methodology

2.1 State Transition Model

The state transition model is used to predict the current state using the previous state and the current control input. Due to actuator uncertainties, this step incorporates a higher process noise covariance to account for potential deviations. This step provides an initial estimate of the robot's state before applying corrections based on sensor measurements.

The state estimation can be expressed as:

$$X_t = f(X_{t-1}, u_t) + \mathcal{N}(0, R),$$

where:

- X_t represents the predicted state vector $(x, y, \theta)^T$ in this case,
- $f(X_{t-1}, u_t)$ is the non-linear motion model that predicts the next state based on the previous state X_{t-1} and the control input u_t,
- $\mathcal{N}(0, R)$ represents Gaussian noise with zero mean and process noise covariance R.

2.2 Linearizing the State Transition Model

If the state transition model is nonlinear, it is necessary to linearize the system to use the Extended Kalman Filter (EKF). Linearization is performed using the first-order Taylor expansion. The linearized state transition function is given by:

$$f(X_{t-1}, u_t) \approx f(\mu_{t-1}, u_t) + G(\mu_{t-1}, u_t) \cdot (X_{t-1} - \mu_{t-1})$$

where μ_{t-1} is the previous state, which serves as the operating point for the Taylor series expansion.

To achieve better linearization, the best possible state estimation up to the current time should be used as the operating point. For time step t, the best estimate available is μ_{t-1}. Therefore, μ_{t-1} is chosen as the operating point of the Taylor series.

Here, G represents the Jacobian matrix of the state transition function, calculated at μ_{t-1}. The Jacobian matrix is defined as:

$$G = \left.\frac{\partial f}{\partial X}\right|_{\mu_{t-1}, u_t}$$

If the state transition function is given as:

$$f(X_{t-1}, u_t) = \begin{bmatrix} f_1 & f_2 & f_3 \end{bmatrix}^T$$

then the Jacobian matrix can be written in matrix form as:

$$G = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial \theta} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial \theta} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial \theta} \end{bmatrix}_{\mu_{t-1}, u_t}$$

This Jacobian matrix is evaluated at the operating point (μ_{t-1}, u_t). By incorporating this linearization, the EKF can approximate the state transition function in a form suitable for computation.

2.3 Prediction Step

In the prediction step of the Extended Kalman Filter (EKF), the following equations are used to compute the predicted state and its associated covariance:

$$\bar{\mu}_t = f(\mu_{t-1}, u_t)$$
$$\Sigma_t = G_t \Sigma_{t-1} G_t^\top + R$$

Here:

- $\bar{\mu}_t$ is the predicted state.
- $f(\mu_{t-1}, u_t)$ is the nonlinear state transition function, evaluated at the previous state μ_{t-1} and control input u_t.
- Σ_t is the predicted covariance matrix of the state.
- G_t is the Jacobian matrix of the state transition model, linearized at μ_{t-1}.
- Σ_{t-1} is the covariance matrix of the previous state.
- R is the process noise covariance matrix.

The EKF can be started with the initial state estimation and its covariance matrix. This step propagates the state estimate and its associated uncertainty forward in time, providing the possible prediction before incorporating any measurements.

2.4 Measurement Model

The measurement model in the Extended Kalman Filter (EKF) is used to relate the state variables to the structure of sensor measurements. This allows for the comparison between the actual sensor readings and the predicted sensor readings obtained from the model. The predicted sensor readings are the measurements expected from the sensor, based on the current state of the system.

If a measurement model is a non-linear system, then that model should be linearized using Taylor's expansion. In that case, we should choose the predicted state as the operating point of the Taylor's expansion since it is the best current estimation for the state in that particular time.

In this application, I have not chosen a non-linear measurement model. Therefore there is no need to linearize the measurement model.

The linear measurement model is defined as:

$$z_t = C_t X_t + \mathcal{N}(0, Q)$$

where:

- z_t is the predicted sensor measurement at time t,
- C_t is the observation matrix, which relates the state vector X_t to the measurement space,
- X_t is the state vector at time t,
- $\mathcal{N}(0, Q)$ represents Gaussian noise with zero mean and covariance Q, accounting for measurement noise,
- Q is the measurement noise covariance matrix, which models the uncertainty in the sensor readings.

The measurement model provides a way to compare the predicted sensor measurements from the model with the actual sensor readings, which is essential for updating the state estimate in the correction step of the EKF.

2.5 Correction Step

In this step, we incorporate measurements to improve the state estimation. The Extended Kalman Filter (EKF) provides an optimal method to correct the state estimate by incorporating information from multiple sensors. By fusing data from the sensors, we can achieve a more accurate and reliable estimate of the robot's state.

The correction step involves updating the predicted state using the measurement model and the measurement covariance matrices from both sensors. The Kalman gain, computed from the predicted state covariance and the measurement noise covariance, determines how much weight to give to each sensor's measurement.

To calculate the Kalman gain K_t, we use the following formula:

$$K_t = \bar{\Sigma}_t C_t^T \left(C_t \bar{\Sigma}_t C_t^T + R_t \right)^{-1}$$

where:

- $\bar{\Sigma}_t$ is the predicted state covariance matrix,
- C_t is the measurement matrix that relates the state to the measurement model,
- R_t is the measurement noise covariance matrix, and
- K_t is the Kalman gain that adjusts the correction step based on the measurement data.

Let $\mathbf{z}_t$ be the measurement vector, which consists of sensor measurements at time step t. The state estimate is corrected as follows:

$$\mu_t = \bar{\mu}_t + K_t(\mathbf{z}_t - \mathbf{h}(\bar{\mathbf{x}}_t))$$

where:

- $\mathbf{x}_t$ is the corrected state estimate after correction,
- $\mathbf{z}_t$ is the measurement vector from sensor,
- $\mathbf{h}(\bar{\mathbf{x}}_t)$ is the predicted measurement based on the previous state estimate,
- K_t is the Kalman gain, which balances the predicted state covariance and measurement noise covariance.

The corrected state estimate results in a more accurate representation of the robot's position and orientation, leveraging the complementary strengths of both the encoder and GPS sensors. This fusion of measurements improves the overall state estimation, especially in situations where one sensor might be more noisy or less accurate than the other.

The covariance matrix is also updated during the correction step to reflect the improved certainty in the state estimate:

$$\Sigma_t = (I - K_t C_t)\bar{\Sigma}_t$$

where:

- Σ_t is the updated state covariance matrix,
- $\bar{\Sigma}_t$ is the previous state covariance matrix,
- C_t is the observation matrix, and
- I is the identity matrix.

2.6 Sensor Fusion

In this section, we describe the process of sensor fusion, where we correct the predicted state first using the encoder measurements, and then further refine the state estimate by incorporating the GPS readings.

The sensor fusion procedure is performed in two stages:

1. **Correction using Encoder Measurements:** The first stage involves correcting the predicted state using the encoder measurements. The encoder provides information about the robot's motion and helps refine the position and orientation estimates. This correction step updates the state estimate and its associated covariance matrix based on the encoder data.
2. **Prediction using GPS Measurements:** After correcting the state using the encoder measurements, we proceed to the second stage. In this stage, we use the corrected state and covariance matrices to perform a new prediction step with the GPS measurements. The GPS provides accurate positional data and is used to further refine the state estimate. The prediction step takes into account the updated state from the encoder correction and the GPS sensor's measurements to provide a better estimate of the robot's position and orientation.

By first correcting the predicted state with the encoder and then using the GPS readings for further refinement, this approach takes advantage of the complementary strengths of both sensors. The encoder measurements help with motion tracking, while the GPS readings provide accurate positioning data, resulting in a more reliable and accurate state estimate after sensor fusion.

3 Implementation

In this section we will discuss about the implementation of the above methodology in differential drive robot state estimation application.

3.1 State Representation

For this project, a two-wheel differential drive robot is used. The states that we aim to predict are represented by the vector:

$$X_t = \begin{bmatrix} x & y & \theta \end{bmatrix}^T,$$

where x and y denote the robot's position in the global frame, and θ represents its orientation relative to the same frame. Starting position of the robot has been taken as the global frame in this implementation.

3.2 State Transition Model for a Differential Drive Robot

The state transition model for this implementation uses the control input vector $u_t = \begin{bmatrix} v & \omega \end{bmatrix}^T$, where:

- v is the linear velocity along the robot's local x-axis,
- ω is the angular velocity around the robot's local z-axis.

The non-linear state transition equations for a differential drive robot are expressed as:

$$X_t = \begin{cases} X_{t-1} + \begin{bmatrix} \frac{v}{\omega}\left(\sin(\theta_{t-1} + \omega\Delta t) - \sin(\theta_{t-1})\right) \\ \frac{v}{\omega}\left(-\cos(\theta_{t-1} + \omega\Delta t) + \cos(\theta_{t-1})\right) \\ \omega\Delta t \end{bmatrix}, & \text{if } \omega \neq 0, \\ \\ X_{t-1} + \begin{bmatrix} v\Delta t\cos(\theta_{t-1}) \\ v\Delta t\sin(\theta_{t-1}) \\ 0 \end{bmatrix}, & \text{if } \omega = 0. \end{cases}$$

Here:

- $X_t = \begin{bmatrix} x_t & y_t & \theta_t \end{bmatrix}^T$ is the predicted state vector at time t,
- $X_{t-1} = \begin{bmatrix} x_{t-1} & y_{t-1} & \theta_{t-1} \end{bmatrix}^T$ is the state vector at time $t-1$,
- v is the linear velocity,
- ω is the angular velocity,
- Δt is the time interval between predictions.

This model accounts for both rotational motion ($\omega \neq 0$) and straight-line motion ($\omega = 0$) scenarios, ensuring accurate predictions based on the robot's motion dynamics.

The following Python implementation corresponds to the above state transition model. It captures the logic for both cases, where $\omega \neq 0$ and $\omega = 0$:

```python
def state_transistion(self, x, u, dt):
    """
    x : Previous state (3x1) vector
    u : Control input (2x1) vector
    dt: Time interval
    """

    x_t_1, y_t_1, theta_t_1 = x[0], x[1], x[2]
    v_t, w_t = u[0], u[1]

    if abs(w_t) > 1e-6:
        x_t = x_t_1 + (v_t / w_t) * (np.sin(theta_t_1 + w_t * dt) - np.sin
            (theta_t_1))
        y_t = y_t_1 + (v_t / w_t) * (-np.cos(theta_t_1 + w_t * dt) + np.
            cos(theta_t_1))
        theta_t = theta_t_1 + w_t * dt
    else:
        x_t = x_t_1 + v_t * dt * np.cos(theta_t_1)
        y_t = y_t_1 + v_t * dt * np.sin(theta_t_1)
        theta_t = theta_t_1

    return np.array([x_t, y_t, theta_t])
```

Listing 1: State Transition Function for a Differential Drive Robot

3.3 Linearizing the State Transition Model for the Differential Drive Robot

The proposed state transition model is nonlinear and thus needs to be linearized using the first-order Taylor expansion at μ_{t-1}. The general format of the state transition model is given as:

$$f(X_{t-1}, u_t) = \begin{bmatrix} f_1 & f_2 & f_3 \end{bmatrix}^T$$

where the individual components of f are defined as:

$$f_1 = x_{t-1} + \frac{v_t}{\omega_t} \left(\sin(\theta_{t-1} + \omega_t \Delta t) - \sin(\theta_{t-1})\right)$$

$$f_2 = y_{t-1} - \frac{v_t}{\omega_t} \left(\cos(\theta_{t-1} + \omega_t \Delta t) - \cos(\theta_{t-1})\right)$$

$$f_3 = \theta_{t-1} + \omega_t \Delta t$$

To linearize the state transition model, it should calculate the Jacobian matrix of f with respect to X_{t-1} at the operating point (μ_{t-1}, u_t). The general form of the Jacobian is:

$$G = \left.\frac{\partial f}{\partial X}\right|_{\mu_{t-1},u_t} = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial \theta} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial \theta} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial \theta} \end{bmatrix}_{\mu_{t-1},u_t}$$

According to the proposed state transition model, the Jacobian matrix is as follows:

$$G = \begin{cases} \begin{bmatrix} 0 & 0 & \frac{v_t}{\omega_t}\left(\cos(\theta_{t-1}+\omega_t\Delta t) - \cos(\theta_{t-1})\right) \\ 0 & 0 & \frac{v_t}{\omega_t}\left(\sin(\theta_{t-1}+\omega_t\Delta t) - \sin(\theta_{t-1})\right) \\ 0 & 0 & 0 \end{bmatrix}, & \text{if } \omega_t \neq 0, \\ \begin{bmatrix} 0 & 0 & -v_t\sin(\theta_{t-1})\Delta t \\ 0 & 0 & v_t\cos(\theta_{t-1})\Delta t \\ 0 & 0 & 0 \end{bmatrix}, & \text{if } \omega_t = 0. \end{cases}$$

The following Python implementation computes the Jacobian matrix for the given state transition model:

```python
def jacobian_F(self, x, u, dt):
    """
    x : Previous state (3x1) vector
    u : Control input  (2x1) vector
    dt: Time interval
    """

    x_t_1 , y_t_1, theta_t_1 = x[0], x[1], x[2]
    v_t , w_t = u[0], u[1]
    F = np.zeros((3,3))

    if abs(w_t) > 1e-6:
        F[0, 2] = (v_t / w_t) * (np.cos(theta_t_1 + w_t * dt) - np.cos(
            theta_t_1))
        F[1, 2] = (v_t / w_t) * (np.sin(theta_t_1 + w_t * dt) - np.sin(
            theta_t_1))
    else:
        F[0, 2] = -v_t * np.sin(theta_t_1) * dt
        F[1, 2] =  v_t * np.cos(theta_t_1) * dt

    return F
```

Listing 2: Jacobian Matrix Calculation for the State Transition Model

This implementation correctly handles both rotational motion ($\omega_t \neq 0$) and straight-line motion ($\omega_t = 0$) scenarios, ensuring accurate computation of the Jacobian matrix.

3.4 Process Noise Covariance

For the process noise covariance, the linear velocity v and angular velocity ω are assumed to have noise associated with them. The process noise covariance is computed using the standard deviations of the linear and angular velocities. The noise values are:

$$v_{\text{noise\_std}} = 0.01, \quad w_{\text{noise\_std}} = 0.01$$

The process noise covariance matrix R is given by:

$$R = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\theta^2 \end{bmatrix}$$

where the individual variances are calculated as:

$$\sigma_x^2 = (v_{\text{noise\_std}})^2, \quad \sigma_y^2 = (v_{\text{noise\_std}})^2, \quad \sigma_\theta^2 = (w_{\text{noise\_std}})^2$$

The process noise covariance matrix R accounts for the uncertainty in the control inputs and helps to refine the estimation of the state during the prediction step of the EKF.

3.5 Initial State Estimation

In this EKF implementation, the initial state is assumed to be at the origin with no rotation, i.e., $(x_0, y_0, \theta_0) = (0, 0, 0)$. The uncertainty in the initial state is represented by the covariance matrix P_0, which is initialized as a small value to reflect the high confidence in the initial state. Therefore, I have used the following initial covariance matrix:

$$\Sigma_0 = \begin{bmatrix} 1 \times 10^{-9} & 0 & 0 \\ 0 & 1 \times 10^{-9} & 0 \\ 0 & 0 & 1 \times 10^{-9} \end{bmatrix}$$

The small values along the diagonal of this matrix represent small uncertainties in the initial state estimate, specifically for the x-position, y-position, and θ value, which are reasonable since it is assumed that the robot starts at $(0, 0, 0)$.

3.6 Encoder Measurement Model for Differential Drive Robot

In this application, the encoder readings are used as sensor measurements. The encoders provide the amount of wheel rotation, which is then converted into the state variables (x, y, θ) in order to align with the state vector used in the Extended Kalman Filter (EKF).

The conversion from the encoder readings to (x, y, θ) is done as follows:

$$\Delta x = \frac{(r \cdot \Delta\text{left} + r \cdot \Delta\text{right})}{2}$$

$$\Delta\theta = \frac{r \cdot (\Delta\text{right} - \Delta\text{left})}{d}$$

$$\theta_t = \theta_{t-1} + \Delta\theta$$
$$x_t = x_{t-1} + \Delta x \cdot \cos(\theta_t)$$
$$y_t = y_{t-1} + \Delta x \cdot \sin(\theta_t)$$

Where:

- r is the wheel radius,
- d is the wheel separation (distance between the left and right wheels),
- Δleft and Δright are the changes in the rotation angle for the left and right wheels respectively,
- θ_t is the orientation angle at time t,
- x_t and y_t are the position coordinates at time t.

The corresponding Python implementation for this model is given below:

```python
wheel_encoder_left  = msg.position[1]  # left motor angle
wheel_encoder_right = msg.position[0]  # right motor angle

delta_left  = wheel_encoder_left  - self.left_motor_pos_init
delta_right = wheel_encoder_right - self.right_motor_pos_init

linear_delta_pos    = (self.wheel_radius * delta_left + self.wheel_radius
    * delta_right) / 2
angular_delta_theta = (self.wheel_radius * delta_right - self.wheel_radius
     * delta_left) / self.wheel_separation
self.theta          += angular_delta_theta
self.x              += linear_delta_pos * np.cos(self.theta)
self.y              += linear_delta_pos * np.sin(self.theta)

self.left_motor_pos_init  = msg.position[1]
self.right_motor_pos_init = msg.position[0]
```

Listing 3: Encoder reading conversion to state variable

Even though the encoders provide wheel rotation data, they are used to directly estimate the robot's state (x, y, θ), which aligns with the state vector of the EKF.

Therefore, the measurement model for the encoder readings is assumed to be linear, as follows:

$$\begin{bmatrix} x_{\text{meas}} \\ y_{\text{meas}} \\ \theta_{\text{meas}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \mathcal{N}(0, Q)$$

where $\begin{bmatrix} x_{\text{meas}} & y_{\text{meas}} & \theta_{\text{meas}} \end{bmatrix}^T$ represents the predicted encoder measurements, and the identity matrix C relates the state vector $\begin{bmatrix} x_t & y_t & \theta_t \end{bmatrix}^T$ to the measurements.

The measurement noise covariance Q is defined as:

$$Q = \begin{bmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_\theta \end{bmatrix}$$

where:

- $\sigma_x = x_{\text{std}}^2$,
- $\sigma_y = y_{\text{std}}^2$,
- $\sigma_\theta = \theta_{\text{std}}^2$,
- $x_{\text{std}} = 0.003$, $y_{\text{std}} = 0.003$, and $\theta_{\text{std}} = 0.002$ are the standard deviations for the encoder measurements.

This measurement model assumes that the encoder measurements are subject to Gaussian noise, with the measurement covariance matrix Q representing the uncertainty in the encoder's output. The model allows the EKF to compare the predicted state from the robot's kinematics with the actual encoder readings, enabling an update to the state estimate.

3.7 GPS Measurement Model for Differential Drive Robot

In this application, GPS readings provide the latitude and longitude of the robot's position. However, since the robot's state vector is defined in terms of the x and y coordinates, it will be simpler to deal with if we convert the GPS data from latitude and longitude to the robot's local coordinate system.

The conversion from latitude and longitude to local x and y coordinates is done by using the Python `pyproj` library. Although the GPS provides latitude and longitude measurements, they are converted to x and y coordinates in the robot's local frame for simplicity. This makes the measurement model and the corresponding measurement matrix much easier to work with in the context of the EKF, as the robot's state vector is represented in terms of (x, y, θ) coordinates.

The GPS measurement model is formulated as follows:

$$\begin{bmatrix} x_{\text{meas}} \\ y_{\text{meas}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \mathcal{N}(0, Q)$$

where $\begin{bmatrix} x_{\text{meas}} & y_{\text{meas}} \end{bmatrix}^T$ represents the predicted GPS measurements, and the matrix C relates the state vector $\begin{bmatrix} x_t & y_t & \theta_t \end{bmatrix}^T$ to the measurements.

This measurement model assumes that the GPS only measures the x and y positions of the robot. The third dimension (θ) is not measured by the GPS, so it is excluded from the measurement matrix. The measurement covariance Q accounts for the noise in the GPS data, which is assumed to be Gaussian.

To align GPS measurements with the robot's local coordinate system, the origin of the local frame is set based on the robot's initial GPS reading. The following Python implementation demonstrates this process:

```python
if not self.is_origin:
    self.origin_x, self.origin_y = self.transformer.transform(msg.
        longitude, msg.latitude)
    self.is_origin = True

current_x, current_y = self.transformer.transform(msg.longitude, msg.
    latitude)
current_x = self.origin_x - current_x
current_y = self.origin_y - current_y
```

Listing 4: GPS reading conversion to local coordinates

The measurement noise covariance for GPS Q is defined as:

$$Q = \begin{bmatrix} \sigma_x & 0 \\ 0 & \sigma_y \end{bmatrix}$$

where:

- $\sigma_x = x_{\text{std}}^2$,
- $\sigma_y = y_{\text{std}}^2$,
- $x_{\text{std}} = 0.001$, $y_{\text{std}} = 0.001$ are the standard deviations for the GPS position measurements.

By converting the GPS data into local coordinates and simplifying the measurement matrix, the EKF can effectively integrate the GPS measurements into the state estimation process.

3.8 Correction with Encoders

In this step, we perform the correction of the predicted state using the encoder measurements. The process involves calculating the Kalman gain for the encoder measurements, K_{encoder}, and then using this gain to update the state estimate and covariance.

The Kalman gain for the encoder, denoted K_{encoder}, is given by:

$$K_{\text{encoder}} = \bar{\Sigma}_t C_{\text{encoder}}^T \left(C_{\text{encoder}} \bar{\Sigma}_t C_{\text{encoder}}^T + R_{\text{encoder}} \right)^{-1}$$

The Kalman gain for the encoder measurements, K_{encoder}, is computed based on the predicted covariance matrix $\bar{\Sigma}_t$, the encoder measurement model matrix C_{encoder}, and the measurement noise covariance matrix R_{encoder}. The measurement noise covariance R_{encoder} accounts for the noise in the encoder readings and is defined as:

$$Q = \begin{bmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_\theta \end{bmatrix}$$

where:

- $\sigma_x = (x_{\text{std}})^2$, with $x_{\text{std}} = 0.003$,
- $\sigma_y = (y_{\text{std}})^2$, with $y_{\text{std}} = 0.003$,
- $\sigma_\theta = (\theta_{\text{std}})^2$, with $\theta_{\text{std}} = 0.002$.

The following Python function implements the computation of K_{encoder}:

```python
def encoder_kalman_gain(self, cov_t_, C):
    """
    cov_t_ : Covariance matrix of prediction state
    C      : Measurement model matrix of Encoders
    """

    x_std     = 0.003
    y_std     = 0.003
    theta_std = 0.002

    sigma_x     = x_std ** 2
    sigma_y     = y_std ** 2
    sigma_theta = theta_std ** 2

    # Measurement noise covariance
    Q = np.diag([sigma_x, sigma_y, sigma_theta])

    innovation = C @ cov_t_ @ C.T + Q
    K = cov_t_ @ C.T @ np.linalg.inv(innovation)

    return K
```

Listing 5: Encoder Kalman Gain Calculation

This function performs the following steps:

- Defines the measurement noise standard deviations $(x_{\text{std}}, y_{\text{std}}, \theta_{\text{std}})$.
- Constructs the measurement noise covariance matrix Q using these standard deviations.
- Computes the innovation matrix as $C_{\text{encoder}}\bar{\Sigma}_t C_{\text{encoder}}^T + Q$, where C_{encoder} is the measurement model matrix.
- Calculates the Kalman gain K_{encoder} using the innovation matrix.

The Kalman gain K_{encoder} enables the Extended Kalman Filter (EKF) to optimally combine the encoder measurements with the predicted state, resulting in an improved state estimate and reduced uncertainty.

The encoder correction step is then performed by updating the state estimate $\bar{\boldsymbol{\mu}}_{\boldsymbol{t}}$ using the Kalman gain K_{encoder} and the residual (the difference between the actual encoder measurement and the predicted encoder measurement):

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_{\boldsymbol{t}} + K_{\text{encoder}} \left(\mathbf{z}_{\text{encoder}} - h(\bar{\boldsymbol{\mu}}_{\boldsymbol{t}})\right)$$

Finally, the state covariance matrix is updated to reflect the new uncertainty in the state estimate after incorporating the encoder data:

$$\Sigma_t = \left(I - K_{\text{encoder}} C_{\text{encoder}}\right) \bar{\Sigma_t}$$

where all symbols represent their usual meaning.

3.9 Correction with GPS

In this step, we perform the correction of the predicted state using the GPS measurements. Similar to the encoder correction, the process involves calculating the Kalman gain for the GPS measurements, K_{GPS}, and then using this gain to update the state estimate and covariance.

The Kalman gain for the GPS, denoted K_{GPS}, is given by:

$$K_{\text{GPS}} = \bar{\Sigma_t} C_{\text{GPS}}^T \left(C_{\text{GPS}} \bar{\Sigma_t} C_{\text{GPS}}^T + R_{\text{GPS}}\right)^{-1}$$

The Kalman gain for the GPS measurements, K_{GPS}, is calculated based on the predicted covariance matrix $\bar{\Sigma_t}$, the GPS measurement model matrix C_{GPS}, and the measurement noise covariance matrix R_{GPS}. The measurement noise covariance R_{GPS} accounts for the noise in the GPS readings and is defined as:

$$Q = \begin{bmatrix} \sigma_x & 0 \\ 0 & \sigma_y \end{bmatrix}$$

where:

- $\sigma_x = (x_{\text{std}})^2$, with $x_{\text{std}} = 0.001$,
- $\sigma_y = (y_{\text{std}})^2$, with $y_{\text{std}} = 0.001$.

The following Python function implements the computation of K_{GPS}:

```python
def GPS_kalman_gain(self, cov_t_, C):
    """
    cov_t_ : Covariance matrix of prediction state
    C      : Measurement model matrix of GPS
    """

    x_std = 0.001
    y_std = 0.001

    sigma_x = x_std ** 2
    sigma_y = y_std ** 2

    # Measurement noise covariance
    Q = np.diag([sigma_x, sigma_y])

    innovation = C @ cov_t_ @ C.T + Q
    K = cov_t_ @ C.T @ np.linalg.inv(innovation)

    return K
```

Listing 6: GPS Kalman Gain Calculation

This function performs the following steps:

- Defines the measurement noise standard deviations $(x_{\text{std}}, y_{\text{std}})$.
- Constructs the measurement noise covariance matrix Q using these standard deviations.
- Computes the innovation matrix as $C_{\text{GPS}}\bar{\Sigma}_t C_{\text{GPS}}^T + Q$, where C_{GPS} is the GPS measurement model matrix.
- Calculates the Kalman gain K_{GPS} using the innovation matrix.

The Kalman gain K_{GPS} allows the Extended Kalman Filter (EKF) to optimally integrate the GPS measurements with the predicted state, improving the state estimate and reducing uncertainty.

The GPS correction step is then performed by updating the state estimate, which has already been corrected by the encoder measurements in the previous step $\boldsymbol{\mu}_t$, using the Kalman gain K_{GPS} and the residual (the difference between the actual GPS measurement and the predicted GPS measurement):

$$\boldsymbol{\mu}_t = \boldsymbol{\mu_t} + K_{\text{GPS}} \left(\mathbf{z}_{\text{GPS}} - h(\boldsymbol{\mu_t}) \right)$$

Finally, the state covariance matrix is updated to reflect the new uncertainty in the state estimate after incorporating the GPS data:

$$\Sigma_t = \left(I - K_{\text{GPS}} C_{\text{GPS}} \right) \bar{\Sigma}_t$$

where:

- $\bar{\Sigma}_t$ is the predicted covariance matrix,
- C_{GPS} is the measurement model matrix for GPS,
- R_{GPS} is the measurement noise covariance matrix for GPS,

- $\mathbf{z}_{\text{GPS}}$ is the actual GPS measurement,
- $h(\boldsymbol{\mu}_t)$ is the predicted GPS measurement based on the state vector.

The following Python code snippet demonstrates how this correction step can be implemented. The arguments will vary depending on the sensor being used (e.g., K_{GPS} for GPS, or other gain matrices for other sensors):

```python
def correction(self, x_t_, z, C, cov_t_, K):
    """
    x_t_   : Predicted state (3x1) vector
    z      : Sensor reading
    C      : Measurement model matrix
    cov_t_ : Covariance matrix from prediction step
    K      : Kalman gain
    """
    h = self.measurment_model(x_t_, C)
    I = np.eye(cov_t_.shape[0])

    x_t = np.asarray(x_t_ + K @ (z - h))
    cov_t = np.asarray((I - K @ C) @ cov_t_)

    return x_t, cov_t
```

Listing 7: Correction Step Implementation

Here:

- `x_t_` is the predicted state vector.
- `z` is the sensor measurement vector .
- `C` is the measurement model matrix for the specific sensor.
- `cov_t_` is the covariance matrix from the prediction step.
- `K` is the Kalman gain for specific sensor.
- `h` is the predicted sensor measurement using the measurement model.

This modular implementation allows for the correction step to be easily adapted to different sensors, including GPS or encoders, by appropriately defining the measurement model C and the Kalman gain K.

4 Results

In this section, the results obtained from the sensor fusion process are presented. The primary focus is on evaluating the performance of the state estimation algorithms, specifically using encoder and GPS data. Visualizations of the robot's trajectory, alongside error analysis, will demonstrate how sensor fusion improves the accuracy of the predicted state compared to the individual sensor measurements. Various trajectory plots are included to show the effectiveness of the fusion process in correcting the robot's position and orientation over time.

The following color coding is used in the trajectory visualizations:

- Red: Predicted state trajectory (based on the initial prediction before correction).
- Green: Corrected trajectory using only encoder measurements.
- Yellow: Corrected trajectory after fusing both encoder and GPS measurements.
- Blue: True state of the robot (ground truth).

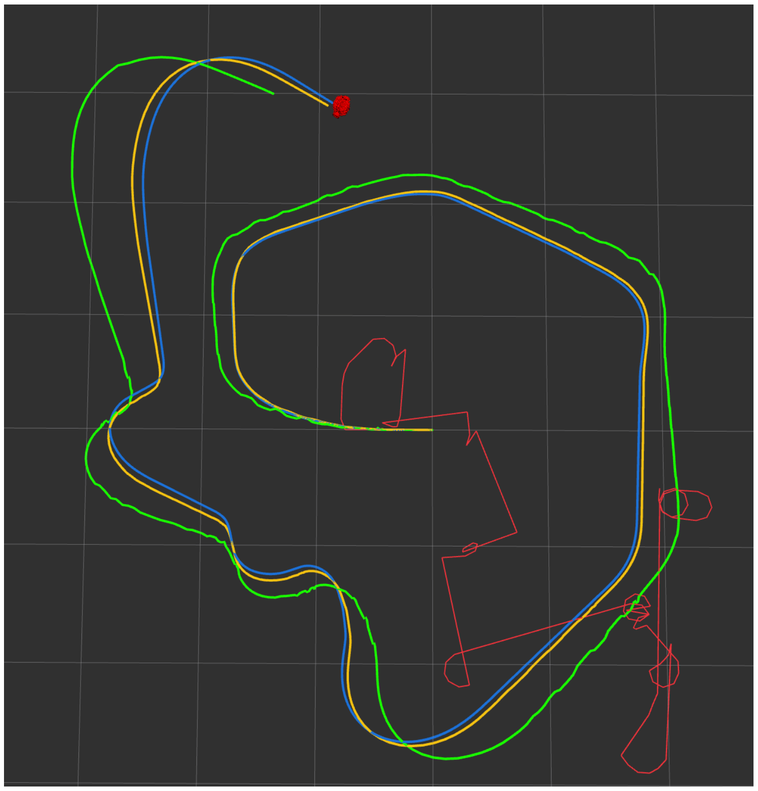

Figure 1: Trajectory Representation.

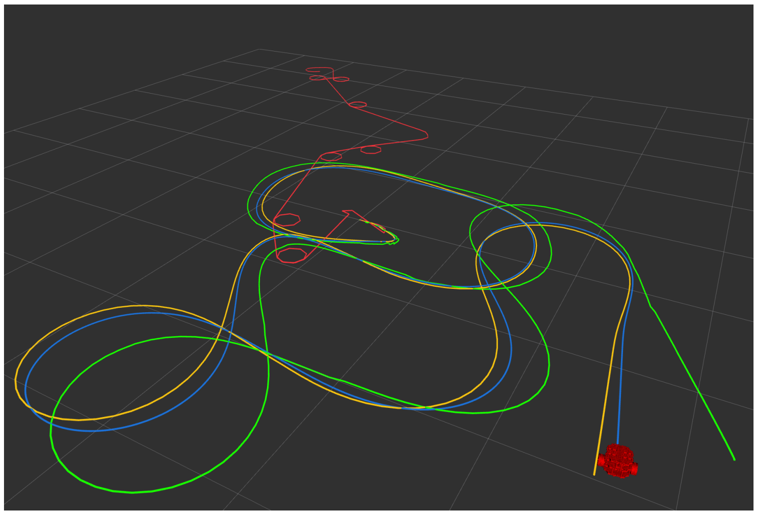

Figure 2: Trajectory Representation.

The trajectories for each state estimation method are presented with different colors to illustrate the improvements at each stage.

- **Blue:** The blue trajectory represents the ground truth (true state) of the robot.
- **Red:** The red trajectory represents the state estimation using only the state transition model. As seen in the results, this trajectory deviates significantly from the ground truth.
- **Green:** The green trajectory shows the state estimation after using only the encoder measurements for correction. As shown in the results, the state estimation has been improved significantly compared to the predicted state. However, there is still some notable inaccuracy as the trajectory length increases.
- **Yellow:** The yellow trajectory represents the state estimation after fusing the encoder and GPS measurements. As demonstrated in the results, this fusion results in a much more accurate state estimation compared to both the predicted state and the corrected state using only encoders.

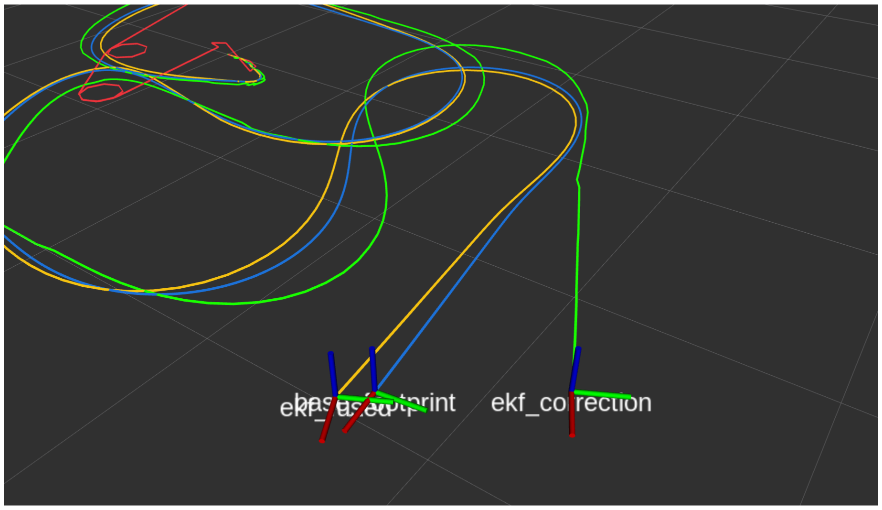

Figure 3: $\theta estimation Representation.$

The above result shows the estimation of the θ value. Since there is no measurement reading from the GPS, the θ value will not be corrected by the GPS, but only by the encoders where each links represent is correspond to the color of the trajectory as mentioned above.

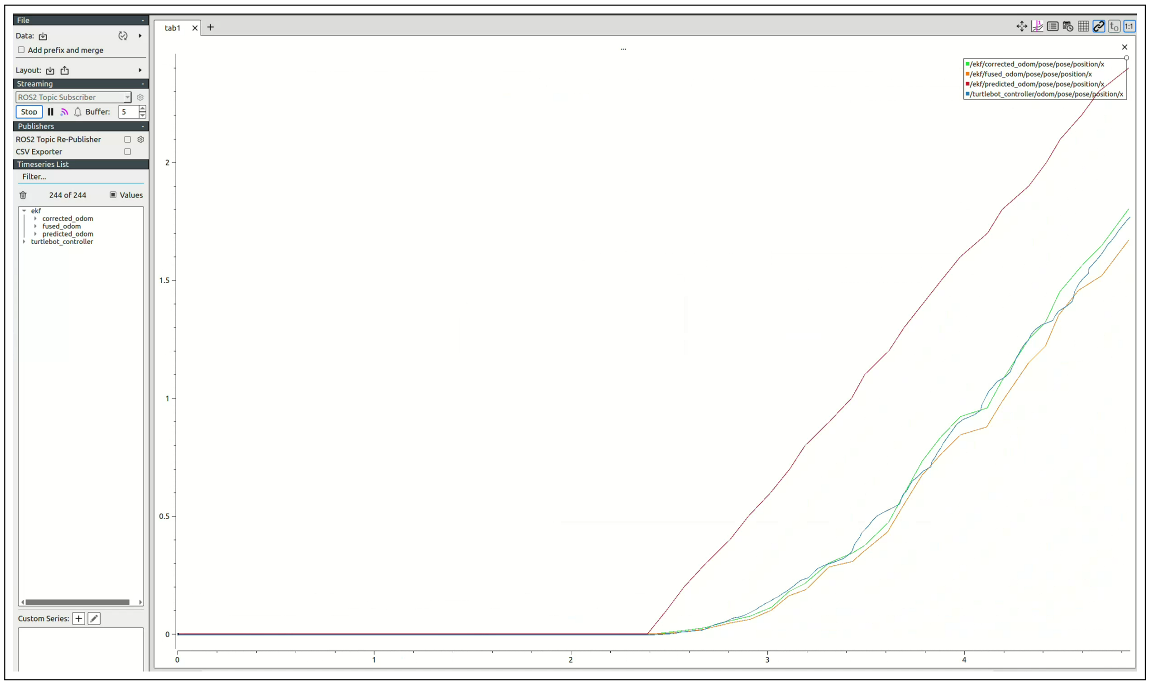

Figure 4: Variation of X coordinate

Each line in the plot corresponds to the previously mentioned color, with the same meaning assigned to each trajectory.

As shown in the above plot, when transitioning from rest to velocity, the state transition model responds quickly. This is because the model does not account for any mass or physical properties of the robot.

Additionally, as shown in the plot, both the state correction using encoders and the state correction by fusing the encoder and GPS measurements yield similar values at the beginning of the trajectory.

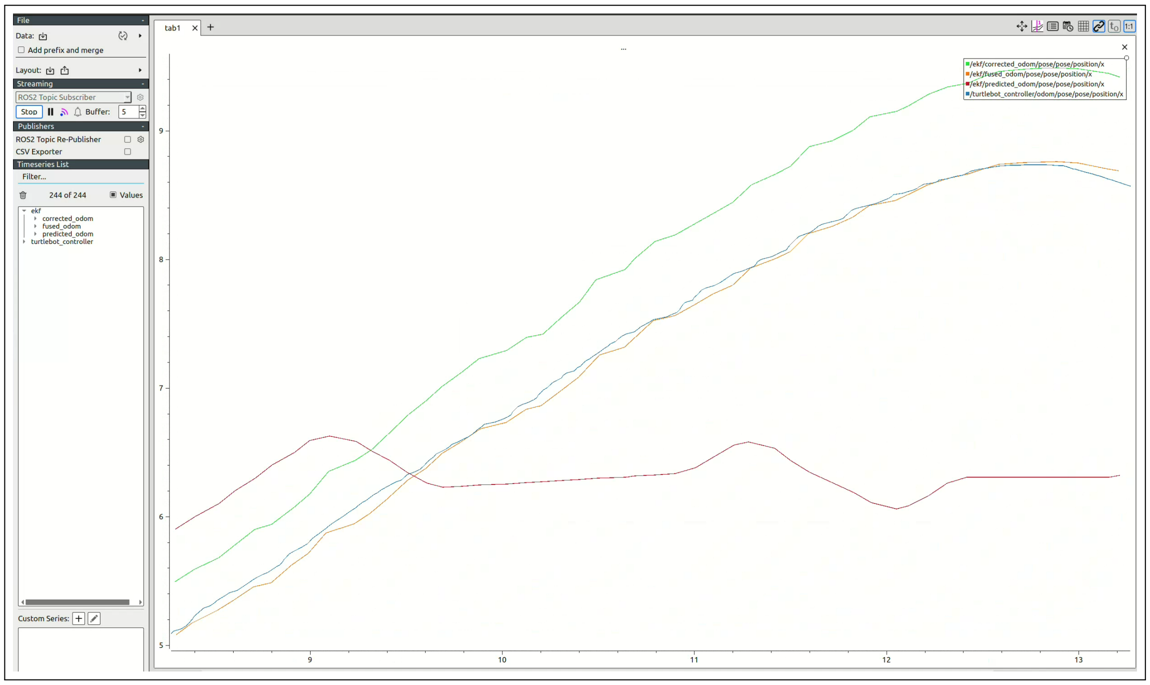

Figure 5: Variation of X coordinate

This plot also highlights the quick response of the state transition model. Additionally, it illustrates how the state correction using only encoders gradually deviates from the global reference, while the correction achieved by fusing the encoder and GPS data maintains a closer alignment with the true state over time.

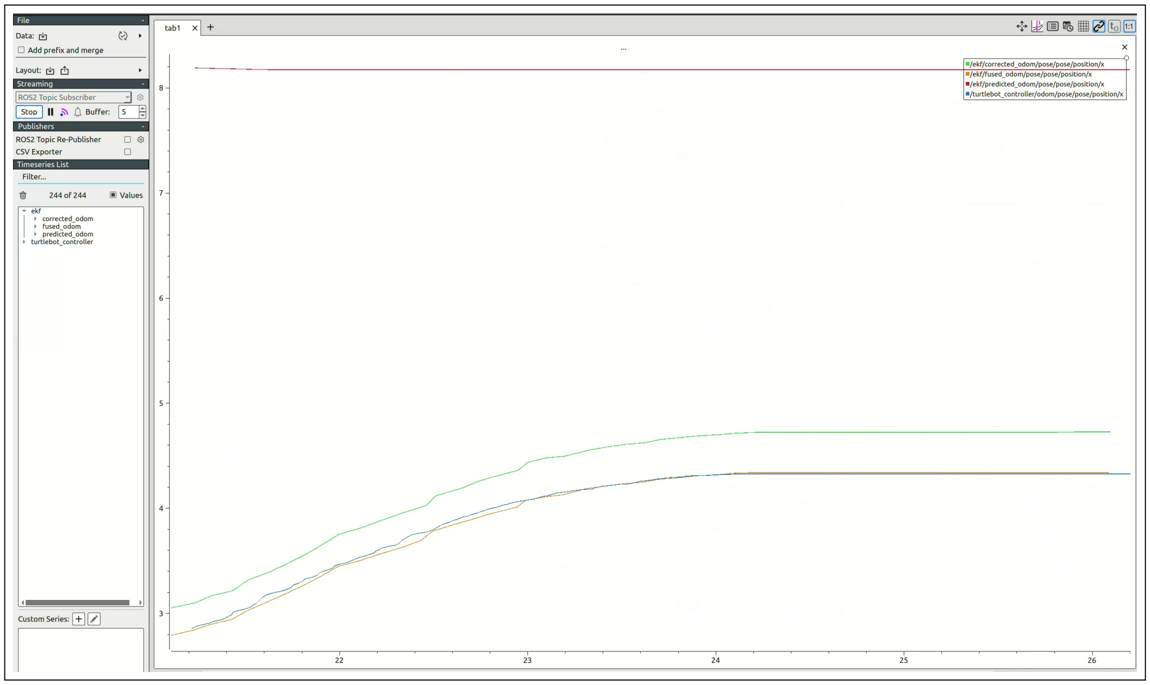

Figure 6: X coordinate at the end of the trjectory

This plot demonstrates how the predicted state deviates by the end of the trajectory. It also shows the deviation of the state corrected by the encoders from the true state, as well as the correction achieved by fusing the encoder and GPS data.

In all cases, however, the fusion-based state estimation is much closer to the ground truth, highlighting the effectiveness of sensor fusion in improving the accuracy of the state estimation.

5 Discussion

Tuning the noise covariance matrices is a critical aspect of implementing the Extended Kalman Filter (EKF) and ensuring optimal performance in state estimation. The two main covariance matrices in the EKF are the process noise covariance matrix Q and the measurement noise covariance matrix R. These matrices characterize the uncertainty in the system's model and measurements, respectively, and need to be carefully tuned for accurate state estimation.

In practice, we set higher values for the process noise covariance Q than measurement noise covariance R. Because even though the process noise covariance Q has a higher uncertainty, by correction step, we will have a Gaussian which has a lower uncertainty because the sensor measurement usually have lower uncertainty than the process step. But if we set much lower uncertainty values for the process noise covariance matrix, then the EKF will give the priority to the predicted state rather than the corrected state, which can lead to inaccurate results.

It is important to note that for this experiment, we assumed the robot to be operating in an outdoor environment with accessible GPS signals. However, in real-world situations, GPS measurements can be affected by short-term noise or signal loss, particularly in environments with poor satellite visibility or interference, such as urban canyons or heavily wooded areas. In such a case, we can implement the EKF to give priority to the encoder measurments. These conditions could impact the performance of the GPS, introducing inaccuracies that might affect the fusion-based state estimation.

Additionally, while the encoders provide a valuable correction to the predicted state, they are also subject to errors such as drift, slippage, and inaccuracies from wheel odometry. These issues can be amplified over longer distances or on uneven terrain, further underscoring the need for sensor fusion to achieve a more reliable and accurate estimation of the robot's state.

Overall, the results demonstrate that sensor fusion, combining encoder and GPS data, provides a significant improvement over relying on a single sensor. This approach helps mitigate the individual limitations of each sensor, leading to a more accurate and robust estimation of the robot's position and orientation. Future work could explore techniques to address the noise and drift inherent in each sensor type, such as incorporating IMUs or employing more advanced filtering methods, to further enhance the state estimation performance in real-world applications.

6 Conclusion

In this work, we have explored the use of sensor fusion to improve state estimation for a mobile robot using the Extended Kalman Filter (EKF). By combining measurements from both encoders and GPS, we were able to obtain more accurate estimates of the robot's state compared to relying solely on the state transition model or encoder data.

The results demonstrated the importance of sensor fusion in reducing the errors that accumulate over time when relying solely on encoders, especially for longer trajectories. While the encoder-based corrections improved the estimates, the fusion of GPS and encoder measurements provided a significant enhancement, resulting in estimates that were much closer to the true state.

We also discussed the critical process of tuning the noise covariance matrices, Q and R, which govern the process and measurement noise. Proper tuning of these matrices is essential to ensure the balance between the predicted state and sensor measurements, thereby optimizing the filter's performance.

Overall, the approach of sensor fusion applied here provides a robust method for improving state estimation in real-world scenarios, especially for outdoor robots where GPS and encoder measurements can be noisy and subject to errors.

In practical applications, it is important to account for real-world challenges such as GPS signal noise and other uncertainties. However, with proper calibration and tuning, sensor fusion can significantly enhance the accuracy of state estimation, making it a valuable tool for autonomous robotic systems.

GitHub Repository

Source code and further details of this project are in the following GitHub repository:

https://github.com/SadeepRathnayaka/Extended_Kalman_Filter_Implementation