1 Introduction

A recorded bag file with data from a Turtlebot3 Gazebo simulation was provided. This bag file included a predefined path that the robot travelled, all topics including various sensor measurements, and ground truth data. The /odom ROS topic provides a dead-reckoning position estimate of the robot, with respect to the starting location of the robot. Ideally, this dead-reckoning estimate would be noise-free and provide the true position of the robot. However, the odometry estimate of a robot tends to accumulate error and thus drift over time. To counteract this, a localization algorithm is employed as it is necessary to have an accurate estimate of the robot's position at all times.

The aim of this lab is to localize and estimate the pose of our robot using a Kalman Filter. Applying a Kalman Filter algorithm to this localization problem is known as "Kalman Filter Localization". The Kalman Filter is a parametric (Gaussian) filter suitable for tracking problems where the system behaviour is linear. It is a fast and efficient algorithm that uses very little memory and computational power compared to a Particle Filter, as it has a closed form equation. The Kalman Filter algorithm essentially filters out noisy inputs and sensor measurements, to provide a better estimate of the robot state.

An important note about the basic Kalman Filter is that the state transition of the motion model must be linear, i.e., able to be represented through linear matrix multiplications.

2 Theory and Methodology

The Kalman Filter Algorithm is shown in Figure 1. The algorithm works in 2 steps: a prediction, and a correction/update. During the prediction step, a state and covariance prediction for the current timestep is computed using past state, covariance, control input u_{k-1}, state transition matrix A, and a state transition uncertainty Q. This is done without incorporating the current sensor measurements. Then during the correction / update step, we take the prediction and compute a best estimate of the state by incorporating/fusing with the latest sensor measurements y_k and measurement uncertainty R.

The main idea behind the Kalman Filter is that each update from the sensors is used to statistically improve the estimate of position of the robot. Simultaneously, the accuracy of the sensor measurements and the state transition model is also determined. This is done through the Kalman gain K, which indicates how much we should rely on sensor measurements y_k vs the prediction from state transition $\hat{x}_{k|k-1}$. This is better than calculating the average of the sensors or motion model input, as these sensors can report inaccurate measurements. So, it is extremely useful to have an algorithm that can account for the measurement and motion model inaccuracies.

Start with initial guess:

$$\hat{x}_{0|-1} = E[x_o] = mean$$
$$P_{o|-1} = var(x_o)$$

Prediction

$$\hat{x}_{k|k-1} = A\hat{x}_{k-1|k-1} + Bu_{k-1}$$
$$P_{k|k-1} = AP_{k-1,k-1}A^T + Q$$

Update

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K(y_k - C\hat{x}_{k,k-1})$$
$$K = P_{k|k-1}C^T(CP_{k|k-1}C^T + R)^{-1}$$
$$P_{k,k} = (I - KC)P_{k|k-1}$$

Figure 1: Kalman Filter Algorithm [1]

The state tracked by the Kalman Filter is the position of the robot in the robot's frame of reference. That is, $\hat{x}_k = [x, \dot{x}, \theta, \omega]^{\mathrm{T}}$, where x is the distance the robot has travelled along the path, and θ is the orientation of the robot with respect to its starting orientation. As we are extremely certain of the initial robot pose, since we know its starting position is at the origin, the initial estimate of the state is $\hat{x}_{0|-1} = [0\ 0\ 0\ 0]^T$ and initial covariance $P_{0|-1} = I$.

These states were selected as above, to obtain a linear state transition matrix, A. The control input to the Kalman Filter is the IMU data, as this is what we expect the robot to have moved based on commanded velocities. That is, $u_{k-1} = \begin{bmatrix} a_x \\ \omega \end{bmatrix}$ where a_x is the IMU's provided linear acceleration measurement in the x direction (forward facing direction of the robot) and ω is the angular velocity of the robot, determined by the IMU's gyroscope in the z direction. Through these, we can obtain the equations for the motion model (state transition) matrices A, B:

$$A = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad and \quad B = \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \\ 0 & 1 \end{bmatrix}$$

Note that although the last row and column of A is all 0's, forming the state and state transition matrix in this way allows a straightforward update of the states using sensor measurements and the C matrix, later on.

The state transition uncertainty Q is found using the equation shown below:

$$Q = BQ_aB^T, \text{where } Q_a = \begin{bmatrix} \sigma^2_{linear} & 0 \\ 0 & \sigma^2_{angular} \end{bmatrix}$$

For determining, σ^2_{linear} and $\sigma^2_{angular}$, the corresponding entries of the "covariance" fields of the IMU sensor messages on the /imu topic were investigated – the sensor shows a linear acceleration variance of 0.00289 in each axis, and an angular velocity variance of $4*10^{-8}$. A plot of the IMU data in the ROS bag (see Appendix 7.4) shows that the linear acceleration measurements from the IMU are quite noisy while the angular velocity measurement appears quite stable. Since the angular velocity input ω is thus much more accurate than a_x, it is evident that we should have $\sigma^2_{linear} >> \sigma^2_{angular}$, which matches the values found above.

Not all states of the robotic system are observable through sensor measurements. In this project, we are not able to directly measure the states of the robot. The sensor measurements that can be measured are $y_k = [u_l \quad u_r \quad \omega_{calculated}]^T$, where u_l and u_r are the velocities of the left and right wheels in rad/s, and $\omega_{calculated} = r * \frac{u_r - u_l}{T}$. These wheel velocities are taken from the wheel encoder measurements on the /joint_states ROS topic. Then using the equations for a 2-wheeled differential drive robot from lectures, the following measurement matrix can be formed:

$$C = \begin{bmatrix} 0 & 1/r & 0 & -T/2 \\ 0 & 1/r & 0 & +T/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This allows mapping of tracked states to a theoretical sensor measurement, for comparison with the actual sensor measurement. The robot dimensions (wheel radius r =33mm and wheelbase T=160mm) were found from the Turtlebot3 Burger datasheet [2].

R is a 3x3 diagonal matrix, where diagonal elements are based on variances listed in the datasheet of the encoders of the TurtleBot3 Burger model [3], However through trial and error for tuning this parameter, diagonal elements of 0.05 variance provided the best results.

2.1 Other implementation notes

Investigating the provided bag file shows several key pieces of info:

- The position of the robot as per the /tf topic is the same as the /odom topic
- Message rate of the /odom topic: ~28Hz
- Message rate of the /joint_states topic: ~28Hz
- Message rate of the /imu topic: ~196Hz

Thus the IMU and joint_states topics and their sensors are outputting data at different rates. To run the Kalman Filter algorithm, a fixed Δt is desired. This implies the Kalman filter can run either at the rate of the slowest sensor, or the most reliable one. In the case of this project, they are the same – the encoder velocities are more accurate and are the slowest data input. For this reason, I decided to run the Kalman Filter at a fixed rate of 25Hz, which is slightly slower than the most reliable sensor measurements.

I implemented this project in Python as a ROS2 package, that works with the Gazebo simulation environment in real time. I created all code from scratch, while referencing the provided Kalman Filter example. As I have implemented the project as a ROS package, I created several ROS nodes based on logical blocks:

- Kalman Filter ROS Node: receives latest control inputs and sensor measurements, and runs Kalman Filter at a fixed rate
- Frame Transformer ROS node: converts between global and robot reference frames, keeping track of incremental updates in each frame
- Path Visualizer ROS Node: visualizes ground truth and Kalman Filter reported path in RViz
- Plotter ROS Node: Calculates MSE and generates plots

The connections between each of these ROS nodes and the rest of the system is shown in the ROS graph in Appendix 7.2.

The ROS node implementation code for the above nodes is shown in Appendix 7.5 to 7.8.

3 Results

Given that the /odom topic provides the same data as the /tf topic, /odom was used as "ground truth" for simplicity. Normally however, the /tf topic would be used for ground truth-comparison.

Through trial and error, I tuned covariance matrices Q, R from their initial values given by datasheets. Figure 2 shows the Robot's path compared to the Ground Truth in the global reference frame. Figure 3 and Figure 4 show plots of the predicted and estimated states by the Kalman Filter compared to the ground truth. Enlarged versions of the plots (with higher resolution) are shown in Appendix 7.1.

We see that the Kalman Filter's estimated states very closely match the Ground truth states. Qualitatively, it appears that the estimated orientation, angular velocity,

and position of the robot are close to the ground truth. The linear velocity of the robot, however, appears to be estimated to be slightly lower than the ground truth.

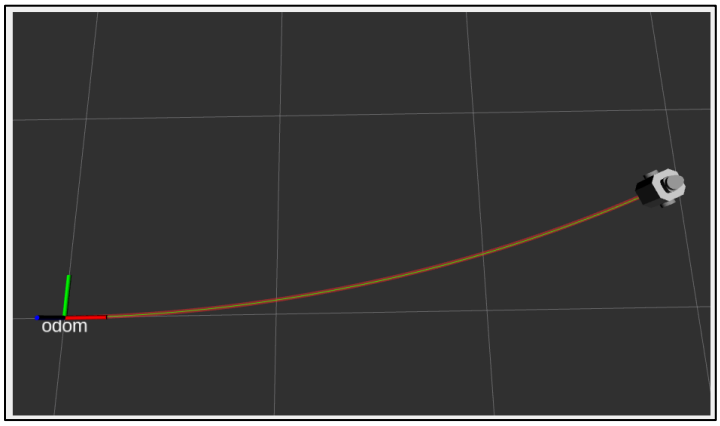

Figure 2: Robot path. Green is ground truth, Red is Kalman Filter

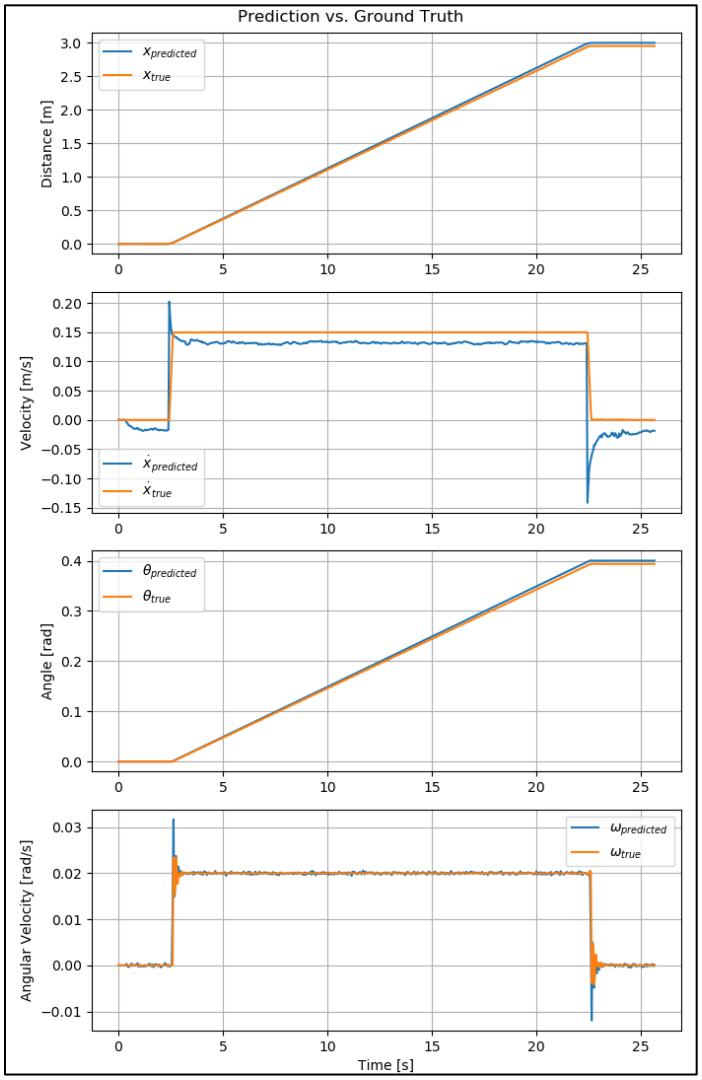

Figure 3: Plot of States - Predicted vs. Ground Truth

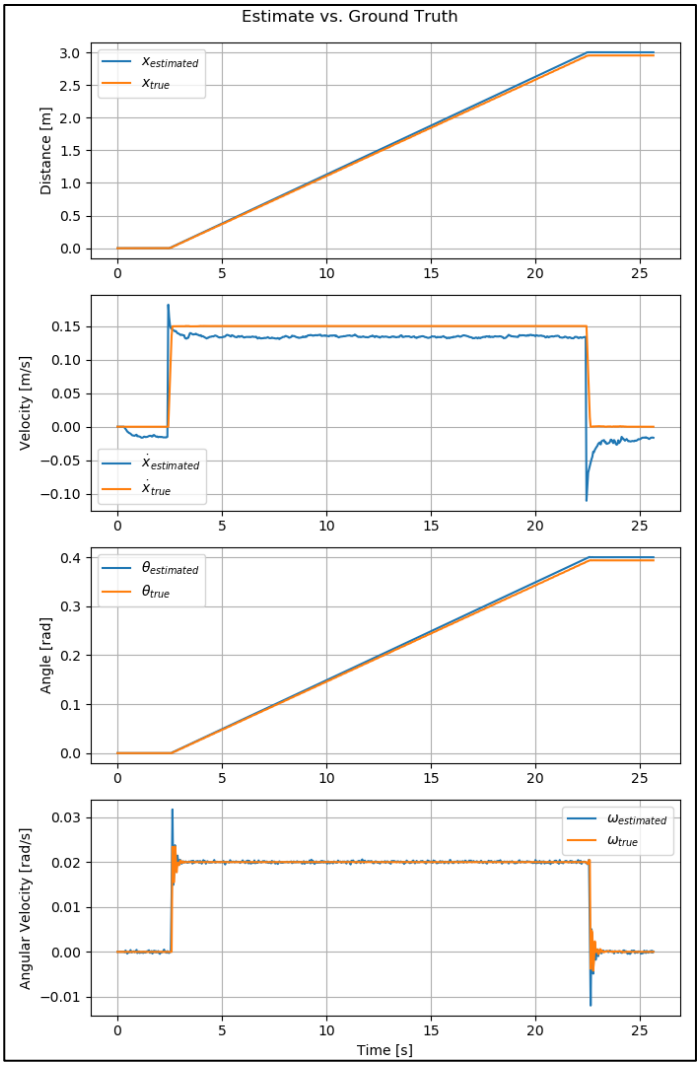

Figure 4: Plot of States – Estimated (after correction) vs. Ground Truth

As a measure of quantitative performance, the mean square error (MSE) is computed between Kalman Filter states and the ground truth. After tuning all parameters (Q, R, filter rate) the MSE is as shown in Table 1.

Table 1: MSE for Predicted and Estimated states on provided bag file

Data Point	MSE	Data Point	MSE
$x_{predicted}$	0.000971	$x_{estimated}$	0.001
$\dot{x}_{predicted}$	0.000759	$\dot{x}_{estimated}$	0.0006
$\theta_{predicted}$	$1.914*10^{-5}$	$\theta_{estimated}$	$1.914*10^{-5}$
$\omega_{predicted}$	$8.799*10^{-7}$	$\omega_{estimated}$	$8.799*10^{-7}$

Interestingly, the implemented linear Kalman Filter algorithm provides very impressive tracking on the orientation of the robot, but less so on the distance the robot has traveled.

4 Discussion

Overall, the Kalman Filter is able to track the state of the robot quite well. As discussed earlier, the plot of the IMU data in Appendix 7.4, shows that the acceleration measurements from the IMU are quite noisy and has a non-zero negative bias when there should be no acceleration (constant velocity). The angular velocity measurement however appears quite stable, and without bias, close to the ground truth of ~0.2rad/s. As shown in the Kalman Filter outputs in the figures above, the Kalman Filter can withstand the erroneous measurements in the IMU acceleration measurement.

Varying covariance values in the Q matrix would influence the Kalman Filter's behaviour. Increasing the magnitude of covariances in the Q matrix would indicate a higher uncertainty in the state transition model, indicating that the Kalman Filter should instead rely more on the sensor measurements rather than the model of the system. Similarly, decreasing the magnitude of covariances in Q would indicate the control inputs and state transition model are extremely accurate.

As mentioned above, since the IMU readings of ω were seen to be much more accurate than a_x, the $\sigma^2_{angular}$ parameter of Q could be tuned to be even smaller than that which was used, and similarly σ^2_{linear} could be increased to indicate the linear acceleration control input is not very accurate. Then, the Kalman Filter estimated states will be even closer to the ground truth states.

Varying covariance values in the R matrix would also influence the Kalman Filter's behaviour. Increasing the magnitude of covariances in the R matrix would indicate a higher uncertainty in the sensor measurements, indicating that the Kalman Filter should instead rely more on the state transition model rather than the erroneous sensor measurements. Similarly, decreasing the magnitude of covariances in R would indicate the sensor measurements y_k are extremely accurate.

Additionally, the static transformation from the robot to the IMU shows a translation of $[x, y, z] = [-0.03, 0, 0.068]$ and 0° rotation (identity quaternion). This implies only a very minor translation, and no rotation. Thus, for this project it was not deemed necessary to perform a coordinate transformation on IMU data. However, for greater accuracy of sensor measurements as well as if the IMU sensor is placed differently on the TurtleBot3, a coordinate transformation will be required.

Furthermore, the /odom topic was used as ground truth. In a real system, assuming that a ROS localization package was also running, the /tf topic should be used as ground truth comparison instead.

Lastly, the provided path is a nearly straight, but slightly curved path. I tested similar paths by generating different paths in a Gazebo Turtlebot3 empty world simulation, which yielded similar results to those shown above. Additionally, I tested with a nonlinear path (large variation in angles) and found that the implemented linear Kalman Filter is in fact not able to cope with these non linearities well. A video and plot of the test in Gazebo is shown in Appendix 7.2. We see that until the robot does many quick turns near the center of the trail, the robot's state has been tracked quite well. However, after this, the Kalman Filter estimated path and the ground truth /odom reported path start to diverge. The implementation of a nonlinear Kalman Filter such as an Extended Kalman Filter or Unscented Kalman Filter would enable tracking of these nonlinearities, as well as possibly provide a direct estimate of the Robot's pose in the global frame by directly tracking the nonlinear states.

5 Conclusion

The aim of this lab was to localize and estimate the pose of our robot using a Kalman Filter, given noisy control inputs and sensor measurements.

Overall, objectives were met for the project as the Kalman Filter algorithm was able to track the state of the robot closely and accurately for the provided path. Estimating the location of a robot, while accounting for sensor noise and odometry drift, is necessary for accurate autonomous navigation.

6 Appendix - References

[1] Y. Hu, "MTE 544 Final Project," 2022.

[2] ROBOTIS, "Turtlebot3," ROBOTIS, 13 Feburary 2022. [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/. [Accessed 18 December 2022].

[3] ROBOTIS, "XL430-W250," Robotis, 3 November 2022. [Online]. Available: https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/. [Accessed 18 December 2022].

7 Other Appendices

7.1 Enlarged Plots for provided bag file

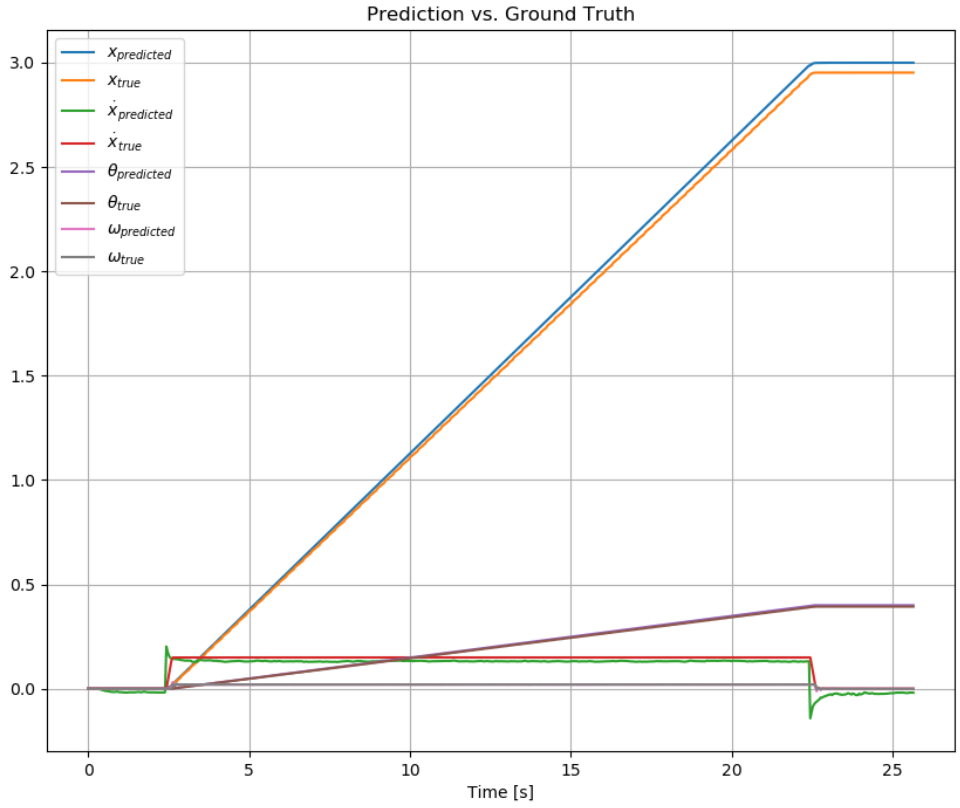

Figure 5: Overall Plot of Predicted states

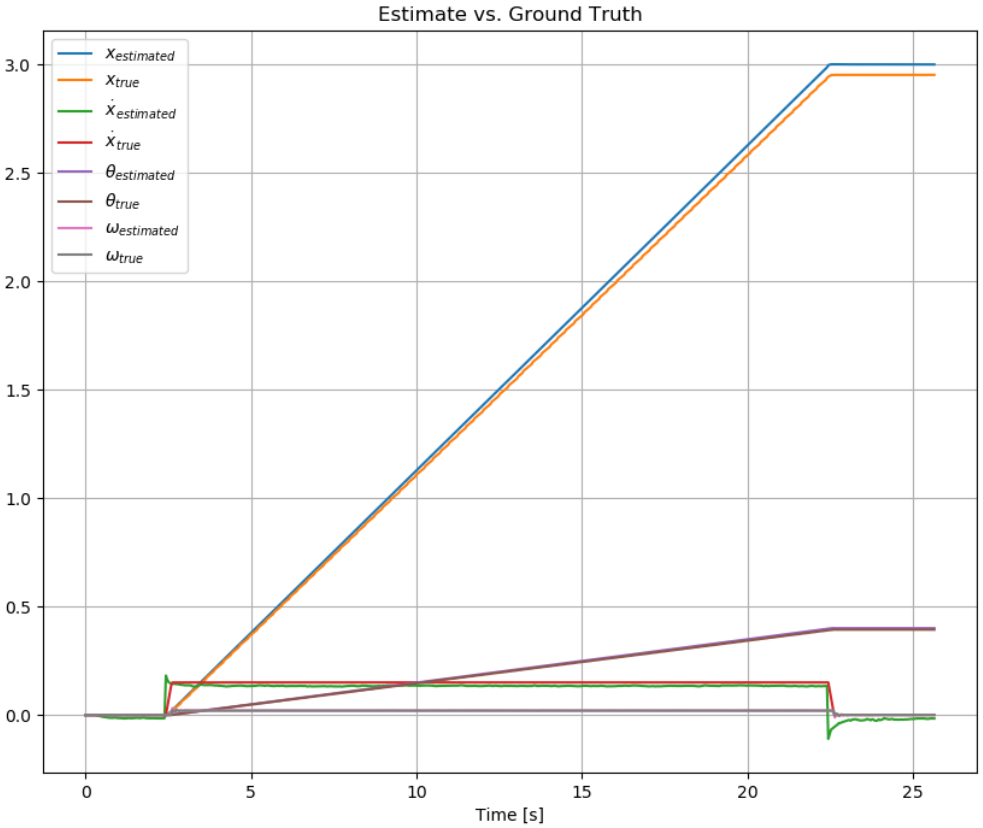

Figure 6: Overall Plot of Estimated states

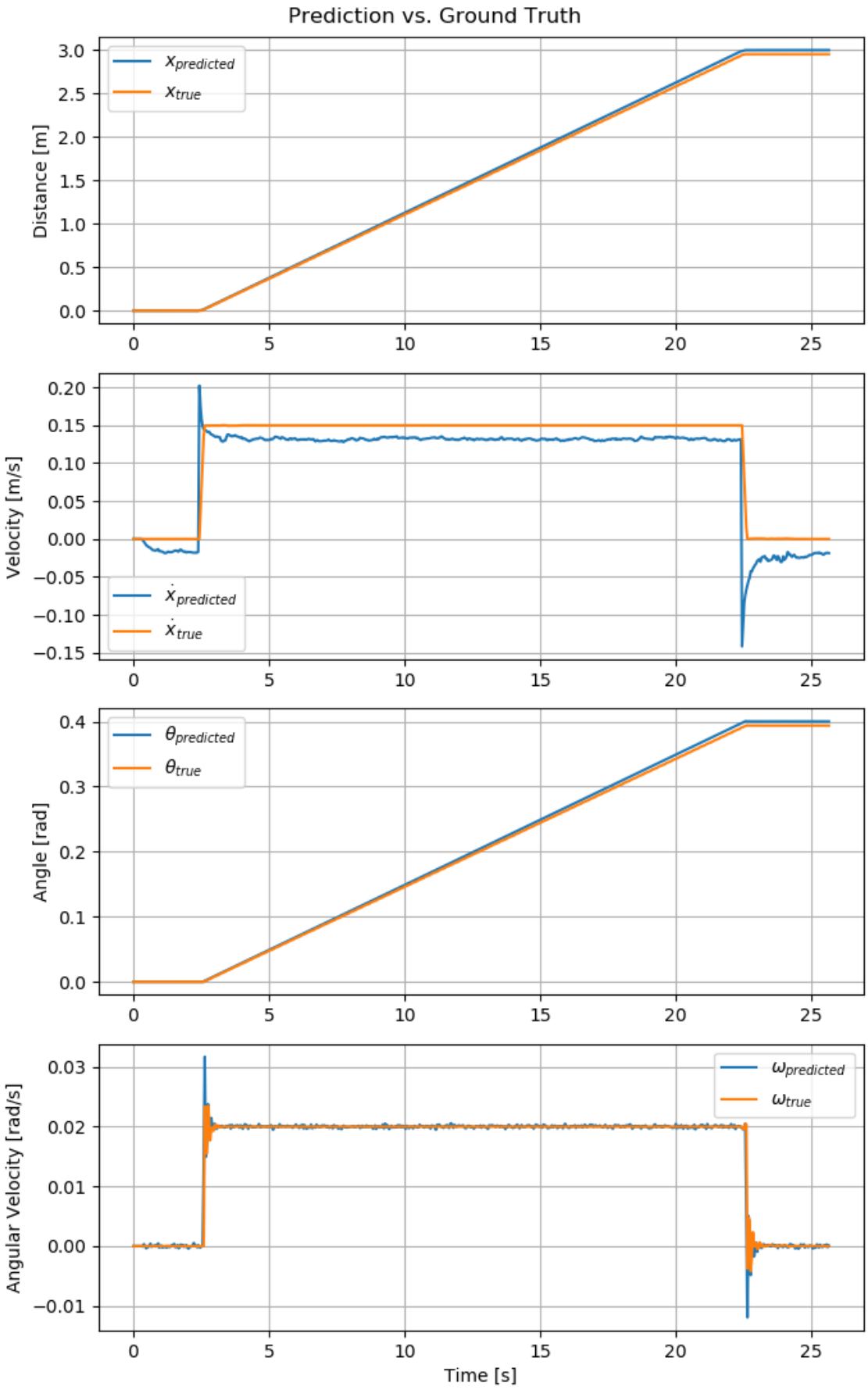

Figure 7: Detailed Plot of States - Predicted vs. Ground Truth

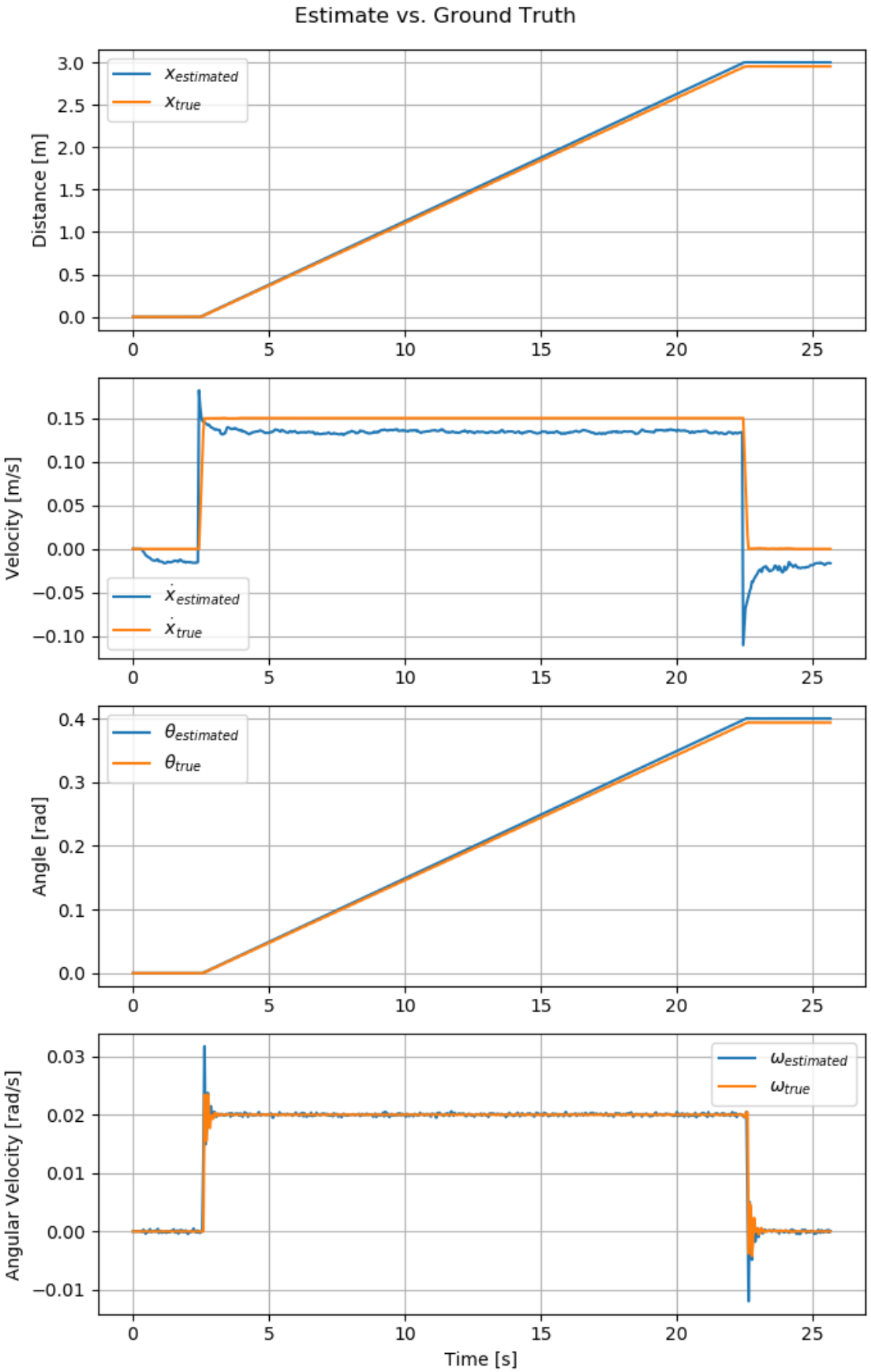

Figure 8: Detailed Plot of States - Estimate vs. Ground Truth

7.2 ROS Graph

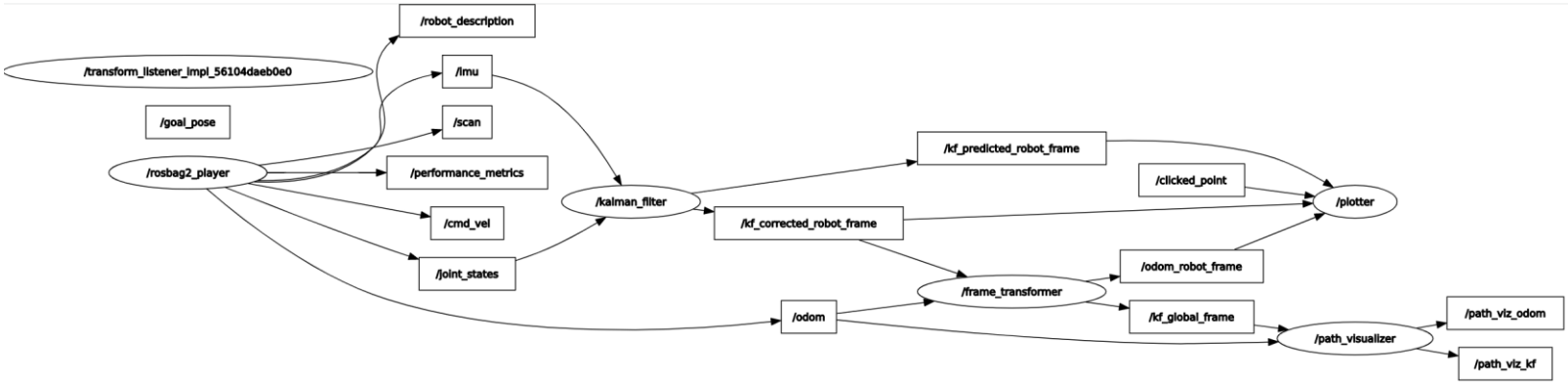

Figure 9: ROS Graph

7.3 Non-linear Paths

Video of nonlinear path simulation: https://youtu.be/7CEmD8rd0zs.

Figure 10 below shows the tracked path in RViz. The green line is the path provided by the /odom topic, while the red line is the path output by the Kalman Filter algorithm.

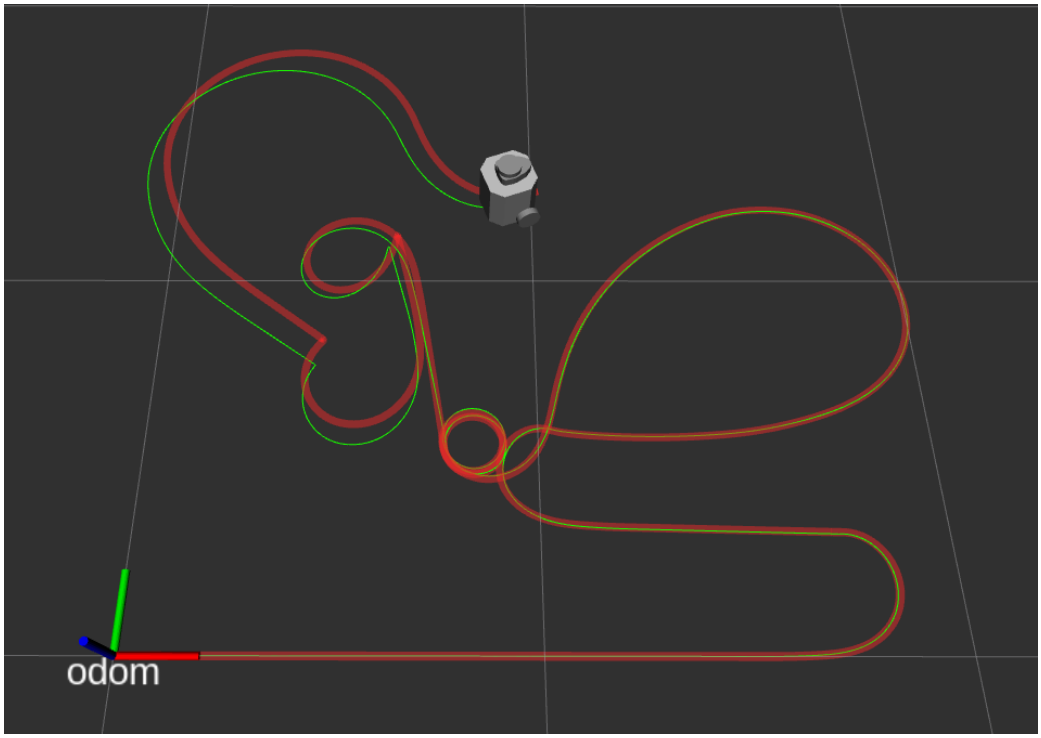

Figure 10: Robot path in RViz. Green is ground truth, Red is Kalman Filter

Table 2: MSE Statistics for Predicted and Estimated states on nonlinear path

Data Point	MSE with Ground Truth	Data Point	MSE with Ground Truth
$x_{predicted}$	0.00248	$x_{estimated}$	0.00254
$\dot{x}_{predicted}$	0.00110	$\dot{x}_{estimated}$	0.000848
$\theta_{predicted}$	79.976	$\theta_{estimated}$	79.976
$\omega_{predicted}$	0.00186	$\omega_{estimated}$	0.00186

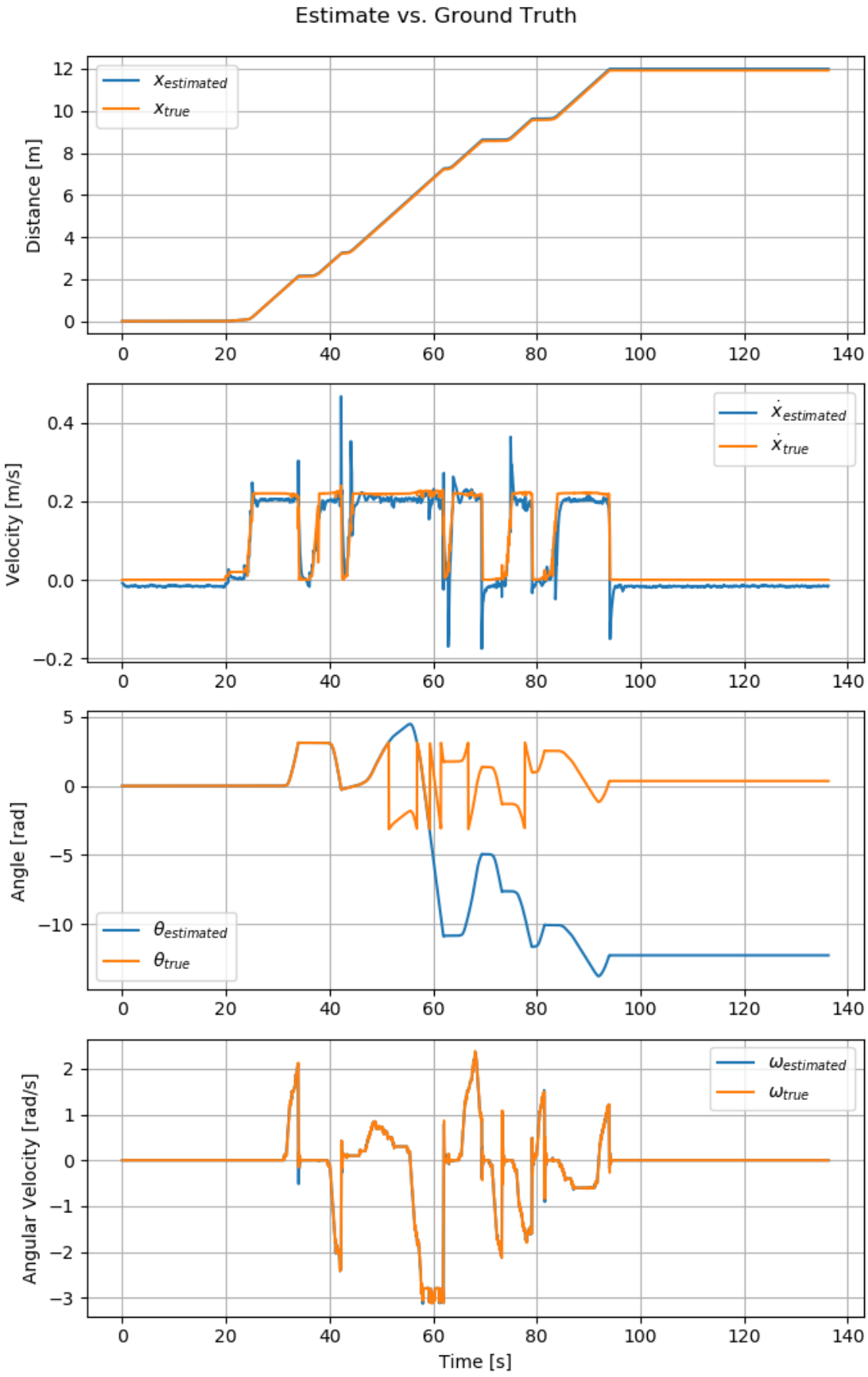

Figure 11: Detailed Plots of States for Nonlinear path

7.4 IMU data plot

Figure 12: Plot of the IMU data

7.5 Kalman Filter ROS Node

```
#!/usr/bin/env python3

import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.qos import ReliabilityPolicy, QoSProfile
from sensor_msgs.msg import Imu, JointState

from kalman_filter_interfaces.msg import RobotFrameState # Custom message type

# Parameters from Turtlebot3 datasheet, in meters
# https://emanual.robotis.com/docs/en/platform/turtlebot3/features/
WHEEL_RADIUS = 0.066/2
WHEEL_BASE = 0.16

class KalmanFilterNode(Node):
    """Kalman Filter to track Robot states in robot reference frame"""

    def __init__(self):
        super().__init__('kalman_filter')
        # Subscribers for receiving IMU input and Encoder velocity measurements
        self.imu_sub = self.create_subscription(Imu, '/imu', self.imu_callback, QoSProfile(depth=300,
reliability=ReliabilityPolicy.BEST_EFFORT))
        self.encoder_sub = self.create_subscription(JointState, '/joint_states', self.encoder_callback,
QoSProfile(depth=300, reliability=ReliabilityPolicy.BEST_EFFORT))
        # Publishers for notifying other nodes on tracked robot position, in the robot reference frame
        self.state_predicted_pub = self.create_publisher(RobotFrameState, '/kf_predicted_robot_frame', 10)
        self.state_corrected_pub = self.create_publisher(RobotFrameState, '/kf_corrected_robot_frame', 10)
        self.state_msg = RobotFrameState()

        self.dt = 1.0/25.0 # Time step of Kalman Filter
        self.xhat = np.matrix([0.0, 0.0, 0.0, 0.0]).transpose() # mean(mu) estimate for the "first" step
        self.xhat_predicted = np.matrix([0.0, 0.0, 0.0, 0.0]).transpose() # mean(mu) estimate for the "first"
step
        self.P = np.identity(4)
        self.A = np.matrix([[1, self.dt, 0, 0],
                            [0, 1, 0, 0],
                            [0, 0, 1, 0],
                            [0, 0, 0, 0]]) # state space transition
        self.B = np.matrix([[1/2*self.dt**2, 0],
                            [self.dt, 0],
                            [0, self.dt],
                            [0, 1]])
        self.Qa = np.matrix([[0.000289, 0],
                            [0, 4*10**(-8)]]) # Covariance values used from IMU message
        self.Q = self.B*self.Qa*self.B.transpose() # Motion Model / state transition covariance
        self.C = np.matrix([[0, 1/WHEEL_RADIUS, 0, -WHEEL_BASE/2],
                            [0, 1/WHEEL_RADIUS, 0, +WHEEL_BASE/2],
                            [0, 0, 0, 1]]) # Measurement Model. Translates tracked states to a corresponding
expected measurement
        self.R = np.matrix([[0.05, 0, 0], [0, 0.05, 0], [0, 0, 0.05]]) # Sensor Model Covariance
        self.u = np.matrix([0.0, 0.0]).transpose() # State Transition Inputs: linear acceleration in x
(forward direction of robot), and omega
        self.y = np.matrix([0.0, 0.0, 0.0]).transpose() # Sensor Measurements: u_l, u_r, omega_calculated

        # Run Kalman Filter at fixed rate, slower than all subscribers. This synchronizes the measurements
        self.timer = self.create_timer(self.dt, self.run_kalman_filter)

    def imu_callback(self, msg: Imu):
        """Record latest IMU Input"""
        w = msg.angular_velocity.z # omega
        a_x = msg.linear_acceleration.x # linear acceleration in x direction
        #Note: we do not need to apply a transform from imu_link to base_footprint as
        # the static transform shows a offset in the x-direction, and no rotational offset
        # Thus, the x-axis of the IMU aligns with x-axis of the robot.
```

```
        self.u = np.matrix([a_x, w]).transpose()
        # self.get_logger().info(f"IMU Angular Velocity: {w}")

    def encoder_callback(self, msg: JointState):
        """Record latest Sensor Measurement"""
        left_wheel_speed = msg.velocity[0] # rad/s
        right_wheel_speed = msg.velocity[1] # rad/s
        omega_calculated = WHEEL_RADIUS * (left_wheel_speed-right_wheel_speed)/WHEEL_BASE
        self.y = np.matrix([left_wheel_speed, right_wheel_speed, omega_calculated]).transpose()

    def run_kalman_filter(self):
        """Run 1 iteration of Kalman Filter Algorithim"""
        # Prediction update
        self.xhat_predicted = self.A * self.xhat + self.B * self.u
        P_predict = self.A*self.P*self.A.transpose() + self.Q
        # Measurement Update and Kalman Gain (Correction)
        K = P_predict * self.C.transpose()*np.linalg.inv(self.C*P_predict*self.C.transpose() + self.R)
        self.xhat = self.xhat_predicted + K * (self.y - self.C * self.xhat_predicted)
        self.P = (np.identity(4) - K * self.C) * P_predict

        self.publish_states()

    def publish_states(self):
        """Publish predicted and corrected states for other nodes"""
        self.state_msg.header.frame_id = 'odom' # set origin to be coincident with 'odom' frame
        self.state_msg.header.stamp = self.get_clock().now().to_msg()
        # Prediction
        self.state_msg.x = self.xhat_predicted[0, 0]
        self.state_msg.x_dot = self.xhat_predicted[1, 0]
        self.state_msg.theta = self.xhat_predicted[2, 0]
        self.state_msg.omega = self.xhat_predicted[3, 0]
        self.state_predicted_pub.publish(self.state_msg)
        # Correction
        self.state_msg.x = self.xhat[0, 0]
        self.state_msg.x_dot = self.xhat[1, 0]
        self.state_msg.theta = self.xhat[2, 0]
        self.state_msg.omega = self.xhat[3, 0]
        self.state_corrected_pub.publish(self.state_msg)

def main(args=None):
    rclpy.init(args=args)
    kalman_filter = KalmanFilterNode()
    try:
        while rclpy.ok():
            rclpy.spin_once(kalman_filter)
    except KeyboardInterrupt:
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

7.6 Frame Transformer ROS Node

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from nav_msgs.msg import Path
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseStamped
from rclpy.qos import ReliabilityPolicy, QoSProfile
from kalman_filter_interfaces.msg import RobotFrameState
import numpy as np
import math

class FrameTransformer(Node):
    """Converts between Robot Frame and global frame"""
```

```
    # ---- Explanation on Reference Frames ------
    # The 'robot frame' tracks states of [x, x_dot, theta, omega]". The `x` in this frame is
    # the distance along the 'path'. `theta` is the orientation of the robot, with respect to the global
frame
    # The 'global frame' tracks position of the robot as [x,y,theta]. `x` and `y` are global positions
    # of the robot. `theta` is again the orientation of the robot, with respect to the global frame

    def __init__(self):
        super().__init__('frame_transformer')
        # Subscribe to ground truth and Kalman Filter state updates
        self.odom_sub = self.create_subscription(Odometry, '/odom', self.odom_callback, QoSProfile(depth=300,
reliability=ReliabilityPolicy.BEST_EFFORT))
        self.kf_correct_sub = self.create_subscription(RobotFrameState, '/kf_corrected_robot_frame',
self.kf_callback, QoSProfile(depth=300, reliability=ReliabilityPolicy.BEST_EFFORT))
        # Publish above states in the oppisite frame
        self.kf_global_pub = self.create_publisher(PoseStamped, '/kf_global_frame', 10)
        self.odom_robot_pub = self.create_publisher(RobotFrameState, '/odom_robot_frame', 10)

        self.kf_global_frame_pose = PoseStamped()
        self.odom_robot_frame_state = RobotFrameState()
        self.kf_robot_frame_state_prev = RobotFrameState()
        self.odom_global_frame_pose_prev = Odometry()

        self.odom_robot_frame_state.header.frame_id = 'odom' # Robot frame origin coincides with odom origin
        self.kf_global_frame_pose.header.frame_id = 'odom' # Set origin as odom, to visualize in RViz

    def odom_callback(self, msg: Odometry):
        """Convert odom message from Global Frame to Robot Frame, for use in MSE comparison"""
        quaternion = (
                msg.pose.pose.orientation.x,
                msg.pose.pose.orientation.y,
                msg.pose.pose.orientation.z,
                msg.pose.pose.orientation.w
                )
        _, _, theta = self.euler_from_quaternion(quaternion)
        delta_x = msg.pose.pose.position.x - self.odom_global_frame_pose_prev.pose.pose.position.x
        delta_y = msg.pose.pose.position.y - self.odom_global_frame_pose_prev.pose.pose.position.y
        delta_x_robot_frame = np.sqrt(delta_x**2 + delta_y**2)
        self.odom_robot_frame_state.header.stamp = self.get_clock().now().to_msg()
        self.odom_robot_frame_state.x += delta_x_robot_frame
        self.odom_robot_frame_state.x_dot = np.sqrt(msg.twist.twist.linear.x**2 +
msg.twist.twist.linear.y**2)
        self.odom_robot_frame_state.theta = theta
        self.odom_robot_frame_state.omega = msg.twist.twist.angular.z
        # Store last message
        self.odom_global_frame_pose_prev = msg

        self.odom_robot_pub.publish(self.odom_robot_frame_state)

    def kf_callback(self, msg: RobotFrameState):
        """Convert Kalman Filter's corrected state from Robot Frame to Global frame, for use in RViz"""
        theta = msg.theta
        delta_x = msg.x - self.kf_robot_frame_state_prev.x
        self.kf_global_frame_pose.header.stamp = self.get_clock().now().to_msg()
        self.kf_global_frame_pose.pose.position.x += delta_x*np.cos(theta)
        self.kf_global_frame_pose.pose.position.y += delta_x*np.sin(theta)
        # Store last message
        self.kf_robot_frame_state_prev = msg

        self.kf_global_pub.publish(self.kf_global_frame_pose)

    def euler_from_quaternion(self, quaternion):
        """
        Convert a quaternion into euler angles (roll, pitch, yaw)
        roll is rotation around x in radians (counterclockwise)
        pitch is rotation around y in radians (counterclockwise)
        yaw is rotation around z in radians (counterclockwise)
        """
```

```
        (x,y,z,w) = quaternion
        t0 = +2.0 * (w * x + y * z)
        t1 = +1.0 - 2.0 * (x * x + y * y)
        roll_x = math.atan2(t0, t1)

        t2 = +2.0 * (w * y - z * x)
        t2 = +1.0 if t2 > +1.0 else t2
        t2 = -1.0 if t2 < -1.0 else t2
        pitch_y = math.asin(t2)

        t3 = +2.0 * (w * z + x * y)
        t4 = +1.0 - 2.0 * (y * y + z * z)
        yaw_z = math.atan2(t3, t4)

        return roll_x, pitch_y, yaw_z # in radians

def main(args=None):
    rclpy.init(args=args)
    transformer = FrameTransformer()
    try:
        while rclpy.ok():
            rclpy.spin_once(transformer)
    except KeyboardInterrupt:
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

7.7 Plotter ROS Node

```
#!/usr/bin/env python3

import rclpy
import matplotlib.pyplot as plt
import numpy as np
from rclpy.node import Node
from geometry_msgs.msg import PointStamped
from rclpy.qos import ReliabilityPolicy, QoSProfile
from kalman_filter_interfaces.msg import RobotFrameState # Custom message type

class Plotter(Node):
    """Creates plots and calculates MSE between Kalman Filter states and ground truth"""

    def __init__(self):
        super().__init__('plotter')
        self.create_subscription(RobotFrameState, '/odom_robot_frame', self.odom_callback,
QoSProfile(depth=300, reliability=ReliabilityPolicy.BEST_EFFORT))
        self.create_subscription(RobotFrameState, '/kf_predicted_robot_frame', self.kf_prediction_callback,
QoSProfile(depth=300, reliability=ReliabilityPolicy.BEST_EFFORT))
        self.create_subscription(RobotFrameState, '/kf_corrected_robot_frame', self.kf_correction_callback,
QoSProfile(depth=300, reliability=ReliabilityPolicy.BEST_EFFORT))
        # create the subscriber object to RViz `Clicked Point`
        self.create_subscription(PointStamped, '/clicked_point', self.perform_analysis, QoSProfile(depth=10,
reliability=ReliabilityPolicy.BEST_EFFORT))

        self.ground_truth = RobotFrameState()
        self.kf_prediction = RobotFrameState()
        self.kf_correction = RobotFrameState()

        self.times = []
        self.ground_truths = []
        self.kf_predictions = []
        self.kf_corrections = []

        self.start_time = self.get_clock().now()

    def get_seconds_since_start(self) -> float:
        """Return seconds since this node started"""
```

```
        return ((self.get_clock().now()-self.start_time).nanoseconds /(10**9))

    def odom_callback(self, msg: RobotFrameState):
        """Store latest odom message"""
        self.ground_truth = msg
        # self.get_logger().info(f"Ground Truth x_dot: {msg.x_dot}")

    def kf_prediction_callback(self, msg: RobotFrameState):
        """Store latest Kalman Filter Prediction message"""
        self.kf_prediction = msg

    def kf_correction_callback(self, msg: RobotFrameState):
        """Store latest Kalman Filter Correction message"""
        self.kf_correction = msg
        # self.get_logger().info(f"Corrected x_dot: {msg.x_dot}")
        # Upon receiving Kalman filter correction, save latest data. This synchronizes ground truth and
Kalman Filter output
        self.record_data()

    def record_data(self):
        """Save current data to an array, for later analysis"""
        self.times.append(self.get_seconds_since_start())
        self.ground_truths.append([self.ground_truth.x, self.ground_truth.x_dot, self.ground_truth.theta,
self.ground_truth.omega])
        self.kf_predictions.append([self.kf_prediction.x, self.kf_prediction.x_dot, self.kf_prediction.theta,
self.kf_prediction.omega])
        self.kf_corrections.append([self.kf_correction.x, self.kf_correction.x_dot, self.kf_correction.theta,
self.kf_correction.omega])

    def perform_analysis(self, msg: PointStamped):
        """Plot and calculate MSE"""
        # Convert to Numpy arrays for analysis
        times = np.array(self.times)
        ground_truths = np.array(self.ground_truths)
        kf_predictions = np.array(self.kf_predictions)
        kf_corrections = np.array(self.kf_corrections)
        # Print data point at ~middle of data
        self.get_logger().info(f"SAMPLE DATA POINT")
        self.get_logger().info(f"Data format: [x x_dot theta omega]")
        self.get_logger().info(f"Time: {times[int(times.shape[0]/2)]}")
        self.get_logger().info(f"Ground Truth: {ground_truths[int(ground_truths.shape[0]/2),:]}")
        self.get_logger().info(f"Prediction: {kf_corrections[int(kf_corrections.shape[0]/2),:]}")
        self.get_logger().info(f"Correction: {kf_predictions[int(kf_predictions.shape[0]/2),:]}")

        self.get_logger().info(f"MSE FOR PREDICTIONS")
        self.make_plots(times,
                        [kf_predictions[:,0], ground_truths[:,0], kf_predictions[:,1], ground_truths[:,1],
kf_predictions[:,2], ground_truths[:,2], kf_predictions[:,3],  ground_truths[:,3]],
                        [r"$x_{predicted}$", r"$x_{true}$", r"$\dot x_{predicted}$", r"$\dot x_{true}$",
r"$\theta_{predicted}$", r"$\theta_{true}$", r"$\omega_{predicted}$", r"$\omega_{true}$"],
                        ['Distance [m]', 'Velocity [m/s]', 'Angle [rad]', 'Angular Velocity [rad/s]'],
                        "Prediction vs. Ground Truth")
        self.get_logger().info(f"MSE FOR ESTIMATES")
        self.make_plots(times,
                        [kf_corrections[:,0], ground_truths[:,0], kf_corrections[:,1], ground_truths[:,1],
kf_corrections[:,2], ground_truths[:,2], kf_corrections[:,3],  ground_truths[:,3]],
                        [r"$x_{estimated}$", r"$x_{true}$", r"$\dot x_{estimated}$", r"$\dot x_{true}$",
r"$\theta_{estimated}$", r"$\theta_{true}$", r"$\omega_{estimated}$", r"$\omega_{true}$"],
                        ['Distance [m]', 'Velocity [m/s]', 'Angle [rad]', 'Angular Velocity [rad/s]'],
                        "Estimate vs. Ground Truth")
        plt.show()

    def make_plots(self, times: np.ndarray, datas: list, line_labels: list, ylabels: list, title: str):
        # Make overall plot
        plt.figure(figsize=(10,8))
        plt.title(title)
        for i in range(len(line_labels)):
            plt.plot(times, datas[i], label=line_labels[i])
```

```
        plt.legend(loc="upper left")
        plt.xlabel("Time [s]")
        plt.grid()

        # Make detailed subplots
        fig, axs = plt.subplots(int(len(line_labels)/2), figsize=(7, 11))
        fig.suptitle(title)
        for i in range(int(len(line_labels)/2)):
            axs[i].plot(times, datas[2*i], label=line_labels[2*i])
            axs[i].plot(times, datas[2*i+1], label=line_labels[2*i+1])
            mse = (np.square(datas[2*i] - datas[2*i+1])).mean()
            self.get_logger().info(f"MSE {line_labels[2*i][1:-1]}<->{line_labels[2*i+1][1:-1]}: {mse}")
            axs[i].legend()
            axs[i].set(ylabel=ylabels[i])
            axs[i].grid()
        axs[-1].set(xlabel="Time [s]")
        fig.tight_layout()

def main(args=None):
    rclpy.init(args=args)
    visualizer = Plotter()
    try:
        while rclpy.ok():
            rclpy.spin_once(visualizer)
    except KeyboardInterrupt:
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

7.8 Path Visualizer ROS Node

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from nav_msgs.msg import Path
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseStamped
from rclpy.qos import ReliabilityPolicy, QoSProfile
class PathVisualizer(Node):
    """Visualizes path topics for Rviz"""

    def __init__(self):
        super().__init__('path_visualizer')
        self.odom_path = Path()
        self.kf_path = Path()
        self.odom_sub = self.create_subscription(Odometry, '/odom', self.odom_callback, QoSProfile(depth=300,
reliability=ReliabilityPolicy.BEST_EFFORT))
        self.kf_sub = self.create_subscription(PoseStamped, '/kf_global_frame', self.kf_odom_callback,
QoSProfile(depth=300, reliability=ReliabilityPolicy.BEST_EFFORT))
        self.odom_path_pub = self.create_publisher(Path, '/path_viz_odom', 10)
        self.kf_path_pub = self.create_publisher(Path, '/path_viz_kf', 10)

    def odom_callback(self, msg: Odometry):
        """Create RViz message for visualizing /odom path"""
        self.odom_path.header = msg.header
        pose = PoseStamped()
        pose.header = msg.header
        pose.pose = msg.pose.pose
        pose.pose.position.z = 0.0
        self.odom_path.poses.append(pose)
        self.odom_path_pub.publish(self.odom_path)

    def kf_odom_callback(self, msg: PoseStamped):
        """Create RViz message for visualizing /kf_global_frame path"""
        self.kf_path.header = msg.header
        self.kf_path.poses.append(msg)
```

```
        self.kf_path_pub.publish(self.kf_path)

def main(args=None):
    rclpy.init(args=args)
    visualizer = PathVisualizer()
    try:
        while rclpy.ok():
            rclpy.spin_once(visualizer)
    except KeyboardInterrupt:
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```