EVER 2024 Autonomous Track

Institution /Team Identification: **IEEE Zagazig SB**

Overview

In this milestone, we achieved making our system navigate any desired path, simulating a car following GPS waypoints while mimicking real-world noise. This ensures that our system can operate effectively in challenging environments. We conducted several experiments. Firstly, we implemented our closed-loop controller. We tested Pure Pursuit, Stanley, and MPC and found that Stanley performed the best for our system. Then, we developed a filtering system to handle the mimicked sensor noises in real-time, using the Kalman Filter. Its performance was satisfactory for our current needs, although we plan to explore other filters in the near future.

Additionally, we implemented an object detection system in the car capable of identifying humans, cars, and cones by using a combination of pre-trained models and training new models for specific cases, such as cones, where no suitable pre-trained open-source model was available. Finally, our system can now interact with the real world.

Methodology Used

Firstly, for the closed-loop controller, we used a simplified vehicle dynamics model considering parameters such as wheelbase, width, and distances from the rear to the front and back of the vehicle obtained from Milestone 1 documentation. We employed CSV files to represent our waypoints and for data logging, including the X and Y positions of the car and its orientation (Yaw). For trajectory planning, we generated it based on the waypoint path using cubic spline interpolation to ensure smooth and continuous motion, as calculated by equations solved with the `math` library in Python.

$$Si(x) = ai + bi(x-xi) + ci(x-xi)^2 + di(x-xi)^3$$

Before beginning to use the feedback data, we want to simulate real-world noise. As required in the milestone, we used Gaussian noise. Thus, we added the noise using the following equation, where x represents the original value and σ is the standard deviation of the noise.

We used 0.5, 0.01, and 0.03 as standard deviations for position, orientation, and velocity noises respectively.

$$noisy_x = x + np.random.normal(0,\sigma)$$

Now, after adding noise to the data, we need to filter it to avoid inputting noisy data into the controller. For this purpose, we used the Kalman Filter to filter the data and make it smoother. We found `pykalman` from `KalmanFilter` to be the most suitable for our needs. We began by initializing the filters with appropriate transition matrices, observation matrices, transition covariances, observation covariances, initial state means, and initial state covariances. For the filtration process, we had two main steps: the "prediction step" and the "update step". In the first step, the filter predicted the current state of the system based on the previous state estimate and the system dynamics model. In the second step, it incorporated new measurements from sensors into the state estimate, adjusting the predicted state based on the measurement residuals and the Kalman Gain, with the library handling the mathematics involved.

For the two previous processes—reading the data, adding noise, filtering the data, and logging the process—we wrote one script to read the data from the `Odom` topic and publish it on a new topic named `filtered_odom` so that our controller can directly work with this data.

Now, we can begin working with the feedback data from the car. We utilized odometry readings such as X position, Y position, Yaw, and speed, and we used the following equations to calculate them from the odometry readings, where Q represents the filtered quaternion and O denotes the orientation.

$$Q = [O_x, O_y, O_z, O_w]$$

$$yaw = euler - from - quaternion(Q)$$

Our main control loop continuously computes steering commands based on feedback from sensors, including Odometry, and the desired trajectory. Additionally, we calculate the lateral error (the distance between the vehicle's position and the desired path) and the heading error (the difference between the vehicle's orientation and the orientation of the path tangent at its current position). The controller computes a steering command based on these errors and other parameters such as the vehicle's velocity and a gain parameter (k).

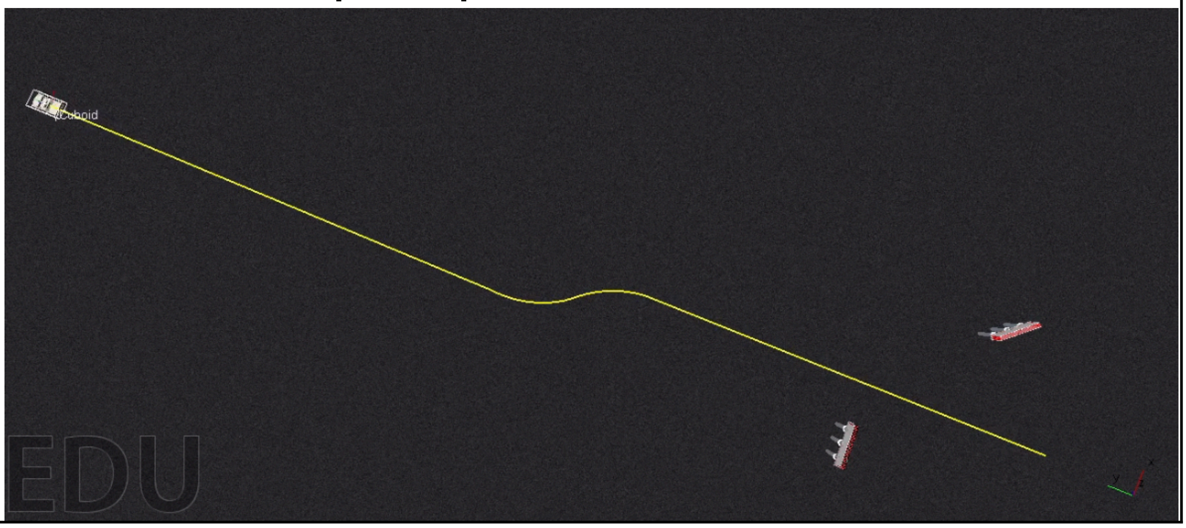
For the object detection part we came across a library called `cvlib` which provides a pre-trained deep learning model designed for detecting common objects. Within this library, there's a particularly useful function called `detect_common_objects` which leverages pre-trained models, using `yolov4` with `yolov4.weights`. This functionality proved instrumental in detecting both cars and people. However, it encountered limitations when it came to identifying traffic cones. To address this gap, we turned to Contour detection, a technique that identifies and extracts the boundaries of objects within an image.

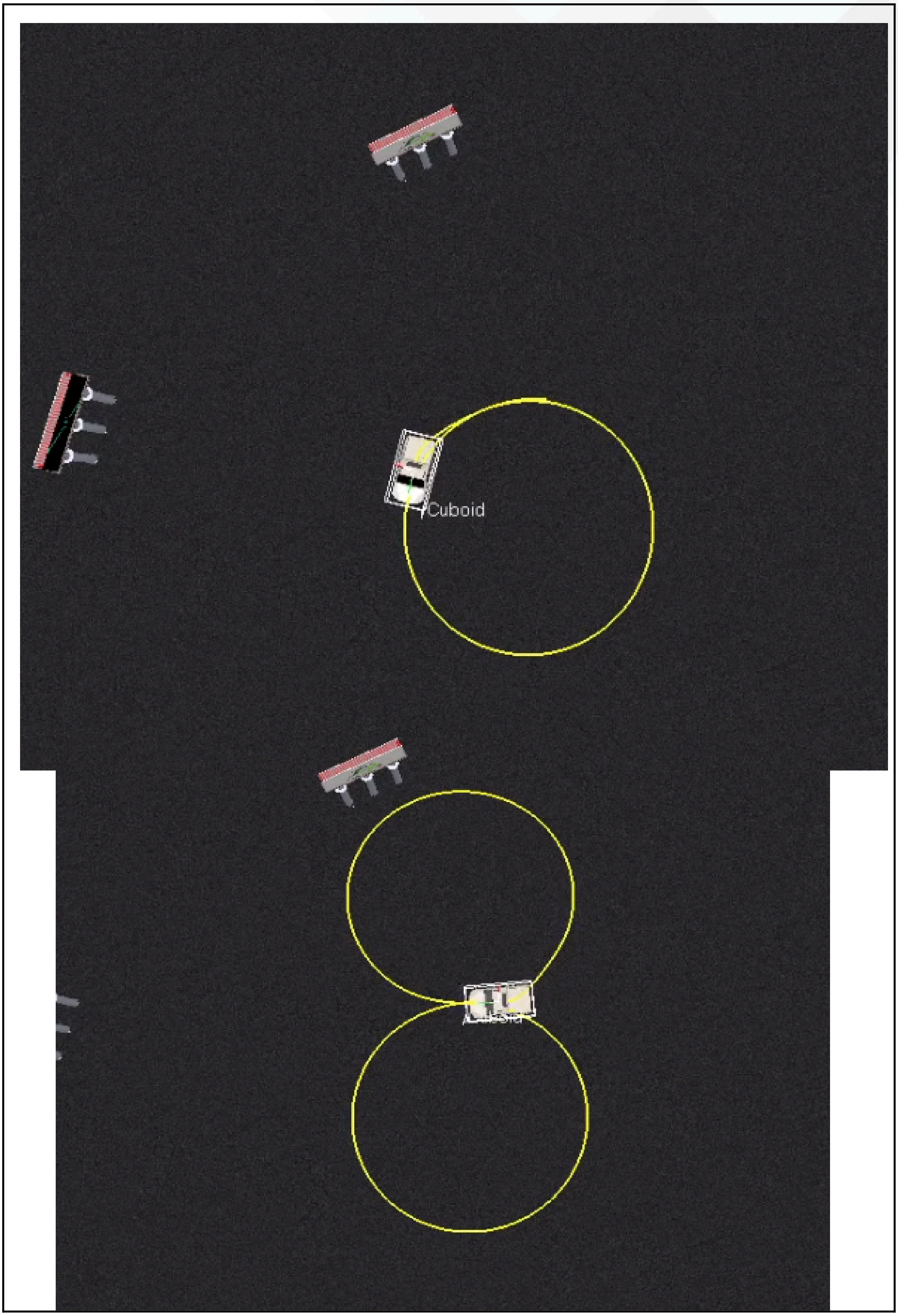
We used the next libraries in the object detection part:

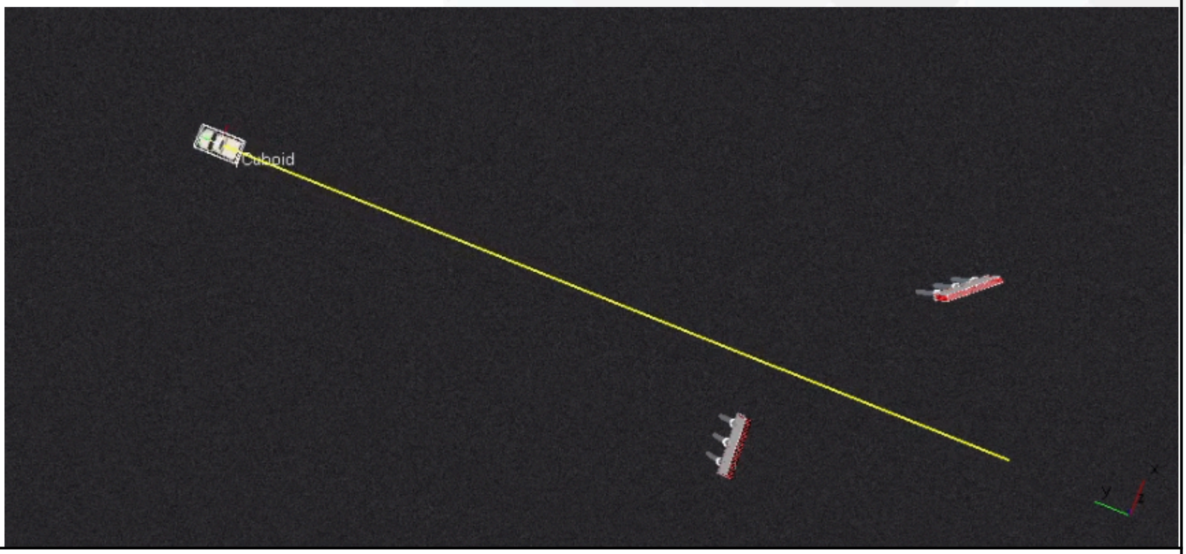
`CvBridge`: A ROS library facilitating the conversion between ROS Image messages and OpenCV images.
`OpenCV` (cv2): A versatile computer vision library renowned for its capabilities in image processing, contour detection, and visualization.
`cvlib`: This high-level computer vision library specializes in common object detection, offering valuable support in our endeavors.
`HSV Color Space`: We used this color space specifically for cone detection, effectively isolating orange hues, which are characteristic of traffic cones.
`Contour Analysis`: This process encompasses contour extraction, approximation, and filtering based on shape, serving as the foundation for our cone detection methodology.

Built Tracks

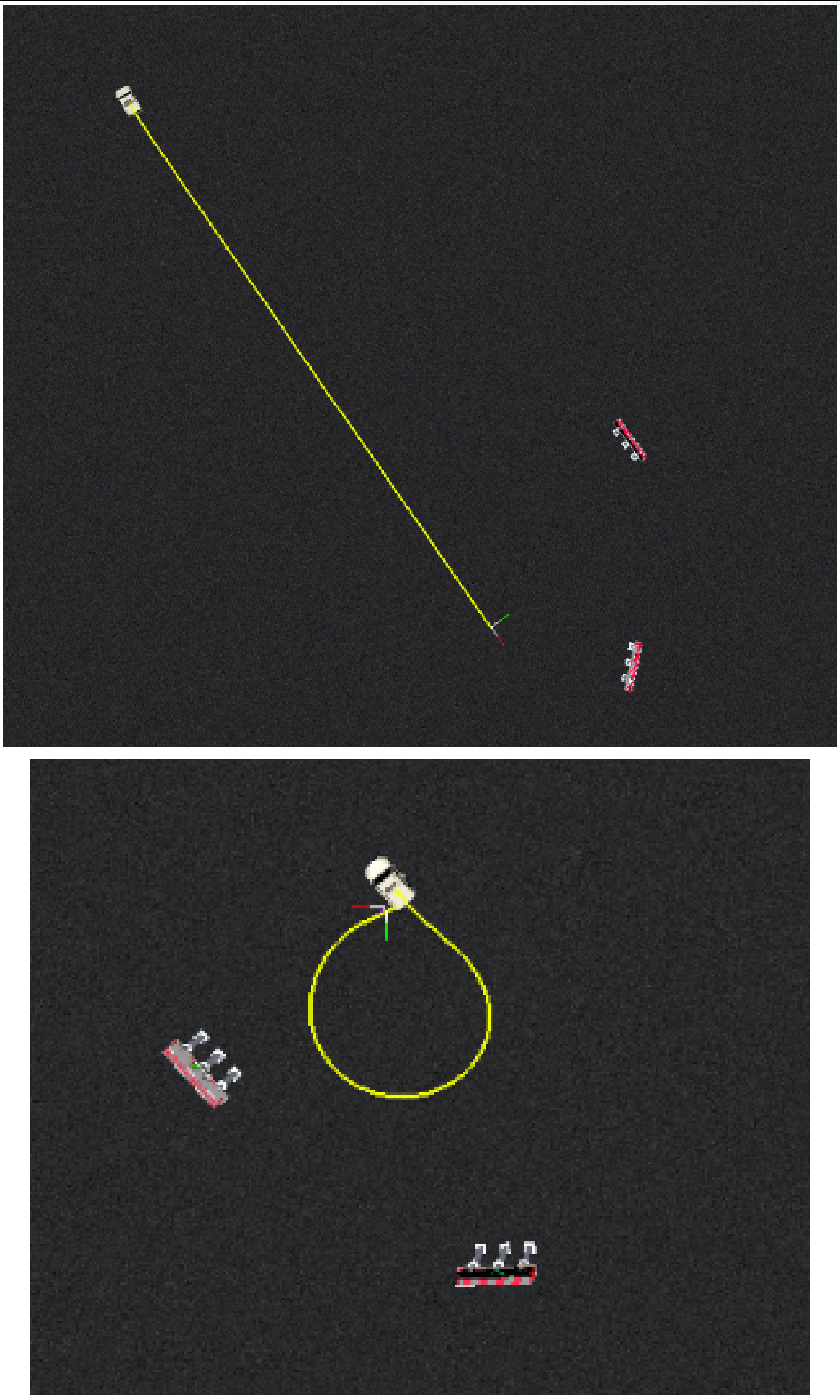
Open loop controller-built tracks

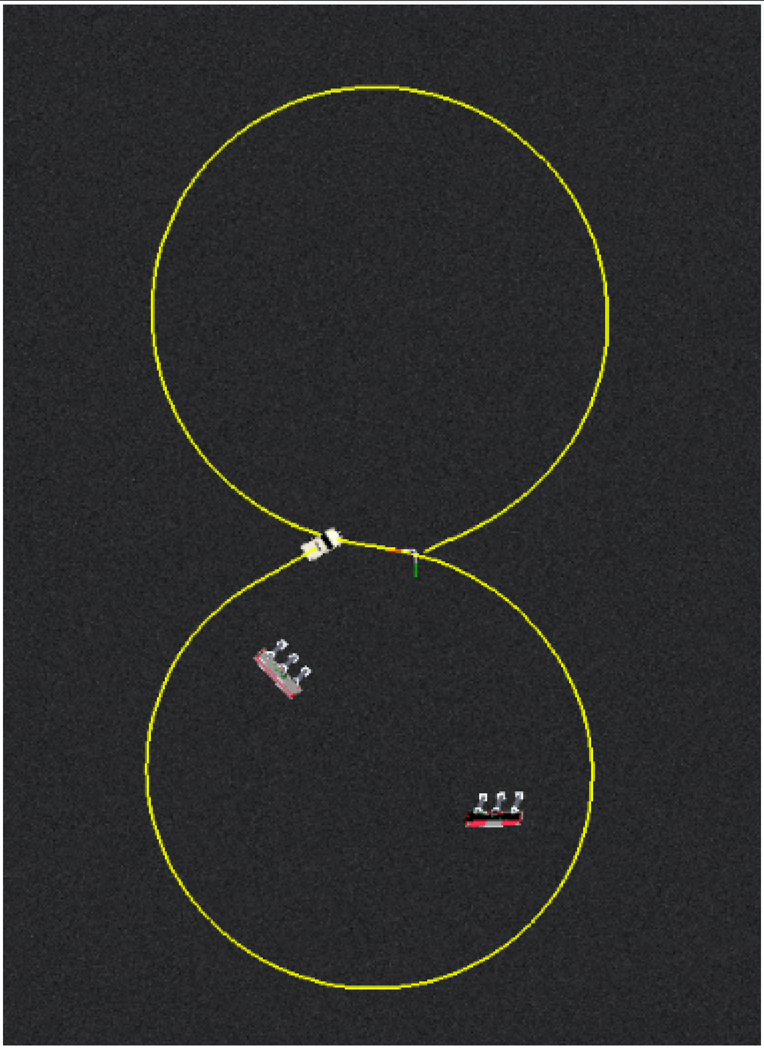


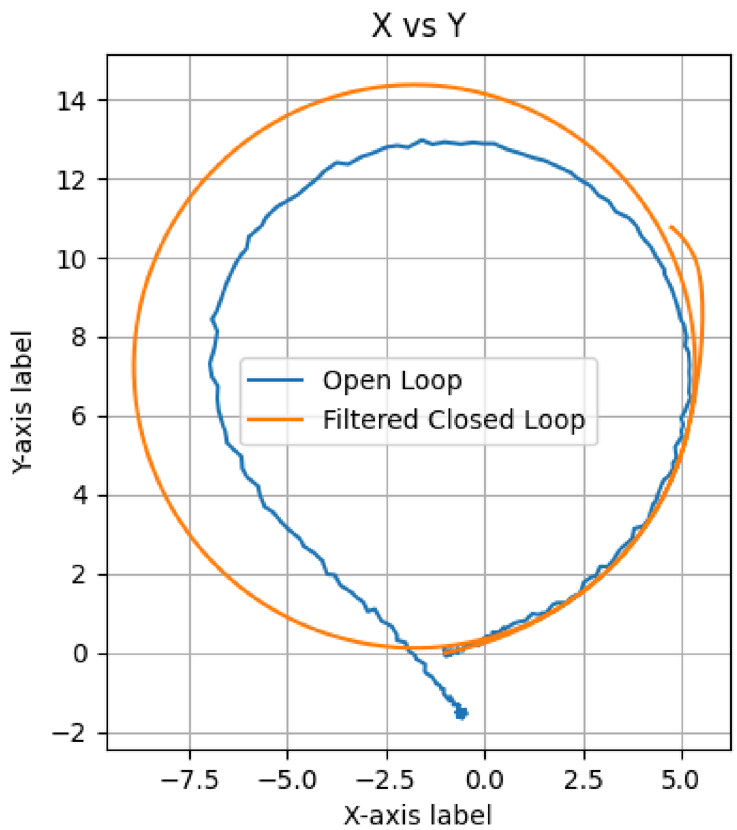

Cuboid

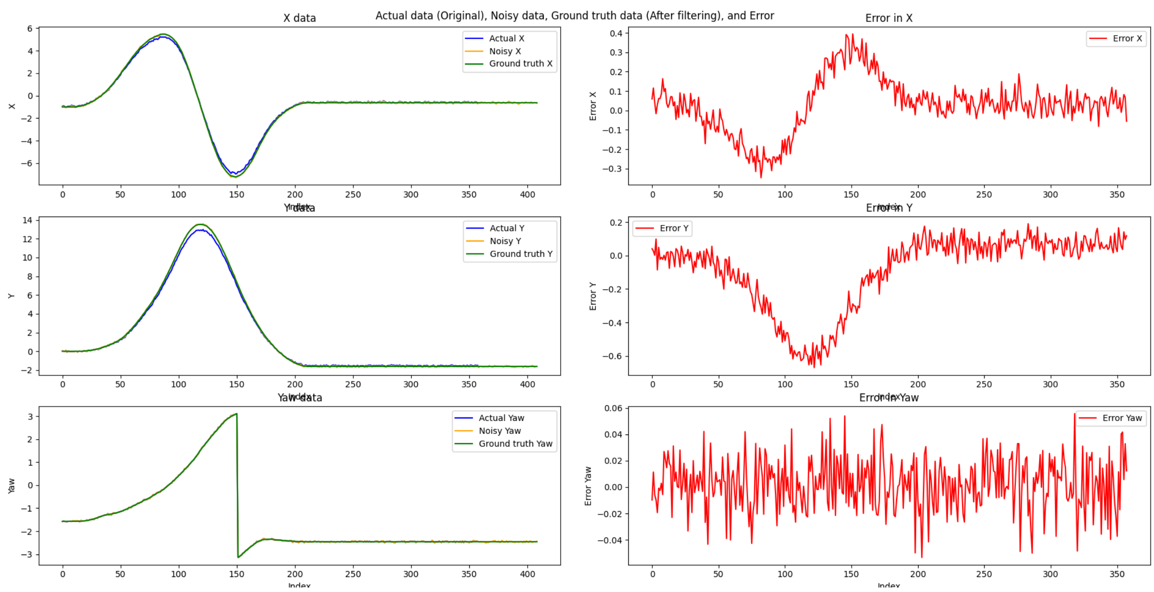


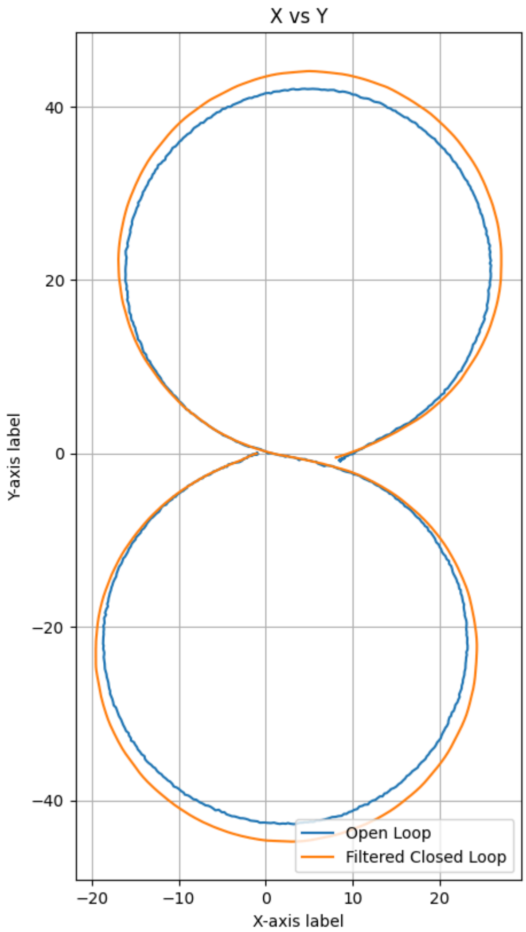
Closed loop controller-built tracks

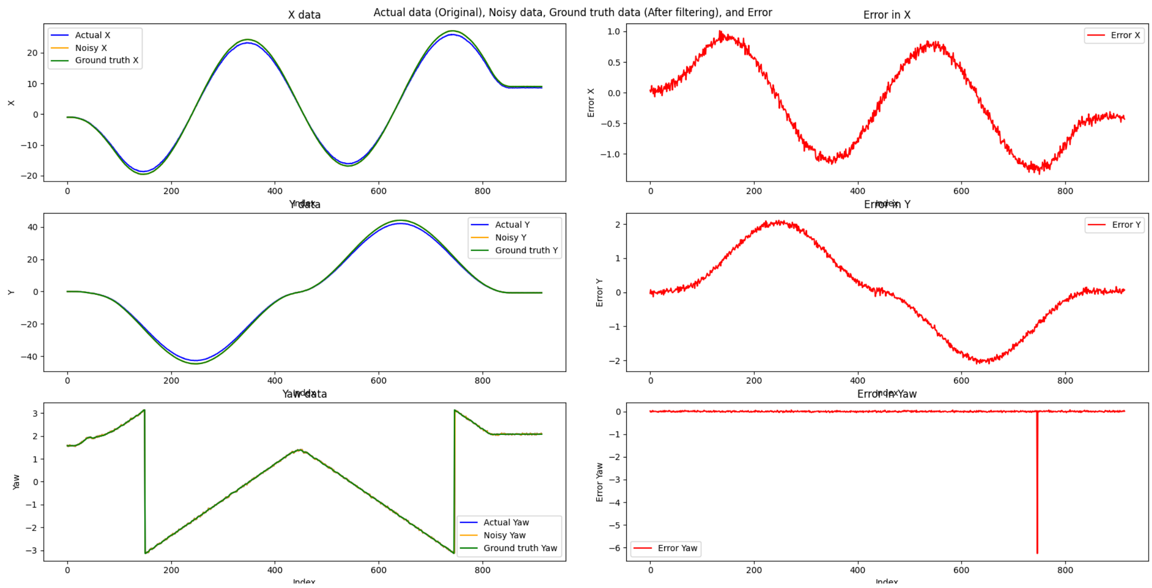


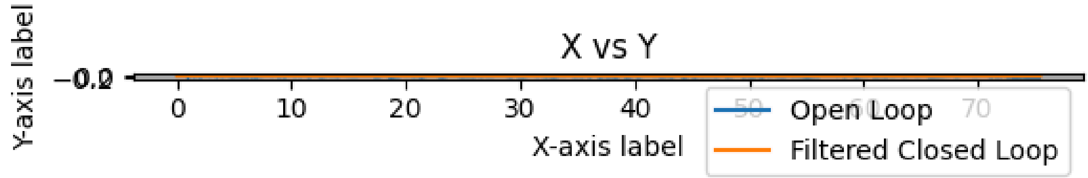


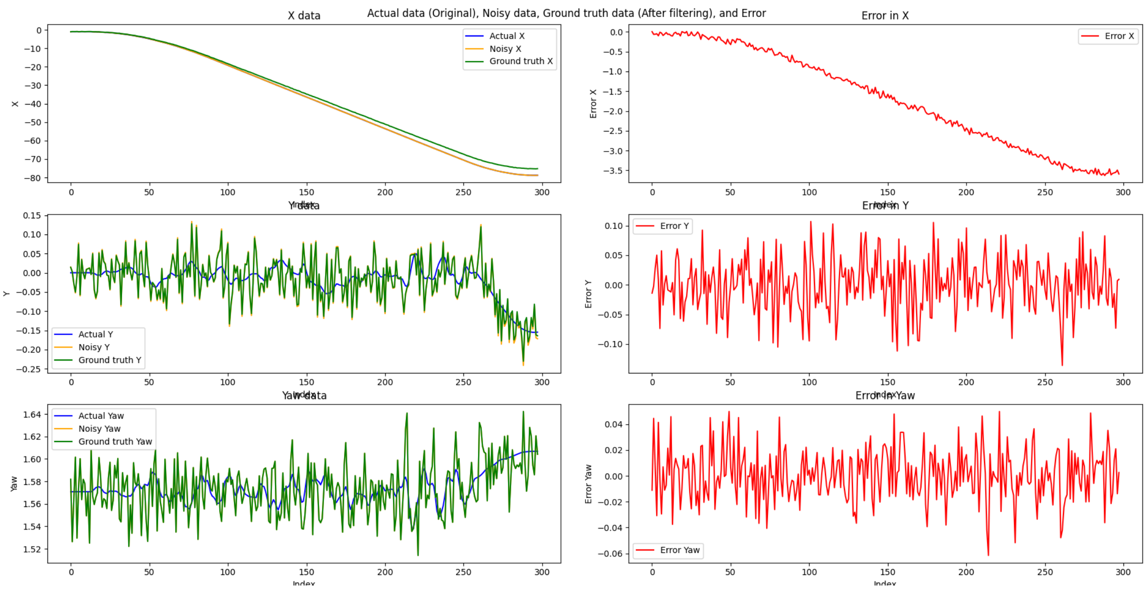

X vs Y
Open Loop
Filtered Closed Loop
Y-axis label
X-axis label

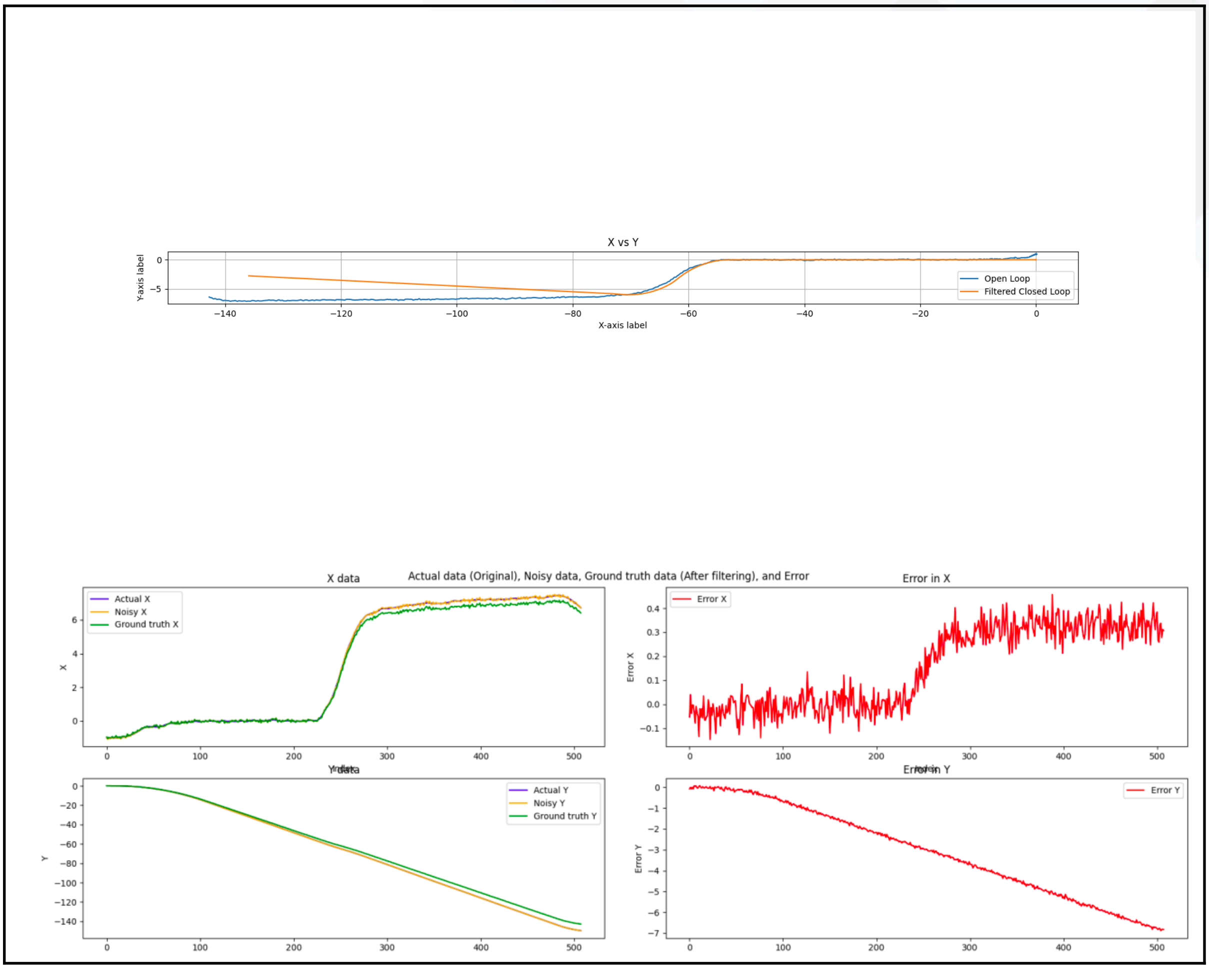

Actual data (Original), Noisy data, Ground truth data (After filtering), and Error
X data
Actual X
Noisy X
Ground truth X
Error in X
Error X
Y data
Actual Y
Noisy Y
Ground truth Y
Error in Y
Error Y
Yaw data
Actual Yaw
Noisy Yaw
Ground truth Yaw
Error in Yaw
Error Yaw


X vs Y
Y-axis label
X-axis label
Open Loop
Filtered Closed Loop


Actual data (Original), Noisy data, Ground truth data (After filtering), and Error
X data
Actual X
Noisy X
Ground truth X
Error in X
Error X
Y data
Actual Y
Noisy Y
Ground truth Y
Error in Y
Error Y
Yaw data
Actual Yaw
Noisy Yaw
Ground truth Yaw
Error in Yaw
Error Yaw


X vs Y
Y-axis label
X-axis label
Open Loop
Filtered Closed Loop


Actual data (Original), Noisy data, Ground truth data (After filtering), and Error
X data
Error in X
Y data
Error in Y
Yaw data
Error in Yaw
Actual X
Noisy X
Ground truth X
Error X
Actual Y
Noisy Y
Ground truth Y
Error Y
Actual Yaw
Noisy Yaw
Ground truth Yaw
Error Yaw



The Written Code

Stanley_Controller.py

```
import time
import math
import numpy as np
import matplotlib.pyplot as plt
import rospy
from std_msgs.msg import Float64
from nav_msgs.msg import Odometry
from csv import reader
from matplotlib import pyplot as plt
from tf.transformations import euler_from_quaternion
import Control.draw as draw
import CurvesGenerator.cubic_spline as cs
```

```
reach_flag = False

# Constants class
class C:
    # PID config
    Kp = 1.2

    # System config
    k = 0.5
    dt = 0.1
    dref = 1.2

    # vehicle config
    RF = 3.3  # [m] distance from rear to vehicle front end of
vehicle
    RB = 0.8  # [m] distance from rear to vehicle back end of
vehicle
    W = 2.4  # [m] width of vehicle
    WD = 0.7 * W  # [m] distance between left-right wheels
    WB = 2.5  # [m] Wheel base
    TR = 0.44  # [m] Tyre radius
    TW = 0.7  # [m] Tyre width
    MAX_STEER = 0.65


class OdometryHandler:
    """
    Class to handle odometry data.
    """
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        self.v = 0.0

        # Initialize the subscriber to receive odometry messages
        rospy.Subscriber('/odom', Odometry,
self.log_callback_odom)
        # Initialize the subscriber to receive odometry messages
        # rospy.Subscriber('/odom', Odometry,
```

```
self.log_callback_odom_test)

    def log_callback_odom(self, msg):
        """
        Callback function for odometry messages.
        """
        # Extract position from odometry
        position = msg.pose.pose.position
        self.x = position.y
        self.y = -position.x

        # Extract orientation from odometry and compute yaw
        orientation = msg.pose.pose.orientation
        orientation_list = [orientation.x, orientation.y,
orientation.z, orientation.w]
        (roll, pitch, yaw) = euler_from_quaternion
(orientation_list)
        # siny_cosp = 2 * (orientation.w * orientation.z +
orientation.x * orientation.y)
        # cosy_cosp = 1 - 2 * (orientation.y * orientation.y +
orientation.z * orientation.z)
        self.yaw = yaw
        # print("yaw: ", self.yaw)
        # Extract linear velocity from odometry and compute speed
        twist = msg.twist.twist.linear
        self.v = math.hypot(twist.x, twist.y)


class Node:
    """
    Node class representing a vehicle.
    """
    def __init__(self, odom_handler, x=0.0, y=0.0, yaw=0.0,
v=0.0):
        self.x = x
        self.y = y
        self.yaw = yaw
        self.v = v
        self.counter = 0

        # Store a reference to the odometry handler object
```

```
        self.odom_handler = odom_handler

        # Initialize ROS node
        rospy.init_node('car_controller', anonymous=True)

        # ROS publishers for steering angle and cmd_vel (gas pedal
position)
        self.steering_pub = rospy.Publisher('/SteeringAngle',
Float64, queue_size=10)
        self.cmd_vel_pub = rospy.Publisher('/cmd_vel', Float64,
queue_size=10)
        self.brakes = rospy.Publisher('/brakes', Float64,
queue_size=10)
        # Initialize the reference path flag
        self.reference_set = False

    def update(self, a, delta):
        """
        Update function that publishes the steering angle and gas
pedal position.
        """
        # Convert delta from radians to degrees
        steering_angle_deg = delta * 180 / math.pi
        # Publish the steering angle
        self.steering_pub.publish(Float64(steering_angle_deg))

        # Clamp acceleration between 0 and 1 for the gas pedal
        gas_pedal = np.clip(a, 0, 1)
        # Publish the gas pedal position
        self.cmd_vel_pub.publish(gas_pedal + 0.2)

    def run(self, ref_path, target_speed):
        """
        Run the control loop for the node.
        """
        # Initialize variables
        t = 0.0
        max_time = 100
        start_time = time.time()
        current_time = time.time()
        print("Start time: ", start_time)
```

```
        xrec, yrec, yawrec = [], [], []
        c = 0

        # Control loop
        while current_time - start_time < max_time:
            current_time = time.time()
            # Get the latest odometry data from the odometry
handler
            self.x = self.odom_handler.x
            self.y = self.odom_handler.y
            self.yaw = self.odom_handler.yaw
            self.v = self.odom_handler.v

            # Calculate feedback control
            delta, target_index =
front_wheel_feedback_control(self, ref_path)

            # Calculate the distance to the end of the path
            dist = math.hypot(self.x - ref_path.cx[-1], self.y -
ref_path.cy[-1])

            # Calculate acceleration using PID control
            ai = pid_control(target_speed, self.v, dist)

            # Update the node's control inputs
            self.update(ai, delta)

            # Update time
            t += C.dt

            # Check if the vehicle is close to the end of the path
            if dist <= C.dref:
                self.brakes.publish(Float64(0.05))
                self.cmd_vel_pub.publish(Float64(0))
                break

            # Save the trajectory
            xrec.append(self.x)
            yrec.append(self.y)
            yawrec.append(self.yaw)
```

```
class Trajectory:
    def __init__(self, cx, cy, cyaw):
        self.cx = cx
        self.cy = cy
        self.cyaw = cyaw
        self.ind_old = 0

    def calc_theta_e_and_ef(self, node):
        """
        Calculate theta_e (heading error) and ef (lateral error)
given the node's current state.
        """
        # Calculate front axle coordinates
        fx = node.x + C.WB * math.cos(node.yaw)
        fy = node.y + C.WB * math.sin(node.yaw)

        # Calculate the differences between the front axle and the
reference path
        dx = [fx - x for x in self.cx]
        dy = [fy - y for y in self.cy]

        # Find the closest point on the reference path
        target_index = np.argmin(np.hypot(dx, dy))
        target_index = max(self.ind_old, target_index)
        self.ind_old = max(self.ind_old, target_index)

        # Calculate lateral error
        front_axle_vec_rot_90 = np.array([[math.cos(node.yaw -
math.pi / 2.0)],
                                          [math.sin(node.yaw -
math.pi / 2.0)]])
        vec_target_2_front = np.array([[dx[target_index]],
                                       [dy[target_index]]])
        ef = np.dot(vec_target_2_front.T, front_axle_vec_rot_90)

        # Calculate heading error
        theta = node.yaw
        theta_p = self.cyaw[target_index]
        theta_e = pi_2_pi(theta_p - theta)
```

```python
        return theta_e, ef, target_index

def front_wheel_feedback_control(node, ref_path):
    """
    Front wheel feedback control algorithm.
    """
    # Calculate heading error and lateral error
    theta_e, ef, target_index = ref_path.calc_theta_e_and_ef(node)

    # Calculate steering angle
    delta = theta_e + math.atan2(C.k * ef, node.v)
    # print("Delta: ", delta)

    return delta, target_index

def pi_2_pi(angle):
    """
    Normalize an angle to the range [-pi, pi].
    """
    if angle > math.pi:
        return angle - 2.0 * math.pi
    if angle < -math.pi:
        return angle + 2.0 * math.pi

    return angle

def pid_control(target_v, v, dist):
    """
    PID control algorithm for speed control.
    """
    a = 0.3 * (target_v - v)

    if dist < 10.0:
        if v > 3.0:
            a = -2.5
        elif v < -2.0:
            a = -1.0

    return a

def main():
```

```python
    # Load reference path from CSV file
    with open('bigger_infinity_shape_path.csv', newline='') as f:
        rows = list(reader(f, delimiter=','))

    # Create an odometry handler object
    odom_handler = OdometryHandler()

    # Parse the path from the CSV file
    ax, ay = [[float(i) for i in row] for row in zip(*rows[1:])]
    ax[0] = odom_handler.x
    ay[0] = odom_handler.y
    # Generate the reference path
    cx, cy, cyaw, _, _ = cs.calc_spline_course(ax, ay, ds=C.dt)
    cyaw[0] = odom_handler.yaw
    ref_path = Trajectory(cx, cy, cyaw)

    # Initialize the node with the odometry handler
    node = Node(odom_handler, x=cx[0], y=cy[0], yaw=cyaw[0],
v=0.0)

    # Run the control loop
    node.run(ref_path, target_speed= 25.0 / 3.6)  # Target speed
in m/s

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

Odom_Handler.py

```python
#!/usr/bin/env python
import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point, Quaternion, Twist
from std_msgs.msg import Header
import numpy as np
from pykalman import KalmanFilter
from tf.transformations import euler_from_quaternion
import csv
import os
```

```python
class OdometryHandler:
    """
    Class to handle odometry data, add Gaussian noise, filter it
using Kalman filters, and log the data.
    """
    def __init__(self):
        # Initialize state variables
        self.state = {
            "position_x": 0.0,
            "position_y": 0.0,
            "orientation_x": 0.0,
            "orientation_y": 0.0,
            "orientation_z": 0.0,
            "orientation_w": 0.0,
            "linear_x": 0.0,
            "linear_y": 0.0,
            "linear_z": 0.0
        }

        # Initialize subscriber for odometry messages
        rospy.Subscriber('/odom', Odometry,
self.log_callback_odom)

        # Initialize publisher for filtered odometry data
        self.filtered_pub = rospy.Publisher('/filtered_odom',
Odometry, queue_size=10)

        # Initialize Kalman filters for position, orientation, and
linear velocity
        self.init_kalman_filters()

        # Define CSV file paths
        self.csv_before_path = 'before_anything.csv'
        self.csv_noise_path = 'after_noise_adding.csv'
        self.csv_filtered_path = 'after_filtering.csv'

        # Write headers if files are empty
        self.write_csv_headers()
```

```python
    def init_kalman_filters(self):
        # Initialize Kalman filters for position, orientation, and
linear velocity
        self.position_filter = KalmanFilter(
            transition_matrices=np.array([[1, 0, 1, 0],
                                          [0, 1, 0, 1],
                                          [0, 0, 1, 0],
                                          [0, 0, 0, 1]]),
            observation_matrices=np.array([[1, 0, 0, 0],
                                           [0, 1, 0, 0]]),
            transition_covariance=np.eye(4) * 0.1,
            observation_covariance=np.eye(2) * 0.1,
            initial_state_mean=np.array([self.state["position_x"],
self.state["position_y"], 0, 0]),
            initial_state_covariance=np.eye(4) * 1.0
        )

        self.orientation_filter = KalmanFilter(
            transition_matrices=np.eye(4),
            observation_matrices=np.eye(4),
            transition_covariance=np.eye(4) * 0.1,
            observation_covariance=np.eye(4) * 0.1,

initial_state_mean=np.array([self.state["orientation_x"],
self.state["orientation_y"], self.state["orientation_z"],
self.state["orientation_w"]]),
            initial_state_covariance=np.eye(4) * 1.0
        )

        self.velocity_filter = KalmanFilter(
            transition_matrices=np.eye(3),
            observation_matrices=np.eye(3),
            transition_covariance=np.eye(3) * 0.1,
            observation_covariance=np.eye(3) * 0.1,
            initial_state_mean=np.array([self.state["linear_x"],
self.state["linear_y"], self.state["linear_z"]]),
            initial_state_covariance=np.eye(3) * 1.0
        )

    def write_csv_headers(self):
        # Write headers to CSV files if they are empty
```

```
        if not os.path.exists(self.csv_before_path):
            with open(self.csv_before_path, 'w', newline='') as f:
                csv.writer(f).writerow(['timestamp', 'x', 'y',
'yaw'])
        if not os.path.exists(self.csv_noise_path):
            with open(self.csv_noise_path, 'w', newline='') as f:
                csv.writer(f).writerow(['timestamp', 'x', 'y',
'yaw'])
        if not os.path.exists(self.csv_filtered_path):
            with open(self.csv_filtered_path, 'w', newline='') as
f:
                csv.writer(f).writerow(['timestamp', 'x', 'y',
'yaw'])

    def filter_position_data(self, position_data, timestamp):
        """
        Filter position data using Kalman filter and log data.
        """
        # Filter position data using Kalman filter
        state_mean, state_covariance =
self.position_filter.filter_update(
            self.position_filter.initial_state_mean,
            self.position_filter.initial_state_covariance,
            position_data  # Pass noisy data directly
        )

        # Update state variables with filtered data
        self.state["position_x"], self.state["position_y"] =
state_mean[:2]


    def filter_orientation_data(self, orientation_data,
timestamp):
        """
        Filter orientation data using Kalman filter and log data.
        """
        # Filter orientation data using Kalman filter
        state_mean, state_covariance =
self.orientation_filter.filter_update(
            self.orientation_filter.initial_state_mean,
            self.orientation_filter.initial_state_covariance,
```

```
            orientation_data  # Pass noisy data directly
        )

        # Update state variables with filtered data
        self.state["orientation_x"], self.state["orientation_y"],
self.state["orientation_z"], self.state["orientation_w"] =
state_mean

        # Calculate yaw from filtered orientation data
        quaternion_filtered = [self.state["orientation_x"],
self.state["orientation_y"], self.state["orientation_z"],
self.state["orientation_w"]]
        yaw_after = euler_from_quaternion(quaternion_filtered)[2]

        # Log filtered orientation data to CSV
        with open(self.csv_filtered_path, 'a', newline='') as f:
            csv.writer(f).writerow([timestamp,
self.state["position_x"], self.state["position_y"], yaw_after])

    def filter_velocity_data(self, velocity_data):
        """
        Filter linear velocity data using Kalman filter.
        """
        # Filter velocity data using Kalman filter
        state_mean, state_covariance =
self.velocity_filter.filter_update(
            self.velocity_filter.initial_state_mean,
            self.velocity_filter.initial_state_covariance,
            velocity_data  # Pass noisy data directly
        )

        # Update state variables with filtered data
        self.state["linear_x"], self.state["linear_y"],
self.state["linear_z"] = state_mean

    def publish_filtered_odom(self, timestamp):
        """
        Publish filtered odometry data.
        """
        # Create a new Odometry message for the filtered data
        filtered_odom_msg = Odometry()
```

```
        filtered_odom_msg.header.stamp = rospy.Time.now()
        filtered_odom_msg.header.frame_id = 'odom'

        # Set the filtered position
        filtered_odom_msg.pose.pose.position.x =
self.state["position_x"]
        filtered_odom_msg.pose.pose.position.y =
self.state["position_y"]

        # Set the filtered orientation
        filtered_odom_msg.pose.pose.orientation.x =
self.state["orientation_x"]
        filtered_odom_msg.pose.pose.orientation.y =
self.state["orientation_y"]
        filtered_odom_msg.pose.pose.orientation.z =
self.state["orientation_z"]
        filtered_odom_msg.pose.pose.orientation.w =
self.state["orientation_w"]

        # Set the filtered linear velocity
        filtered_odom_msg.twist.twist.linear.x =
self.state["linear_x"]
        filtered_odom_msg.twist.twist.linear.y =
self.state["linear_y"]
        filtered_odom_msg.twist.twist.linear.z =
self.state["linear_z"]

        # Publish filtered odometry data
        self.filtered_pub.publish(filtered_odom_msg)

    def close_csv_files(self):
        # No need to close files, as we use context managers with
'with'
        pass

    def log_callback_odom(self, msg):
        """
        Callback function for odometry messages that injects
Gaussian noise into read data.
        """
        # Extract position, orientation, and velocity data from
```

```
the incoming message
        position = msg.pose.pose.position
        orientation = msg.pose.pose.orientation
        linear_velocity = msg.twist.twist.linear
        # Log original data to CSV

        # Compute yaw from orientation data (quaternion) with
noise
        quaternion = [orientation.x, orientation.y, orientation.z,
orientation.w]
        yaw = euler_from_quaternion(quaternion)[2]

        timestamp = msg.header.stamp.to_sec()
        with open(self.csv_before_path, 'a', newline='') as f:
            csv.writer(f).writerow([timestamp, position.x,
position.y, yaw])

        # Parameters for Gaussian noise
        pos_noise_std_dev = 0.05  # Standard deviation for
position noise
        ori_noise_std_dev = 0.01  # Standard deviation for
orientation noise
        vel_noise_std_dev = 0.03  # Standard deviation for
velocity noise

        # Adding Gaussian noise to the data
        noisy_position_x = position.x + np.random.normal(0,
pos_noise_std_dev)
        noisy_position_y = position.y + np.random.normal(0,
pos_noise_std_dev)
        noisy_orientation_x = orientation.x + np.random.normal(0,
ori_noise_std_dev)
        noisy_orientation_y = orientation.y + np.random.normal(0,
ori_noise_std_dev)
        noisy_orientation_z = orientation.z + np.random.normal(0,
ori_noise_std_dev)
        noisy_orientation_w = orientation.w + np.random.normal(0,
ori_noise_std_dev)
        noisy_linear_velocity_x = linear_velocity.x +
np.random.normal(0, vel_noise_std_dev)
        noisy_linear_velocity_y = linear_velocity.y +
```

```
np.random.normal(0, vel_noise_std_dev)
        noisy_linear_velocity_z = linear_velocity.z +
np.random.normal(0, vel_noise_std_dev)

        # Compute yaw from orientation data (quaternion) with
noise
        noisy_quaternion = [noisy_orientation_x,
noisy_orientation_y, noisy_orientation_z, noisy_orientation_w]
        yaw_before = euler_from_quaternion(noisy_quaternion)[2]

        # Log noised orientation data to CSV
        with open(self.csv_noise_path, 'a', newline='') as f:
            csv.writer(f).writerow([timestamp, noisy_position_x,
noisy_position_y, yaw_before])

        # Process and filter noisy data
        self.filter_position_data(np.array([noisy_position_x,
noisy_position_y]), timestamp)

self.filter_orientation_data(np.array([noisy_orientation_x,
noisy_orientation_y, noisy_orientation_z, noisy_orientation_w]),
timestamp)

self.filter_velocity_data(np.array([noisy_linear_velocity_x,
noisy_linear_velocity_y, noisy_linear_velocity_z]))

        # Publish the filtered odometry data
        self.publish_filtered_odom(timestamp)

def main():
    rospy.init_node('filtered_odom_publisher', anonymous=True)

    odom_handler = OdometryHandler()

    rospy.on_shutdown(odom_handler.close_csv_files)
    rospy.spin()

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

Object_Detection.py

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

import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2
import cvlib as cv
from cvlib.object_detection import draw_bbox
import numpy as np

class ImageSubscriber:
    def __init__(self):
        rospy.init_node('object_detection_subscriber',
anonymous=True)
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/image", Image,
self.image_callback)
        self.window_name = 'Object Detection'
        self.color_green = (0, 255, 0)
        self.color_blue = (255, 0, 0)

    def image_callback(self, data):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
            self.detect_objects(cv_image)
            self.detect_cones(cv_image)
            self.draw_info(cv_image)
        except Exception as e:
            print(e)

    def detect_objects(self, cv_image):
        bbox, label, conf = cv.detect_common_objects(cv_image)

        for i in range(len(bbox)):
            text = f"{label[i]}: {conf[i]:.2f}"
            x, y, w, h = bbox[i]

            # Draw a bounding box
```

```
            cv2.rectangle(cv_image, (x, y), (w, h),
self.color_green, 2)

            # Draw a plus sign at the centroid
            centroid_x = int((x + w) / 2)
            centroid_y = int((y + h) / 2)
            plus_size = 10
            cv2.line(cv_image, (centroid_x - plus_size,
centroid_y), (centroid_x + plus_size, centroid_y),
self.color_green, 2)
            cv2.line(cv_image, (centroid_x, centroid_y -
plus_size), (centroid_x, centroid_y + plus_size),
self.color_green, 2)

            # Add text with the label and confidence
            cv2.putText(cv_image, text, (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, self.color_green, 2)

    def detect_cones(self, cv_image):
        hsv_image = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
        lower_orange = np.array([10, 100, 100])
        upper_orange = np.array([30, 255, 255])
        mask = cv2.inRange(hsv_image, lower_orange, upper_orange)
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        for contour in contours:
            # Calculate contour area and bounding rectangle
            area = cv2.contourArea(contour)
            x, y, w, h = cv2.boundingRect(contour)
            confidence = area / (w * h)

            # Check if contour area is reasonable for a cone
            if area > 1000:
                # Draw bounding rectangle and centroid marker
                cv2.rectangle(cv_image, (x, y), (x + w, y + h),
self.color_blue, 2)
                centroid_x = x + w // 2
                centroid_y = y + h // 2
                cv2.line(cv_image, (centroid_x - 10, centroid_y),
(centroid_x + 10, centroid_y), self.color_blue, 2)
```

```
                cv2.line(cv_image, (centroid_x, centroid_y - 10),
(centroid_x, centroid_y + 10), self.color_blue, 2)

                # Add confidence as text
                cv2.putText(cv_image, f"Cone: {confidence:.2f}",
(x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, self.color_blue, 2)

    def draw_info(self, image):
        cv2.imshow(self.window_name, image)
        cv2.waitKey(1)

    def run(self):
        rospy.spin()

if __name__ == '__main__':
    try:
        image_subscriber = ImageSubscriber()
        image_subscriber.run()
    except rospy.ROSInterruptException:
        pass
```

Conclusion

In conclusion, we've come a long way in making a system that can drive itself on tracks. We faced some tough problems, but we worked hard and made big progress.

Our main goal was to make sure our self-driving system could handle all the twists and turns of a track. We've done a lot of testing and thinking to get where we are now.

Looking ahead, we'll keep improving our system based on what we've learned. Every step forward brings us closer to our goal of a smart, safe, and smooth-driving system.