EVER 2024 Autonomous Track

Institution /Team Identification……………………………………………………………………

System Overview

Describe the integrated systems and how each module contributes to handling the specific scenarios.

Integrated Systems:

- LiDAR: Measures distances to objects and obstacles.
- Camera: Detects humans and other relevant objects.
- Steering: Controls the direction of the vehicle.
- Odometry: Tracks the vehicle's position and movement.
- Velocity Control: Manages the speed of the vehicle.

Scenario Handling:

1. **General Operation:**
   - As the vehicle moves, the LiDAR continuously measures distances to surrounding objects.
   - The camera detects humans and other obstacles.
   - When a human is detected, the LiDAR provides the distance to the individual. The emergency stop algorithm calculates the necessary stopping time, and the braking system is activated accordingly.
   - If a cone or another vehicle is detected, the LiDAR measures the distance, and the emergency stop algorithm calculates the time required to either change direction or switch lanes.
2. **Path 1: Straight Line:**
   - The vehicle moves in a straight line.
   - When the camera detects a human, the LiDAR measures the distance. The vehicle performs an emergency stop, halting until the pedestrian crosses the street.
3. **Path 2: Straight Line with Lane Change:**
   - The vehicle moves in a straight line.
   - Upon detecting a human, the LiDAR measures the distance, and the vehicle performs an emergency stop until the pedestrian crosses the street.
   - When detecting a cone or another vehicle, the LiDAR provides the distance, and the vehicle changes lanes as needed.
4. **Path 3: Circular Path:**
   - The vehicle moves in a circular path.

- When the camera detects a human, the LiDAR measures the distance. The vehicle performs an emergency stop, halting until the pedestrian crosses the street.
- When the camera detects a cone or car the car moves in the next circle path.

5. **Custom Track:**
   - The vehicle moves in a straight line.
   - When the camera detects a human, the LiDAR measures the distance. The vehicle performs an emergency stop, halting until the pedestrian crosses the street.
   - When the camera detects a cone or car the car moves in the next line
   - Then the vehicle moves in a circular path.
   - When the camera detects a human, the LiDAR measures the distance. The vehicle performs an emergency stop, halting until the pedestrian crosses the street.
   - When the camera detects a cone or car the car moves in the next circle path.

Methodology Used

Here you should mention clearly the techniques you used to achieve the results you got and plan the trajectories of your motion.
You must write in a full detailed manners ex: (mention any equations you used, tools, methods, libraries, packages, etc.)

- Pure Pursuit Algorithm

The Pure Pursuit algorithm is used to generate the target points for the vehicle to follow. The control equation used for the curvature (κ) is derived from the geometry of the look-ahead point:
where: $k = \frac{2y}{L^2}$

- y is the lateral distance to the look-ahead point,
- L is the look-ahead distance.

- Time to Collision (TTC)

The Time to Collision (TTC) is calculated using the relative velocity and distance:

$$TTC = \frac{Distance}{Relative\ Velocity}$$

- **Tools and Packages**

Simulation Tools:

- RViz
- CoppeliaSim

ROS Packages:

- pc2
- cv_bridge
- std_msgs
- nav_msgs
- sensor_msgs
- geometry_msgs

Programming Libraries:

- numpy
- math
- time
- opencv

Machine Learning Libraries:

- YOLO

Multithreading and Data Handling:

- Thread
- csv

Equations

Path Equations:

- Circular
- Polynomial
- Straight Line

Control Equations:

- Pure Pursuit (Curvature Equation)

General Workflow

1. **Noise Application and Filtering:** Introduce noise to the sensor data and apply appropriate filtering techniques.
2. **System Initialization:** Ensure all systems, including the vehicle, are operational at the start of the track.
3. **Algorithm Implementation:** Implement Pure Pursuit algorithms for vehicle guidance.
4. **Look-Ahead Point Calculation:** Use algorithms to calculate look-ahead points and determine the vehicle's path.
5. **Object Detection:** Detect objects using sensors.
6. **Distance Measurement:** Measure the distance to detected objects.
7. **Object Classification:** Identify the type of detected objects.
8. **Emergency Stop:** Apply emergency stop algorithms if a human is detected.
9. **Lane Change:** Execute lane changes when necessary.

10. **Track Completion:** Ensure the vehicle completes the track and stops at the end.

Specific Path Implementations

Path 1: Straight Line

- For straight-line movement, the Pure Pursuit algorithm generates target points. Using the straight-line equation, the vehicle moves towards the next point, adjusting its path with the look-ahead parameter.

Path 2: Straight Line with Lane Change

- The vehicle moves in a straight line, utilizing its camera to detect humans and LiDAR to measure distance. Upon detecting a human, the vehicle performs an emergency stop until the pedestrian crosses. For other obstacles like cones or vehicles, a lane change is triggered. If the vehicle is in the right lane, it moves to the left and vice versa.

Path 3: Circular Path

- For circular paths, coordinates are converted from Cartesian (x, y) to polar form (r, θ). This conversion simplifies navigation by using appropriate circular equations.

Custom Track

The custom track is divided into two parts: a straight path and a circular path. For the straight section, we use Cartesian coordinates (x, y) and straight-line equations. For the circular section, we use polar coordinates (r, θ) and circular equations. This approach ensures smooth navigation through both types of paths.

Tracks Description

Detail the implementation and challenges of each re-implemented track and the innovative track that you build on your own.

Initial Challenges:

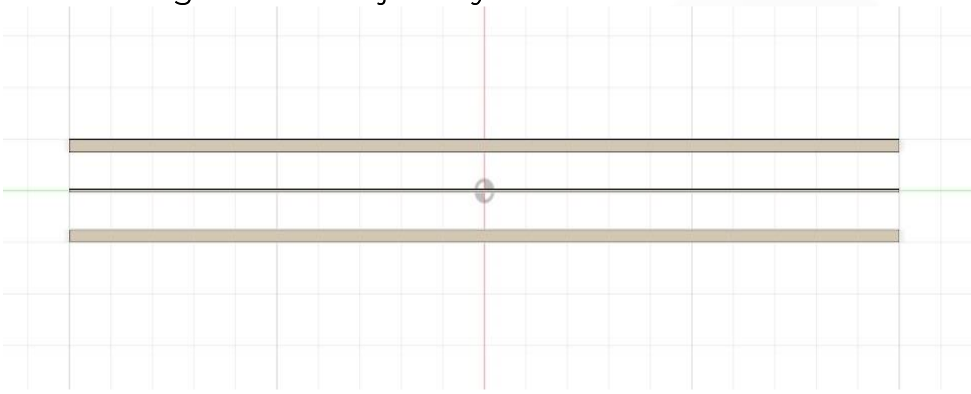
The primary challenge involved familiarizing ourselves with CoppeliaSim. This included defining physical dimensions, modifying object geometries, configuring vehicle parameters, and ensuring accurate movement in both circular paths and custom tracks.

Path Design:

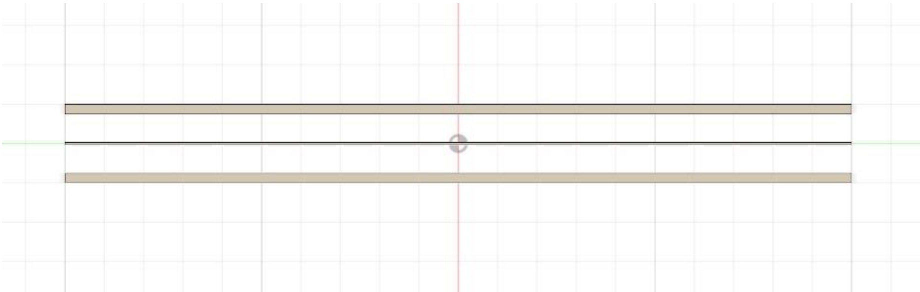
We designed our paths using Fusion 360, converting them into objects that could be imported and utilized in CoppeliaSim for simulation.

Track Implementations:

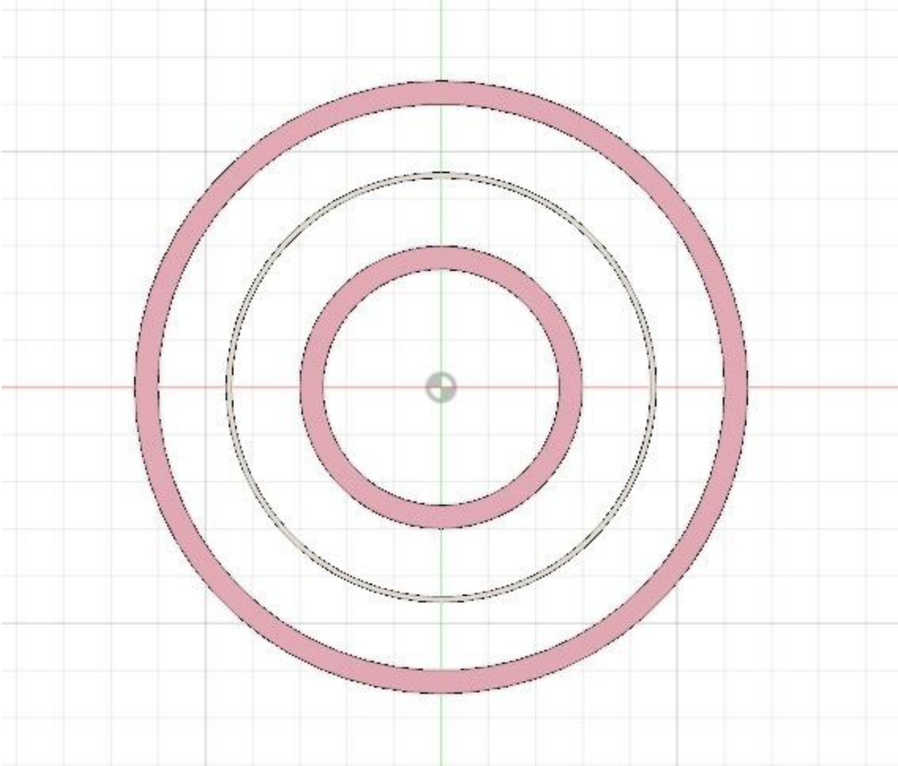
1. Path 1: Straight Line:
   - Implementation: The vehicle is programmed to move in a straight line. Waypoints are iteratively selected to guide the vehicle, ensuring smooth and consistent movement.
   - Challenges: Ensuring accurate waypoint selection and maintaining a stable trajectory.



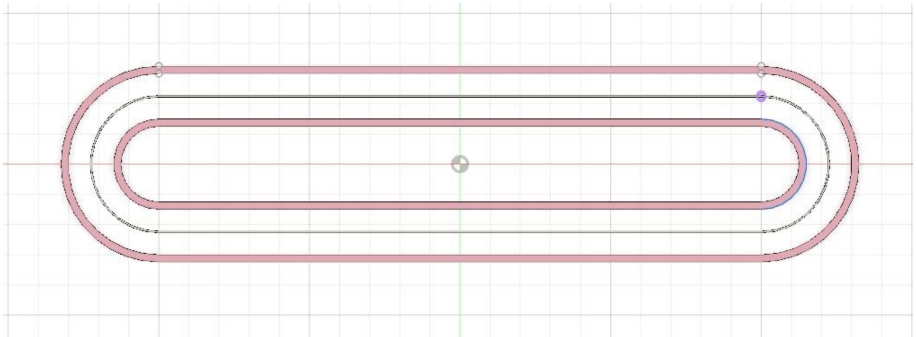
2. Path 2: Straight Line with Lane Changes:
   - Implementation: The vehicle moves in a straight line but includes algorithms to detect obstacles like slow vehicles or cones. Upon detection, the vehicle executes a lane change.
   - Challenges: Developing a robust algorithm to decide when to change lanes and ensuring smooth and safe transitions between lanes.



3. Path 3: Circular Path:
   - Implementation: The vehicle navigates a circular path by converting Cartesian coordinates to polar form and selecting waypoints with higher theta values.
   - Challenges: Accurately transforming coordinates and ensuring the vehicle follows the circular trajectory without deviation.



4. Custom Track:
    - **Implementation**: The custom track was designed with unique features and challenges, incorporating both straight and curved segments.
    - **Challenges**: Tailoring the control algorithms to handle diverse track features and ensuring seamless integration of all vehicle systems.



This path is from our university street

This structured approach facilitated the successful implementation of each track, overcoming initial challenges and leveraging innovative solutions to achieve precise and reliable vehicle navigation.

Algorithm Description

- **Initial Decision:**

We opted to utilize the Pure Pursuit control algorithm for each track, as it effectively handles both straight lines and smooth curves. The primary challenge across all paths was selecting the appropriate waypoints for implementing Pure Pursuit.

- **Path 1: Straight Line**

For straight-line paths, selecting the waypoint was straightforward. We iterated through the waypoints and chose the nearest one to the look-ahead point, ensuring smooth navigation.

Node Initialization:

- Initialize the ROS node.
- Set up publishers for velocity, steering, and braking.
- Set up subscribers for object detection and lidar points.

Straight Line Control:

- Publish the initial velocity and steering angle.
- Continuously check the TTC and the lateral distance (xxx).
- If TTC is below 14 seconds and the lateral distance is within 3 meters, stop the vehicle.
- Otherwise, maintain a low velocity to move forward.

Object Management:

- Subscribe to object detection messages.
- Adjust the vehicle behaviour based on detected objects.

Lidar Data Management:

- Subscribe to lidar point messages.
- Extract the relevant data (x, y, z coordinates, distance, relative velocity, and TTC).
- Update the TTC and lateral distance (x) variables.

- **Path 2: Straight Line with Lane Changes**

For this path, we developed an algorithm to determine whether to continue on the same track or change lanes when encountering slow vehicles or cones. We searched for waypoints that were further ahead than the current position plus the look-ahead distance but not excessively beyond it. This enabled efficient lane changes while maintaining smooth motion.

Node Initialization:

- Initialize the ROS node.
- Set up subscribers for odometry, object detection, and lidar points.
- Set up publishers for velocity, steering, and braking.

Object Management:

- Subscribe to object detection messages and update the detected object variable.
- Subscribe to lidar point messages and extract the relevant data (x, y coordinates).

Emergency Stop and Lane Change:

- If a human is detected within human_stop_distance and the lateral distance |x||x||x| is less than 3 meters, stop the vehicle.
- If a car or cone is detected within lane_change_distance and the cooldown period has elapsed, initiate a lane change.
- Update the last lane change time after a lane change.

Path Following:

- Subscribe to odometry messages and extract the current pose and orientation of the vehicle.
- Use the Pure Pursuit algorithm to follow waypoints, adjusting the steering based on the calculated curvature.

Waypoints and Paths:

- Predefined waypoints for left and right paths are used to guide the vehicle.
- The flag variable determines which path to follow (left or right).

Lidar Data Management:

- Subscribe to lidar point messages.
- Extract the relevant data (x, y, z coordinates, distance, relative velocity, and TTC).
- Update the TTC and lateral distance (x) variables.

- ### Path 3: Circular Path

For circular paths, simply choosing the largest waypoint number was insufficient. Instead, we converted the Cartesian circle coordinates to polar form. By using points with higher theta values, we ensured accurate navigation around the circular track.

Node Initialization:

- Initialize the ROS node.
- Set up subscribers for odometry, object detection, and lidar points.

- Set up publishers for velocity, steering, and braking.

Object Management:

- Subscribe to object detection messages and update the detected object variable.
- Subscribe to lidar point messages and extract the relevant data (x, y coordinates).

Emergency Stop and Lane Change:

- If a human is detected within human_stop_distance, stop the vehicle.
- If a car or cone is detected within lane_change_distance and the cooldown period has elapsed, initiate a lane change.
- Update the last lane change time after a lane change.

Path Following:

- Subscribe to odometry messages and extract the current pose and orientation of the vehicle.
- Use the Pure Pursuit algorithm to follow waypoints, adjusting the steering based on the calculated curvature.

Waypoints and Paths:

- Predefined waypoints for circular paths are used to guide the vehicle.
- The path_flag variable determines which circular path to follow (radius_1 or radius_2).

Lidar Data Management:

- Subscribe to lidar point messages.
- Extract the relevant data (x, y, z coordinates, distance, relative velocity, and TTC).
- Update the TTC and lateral distance (x) variables.

- **Custom Track**

First

For straight-line paths, selecting the waypoint was straightforward. We iterated through the waypoints and chose the nearest one to the look-ahead point, ensuring smooth navigation

Second

For circular paths, simply choosing the largest waypoint number was insufficient. Instead, we converted the Cartesian circle coordinates to polar form. By using points with higher theta values, we ensured accurate navigation around the circular track.

Node Initialization:

- Initialize the ROS node.
- Set up subscribers for odometry, object detection, and lidar points.
- Set up publishers for velocity, steering, and braking.

Object Management:

- Subscribe to object detection messages and update the detected object variable.
- Subscribe to lidar point messages and extract the relevant data (x, y coordinates).

Emergency Stop and Lane Change:

- If a human is detected within human_stop_distance, stop the vehicle.
- If a car or cone is detected within lane_change_distance and the cooldown period has elapsed, initiate a lane change.
- Update the last lane change time after a lane change.

Path Following:

- Subscribe to odometry messages and extract the current pose and orientation of the vehicle.
- Use the Pure Pursuit algorithm to follow waypoints, adjusting the steering based on the calculated curvature.

Waypoints and Paths:

- Predefined waypoints for circular paths are used to guide the vehicle.
- The path_flag variable determines which circular path to follow (radius_1 or radius_2).

Lidar Data Management:

- Subscribe to lidar point messages.
- Extract the relevant data (x, y, z coordinates, distance, relative velocity, and TTC).
- Update the TTC and lateral distance (x) variables.

- **Codes:**

1- **Detection code**

```python
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import Image as ROSImage
from std_msgs.msg import String
from cv_bridge import CvBridge
import cv2
import torch
from ultralytics import YOLO
import numpy as np
from threading import Thread

#initiate variables
detected = ''
label_publisher = rospy.Publisher('detected_labels', String, queue_size=10)
# Initialize CvBridge
bridge = CvBridge()

# Set device to GPU if available
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Load the YOLO model and specify the device
#model = YOLO("/home/kareem/catkin_ws/src/t3/scripts/best.pt").to(device)
model = YOLO("/home/kareem/catkin_ws/src/t2/scripts/yolov8n-oiv7.pt").to(device)
model1 = YOLO("/home/kareem/catkin_ws/src/t3/scripts/Cone.pt").to(device)
model2 = YOLO("/home/kareem/catkin_ws/src/t3/scripts/yolov8n.pt").to(device)
def get_centroid(x_min, y_min, x_max, y_max):
    center_x = (x_min + x_max) // 2
    center_y = (y_min + y_max) // 2
    return center_x, center_y

def detect_objects(img):
    global detected
    detected = "none"
    prev = detected

    # Resize image
    resized_img = cv2.resize(img, (640, 640))

    # Convert the image to the correct format and device
    img_tensor = torch.from_numpy(resized_img).permute(2, 0, 1).unsqueeze(0).float().to(device)
    img_tensor /= 255.0

    results = model.predict(img_tensor)
    results1 = model1.predict(img_tensor)
    results2 = model2.predict(img_tensor)
    for r in results:
        for box in r.boxes:
            label_text = model.names[int(box.cls)]  # Get the class name directly from the model's names
            confidence = box.conf.item()  # Extract the scalar confidence value
            if label_text in ["Car"]:
                x1, y1, x2, y2 = map(int, box.xyxy[0])
                # Format the label to include confidence score
                center_x, center_y = get_centroid(x1, y1, x2, y2)
                label_with_conf = f"{label_text} {confidence:.2f}"
                # Draw the bounding box and label on the resized_img directly
                cv2.rectangle(resized_img, (x1, y1), (x2, y2), (255, 0, 255), 3)
                cv2.putText(resized_img, label_with_conf, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
                cv2.circle(resized_img, (center_x, center_y), 5, (0, 255, 0), -1)
                detected = "car"

   # return resized_img
    for r in results1:
        for box in r.boxes:
            label_text = model1.names[int(box.cls)]
            confidence = box.conf.item()
            x1, y1, x2, y2 = map(int, box.xyxy[0])

            center_x, center_y = get_centroid(x1, y1, x2, y2)
            label_with_conf = f"cone {confidence:.2f}"

            cv2.rectangle(resized_img, (x1, y1), (x2, y2), (255, 0, 255), 3)
            cv2.putText(resized_img, label_with_conf, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
            cv2.circle(resized_img, (center_x, center_y), 5, (0, 255, 0), -1)
            detected = "cone"

    for r in results2:
        for box in r.boxes:
            label_text = model2.names[int(box.cls)]
            confidence = box.conf.item()
            if label_text in ["person"]:
                x1, y1, x2, y2 = map(int, box.xyxy[0])

                center_x, center_y = get_centroid(x1, y1, x2, y2)
                label_with_conf = f"person {confidence:.2f}"

                cv2.rectangle(resized_img, (x1, y1), (x2, y2), (255, 0, 255), 3)
                cv2.putText(resized_img, label_with_conf, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
                cv2.circle(resized_img, (center_x, center_y), 5, (0, 255, 0), -1)
                detected = "person"
```

```python

    label_publisher.publish(detected)
    return resized_img



def image_callback(msg):
    try:

        cv_image = bridge.imgmsg_to_cv2(msg, desired_encoding="passthrough")
        cv_image = cv2.cvtColor(cv_image, cv2.COLOR_RGB2BGR)
        # Detect objects in the image
        detected_image = detect_objects(cv_image)

        # Display the image with detections
        cv2.imshow("YOLO Object Detection", detected_image)
        cv2.waitKey(1)

    except Exception as e:
        rospy.logerr(e)

def main():
    # Initialize ROS node
    rospy.init_node('yolo_image_detection_node')



    # Subscribe to the image topic
    image_subscriber = rospy.Subscriber('image', ROSImage, image_callback)

    # Spin
    rospy.spin()

if __name__ == '__main__':
    main()

```

2- Lidar code

```python
#!/usr/bin/env python

import rospy
import sensor_msgs.point_cloud2 as pc2
from sensor_msgs.msg import PointCloud2
from std_msgs.msg import Float32MultiArray
import numpy as np
import math
import time

# Initialize the counter
counter = 0

# Initialize the publisher
pub = rospy.Publisher('lidar_points', Float32MultiArray, queue_size=10)

# Initialize global variables to store previous distance and time
prev_distance = None
prev_time = None
relative_velocity = 100
TTC = 50
distance = 50

def point_cloud_callback(msg):
    global counter, prev_distance, prev_time, relative_velocity, TTC, distance

    # Read points from the PointCloud2 message
    points = np.array(list(pc2.read_points(msg, field_names=("x", "y", "z"), skip_nans=True)))

    # Parameters for filtering and clustering
    target_height = 0.1  # 0.5 meters below the LiDAR
    height_tolerance = 0.2  # Tolerance for height filtering
    x_range = (-10, 10)  # Range of x coordinates
    y_range = (0, 14)  # Range of y coordinates
    clustering_distance = 2.0  # Maximum distance for clustering points together

    # Filtered and clustered points initialization
    filtered_points = []
    clustered_points = []

    # Filter points by height and position
    for point in points:
        x, y, z = point[:3]

        # Check if the point is within the height tolerance
        if abs(z - target_height) <= height_tolerance:
```

```python
            # Check if the point is within the x, y range
            if x_range[0] <= x <= x_range[1] and y_range[0] <= y <= y_range[1]:
                filtered_points.append((x, y, z))

    # Convert filtered points to numpy array for clustering
    filtered_points = np.array(filtered_points)

    # Perform clustering
    while len(filtered_points) > 0:
        # Initialize cluster with the first point
        cluster = [filtered_points[0]]
        remaining_points = []

        # Find nearby points and form clusters
        for point in filtered_points[1:]:
            dist = np.linalg.norm(np.array(cluster)[:, :2] - point[:2], axis=1)
            if np.min(dist) <= clustering_distance:
                cluster.append(point)
            else:
                remaining_points.append(point)

        # Compute the centroid of the cluster
        cluster = np.array(cluster)
        centroid = np.mean(cluster, axis=0)
        clustered_points.append(centroid)

        # Update filtered points to remaining points
        filtered_points = np.array(remaining_points)

    # Convert clustered points to numpy array for further processing
    clustered_points = np.array(clustered_points)
    counter += 1

    # Check if there are no clustered points
    if len(clustered_points) == 0:
        # Publish default values
        pub.publish(Float32MultiArray(data=[100] * 6))
    else:
        # Process the clustered points to calculate velocity
        for point in clustered_points:
            x, y, z = point
            rospy.loginfo(f"({counter}) Clustered Point: x={x}, y={y + 0.1}, z={z - 1.54}")  # With position offset to the vehicle origin

            # Measure the horizontal distance from the vehicle (neglecting Z)
            distance = math.sqrt(x**2 + y**2)

            # Get the current time
            current_time = time.time()

            # If we have a previous measurement, calculate the velocity
            if prev_distance is not None and prev_time is not None:
                # Calculate the change in distance
                delta_distance = distance - prev_distance

                # Calculate the change in time
                delta_time = current_time - prev_time

                # Avoid division by zero
                if delta_time > 0:
                    # Calculate the velocity
                    relative_velocity = delta_distance / delta_time

            # Update previous distance and time
            prev_distance = distance
            prev_time = current_time
            if relative_velocity > 0:
                TTC = distance / relative_velocity
            # Publish the results
            pub.publish(Float32MultiArray(data=[x, y, z, distance, relative_velocity, TTC]))  # Keep track of the arrangement of data

def main():
    rospy.init_node('clustered_point_lidar', anonymous=True)
    rospy.Subscriber('/velodyne_points', PointCloud2, point_cloud_callback, queue_size=1)
    rospy.spin()
```

3- Path 1 (Stright line)

```
#!/usr/bin/env python

import rospy
import numpy as np
from std_msgs.msg import Float64, String, Float32MultiArray
from tf.transformations import euler_from_quaternion

# Global variables
global wheel_base
global lookAhead
global current_position
global maxWheelVelocity
global object_detected
global TTC, x

maxWheelVelocity = 114.3202437
TTC = 0
x = 0

wheel_base = 2.26963
lookAhead = 2 # TUNABLE
current_position = [0, 0]

def init_node():
    global cmd_pub, steering_pub, brake_pub

    rospy.init_node("straight_line_control", anonymous=True)
    cmd_pub = rospy.Publisher('/cmd_vel', Float64, queue_size=10)
    steering_pub = rospy.Publisher('/SteeringAngle', Float64, queue_size=10)
    brake_pub = rospy.Publisher('/brakes', Float64, queue_size=10)
    object_sub = rospy.Subscriber('/detected_labels', String, manage_object)
    lidar_sub = rospy.Subscriber('/lidar_points', Float32MultiArray, manage_lidar)
    control_line(0)
    rate = rospy.Rate(10)
    rate.sleep()

def control_line(steering):
    global steering_pub, cmd_pub, brake_pub, TTC, x

    steering_pub.publish(steering)
    cmd_pub.publish(0.267)

    while True:
        if TTC < 14 and abs(x) < 3: # TUNABLE
            steering_pub.publish(0)
            cmd_pub.publish(0)
```

```python

    while True:
        if TTC < 14 and abs(x) < 3: # TUNABLE
            steering_pub.publish(0)
            cmd_pub.publish(0)
            brake_pub.publish(1)
        else:
            steering_pub.publish(0)
            cmd_pub.publish(0.1)
            brake_pub.publish(0)

def manage_object(msg):
    global object_detected
    object_detected = msg

def manage_lidar(msg):
    global TTC, x
    x = msg.data[0]
    y = msg.data[1]
    z = msg.data[2]
    distance = msg.data[3]
    relative_velocity = msg.data[4]
    TTC = msg.data[5]

if __name__ == '__main__':
    try:
        init_node()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass

```

4- Path 2 (Stright line with lane):

```python
#!/usr/bin/env python

import rospy
import numpy as np
from nav_msgs.msg import Odometry
from std_msgs.msg import Float64, String, Float32MultiArray
from tf.transformations import euler_from_quaternion

# Tunable parameters
look_ahead = 2  # Look-ahead distance
wheel_base = 2.26963  # Vehicle's wheelbase
human_stop_distance = 14  # Distance to stop for a human
lane_change_distance = 10  # Distance to change lane for a car or cone
lane_change_cooldown = 8  # Cooldown period for lane changes (seconds)
speed = 0.15  # Constant speed

# Global variables
detected_object = ""
x = y = 0
flag = "R"
last_lane_change_time = 0

def init_node():
    global cmd_pub, steering_pub, brake_pub, last_lane_change_time

    rospy.init_node("lane_change_control", anonymous=True)

    rospy.Subscriber('/odom', Odometry, call_back_odom)
    rospy.Subscriber('/detected_labels', String, manage_object)
    rospy.Subscriber('/lidar_points', Float32MultiArray, manage_lidar)
    cmd_pub = rospy.Publisher('/cmd_vel', Float64, queue_size=10)
    steering_pub = rospy.Publisher('/SteeringAngle', Float64, queue_size=10)
    brake_pub = rospy.Publisher('/brakes', Float64, queue_size=10)

    last_lane_change_time = rospy.get_time()

    rate = rospy.Rate(10)
    rate.sleep()

def emergency_stop():
    cmd_pub.publish(0)
    steering_pub.publish(0)
    brake_pub.publish(1)
    rospy.sleep(3)

def manage_object(msg):
    global detected_object
    detected_object = msg.data
```

```
    if flag == "L":
        waypoints = L_path
    elif flag == "R":
        waypoints = R_path

    for point in waypoints:
        if point[1] > (C_pose[1] + look_ahead) and point[1] <= (C_pose[1] + look_ahead + 6):

            dy = abs(point[1] - C_pose[1])
            dx = C_pose[0] - point[0]
            local_x = np.cos(yaw) * dy + np.sin(yaw) * dx
            local_y = -np.sin(yaw) * dy + np.cos(yaw) * dx

            curvature = 2 * local_y / (local_x ** 2 + local_y ** 2)
            steering = np.arctan(curvature * wheel_base) * 180 / np.pi
            if abs(steering) <= 1:  # TUNABLE
                steering = 0

            steering_pub.publish(steering)
            cmd_pub.publish(speed)  # TUNABLE
            brake_pub.publish(0)

if __name__ == '__main__':
    try:
        y_values = np.linspace(0, 200, 200)
        x_values = y_values * 0
        R_path = list(zip(x_values, y_values))
        x_values[:] = 4
        L_path = list(zip(x_values, y_values))

        init_node()
        rospy.spin()

    except rospy.ROSInterruptException:
        pass
```


```
    if flag == "L":
        waypoints = L_path
    elif flag == "R":
        waypoints = R_path

    for point in waypoints:
        if point[1] > (C_pose[1] + look_ahead) and point[1] <= (C_pose[1] + look_ahead + 6):

            dy = abs(point[1] - C_pose[1])
            dx = C_pose[0] - point[0]
            local_x = np.cos(yaw) * dy + np.sin(yaw) * dx
            local_y = -np.sin(yaw) * dy + np.cos(yaw) * dx

            curvature = 2 * local_y / (local_x ** 2 + local_y ** 2)
            steering = np.arctan(curvature * wheel_base) * 180 / np.pi
            if abs(steering) <= 1:  # TUNABLE
                steering = 0

            steering_pub.publish(steering)
            cmd_pub.publish(speed)  # TUNABLE
            brake_pub.publish(0)

if __name__ == '__main__':
    try:
        y_values = np.linspace(0, 200, 200)
        x_values = y_values * 0
        R_path = list(zip(x_values, y_values))
        x_values[:] = 4
        L_path = list(zip(x_values, y_values))

        init_node()
        rospy.spin()

    except rospy.ROSInterruptException:
        pass
```


5- Path3 (circular path):

```python
#!/usr/bin/env python

import rospy
import numpy as np
from std_msgs.msg import Float64, String, Float32MultiArray
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion

# Global parameters
global wheel_base
global lookAhead
global current_position
global maxWheelVelocity
global path_flag
global object_detected
global lane_change_cooldown
global last_lane_change_time

maxWheelVelocity = 114.3202437
wheel_base = 2.26963
lookAhead = 2  # TUNABLE
current_position = [0, 0]
radius_1 = 18  # Radius of the first circular path
radius_2 = 23  # Radius of the second circular path
path_flag = 1  # 1 for path with radius_1, 2 for path with radius_2
object_detected = ""
TTC = float('inf')
x = y = 0

# Thresholds
human_stop_distance = 11 # Distance to stop for human
car_cone_stop_distance = 11  # Distance to stop for car or cone
lane_change_distance = 10  # Distance to change lane for car or cone
lane_change_cooldown = 7  # Cooldown period in seconds

def init_node():
    global cmd_pub, steering_pub, brake_pub, last_lane_change_time

    rospy.init_node("pure_pursuit_control", anonymous=True)
    odom_sub = rospy.Subscriber('/odom', Odometry, calculate_lookAhead_waypoint)
    cmd_pub = rospy.Publisher('/cmd_vel', Float64, queue_size=10)
    steering_pub = rospy.Publisher('/SteeringAngle', Float64, queue_size=10)
    brake_pub = rospy.Publisher('/brakes', Float64, queue_size=10)
    object_sub = rospy.Subscriber('/detected_labels', String, manage_object)
    lidar_sub = rospy.Subscriber('/lidar_points', Float32MultiArray, manage_lidar)
```

```python
    last_lane_change_time = rospy.get_time()

    rate = rospy.Rate(10)
    rate.sleep()

def calculate_lookAhead_waypoint(odom):
    global current_position, yaw, path_flag

    # Update current position and orientation
    current_position[0] = odom.pose.pose.position.x
    current_position[1] = odom.pose.pose.position.y
    orientation_q = odom.pose.pose.orientation
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
    (_, _, yaw) = euler_from_quaternion(orientation_list)

    # Select the radius based on the path_flag
    if path_flag == 1:
        radius = radius_1
    else:
        radius = radius_2

    # Generate look-ahead points along the selected circular path
    angle = np.arctan2(current_position[1], current_position[0])
    lookAhead_angle = angle + lookAhead / radius
    lookAhead_point = (radius * np.cos(lookAhead_angle), radius * np.sin(lookAhead_angle))

    calculate_Curvature_nd_Steering(lookAhead_point)

def calculate_Curvature_nd_Steering(lookAhead_point):
    global yaw, current_position, wheel_base

    dx = lookAhead_point[0] - current_position[0]
    dy = lookAhead_point[1] - current_position[1]
    local_x = np.cos(yaw) * dx + np.sin(yaw) * dy
    local_y = -np.sin(yaw) * dx + np.cos(yaw) * dy

    if local_x == 0:
        control_line(0)
        return

    curvature = 2 * local_y / (local_x**2 + local_y**2)
    steering_angle = np.arctan(curvature * wheel_base) * 180 / np.pi

    if abs(steering_angle) < 0.1:  # TUNABLE
```

```python
    if abs(steering_angle) < 0.1:  # TUNABLE
        steering_angle = 0

    control_line(steering_angle)

def control_line(steering):
    global steering_pub, cmd_pub, brake_pub, object_detected, TTC, x, y, last_lane_change_time

    rospy.loginfo(steering)
    steering_pub.publish(steering)

    # Check for human
    if object_detected == "person" and np.sqrt(x**2 + y**2) <= human_stop_distance:
        cmd_pub.publish(0)
        brake_pub.publish(1)
        rospy.sleep(3)
    # Check for lane change condition
    elif object_detected in ["car", "cone"] and np.sqrt(x**2 + y**2) <= lane_change_distance:
        current_time = rospy.get_time()
        if current_time - last_lane_change_time > lane_change_cooldown:
            change_lane()
            last_lane_change_time = current_time
    else:
        cmd_pub.publish(0.1)  # Constant speed
        brake_pub.publish(0)

    # Example stopping condition, you can adjust this based on your needs

def manage_object(msg):
    global object_detected
    object_detected = msg.data

def manage_lidar(msg):
    global TTC, x, y
    x = msg.data[0]
    y = msg.data[1]
    z = msg.data[2]
    distance = msg.data[3]
    relative_velocity = msg.data[4]
    TTC = msg.data[5]

def change_lane():
    global path_flag
    if path_flag == 1:
        path_flag = 2
    else:
```

```python
    else:
        path_flag = 1

if __name__ == '__main__':
    try:
        init_node()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

6- Custom path

```python
#!/usr/bin/env python

import rospy
import numpy as np
from std_msgs.msg import Float64
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from std_msgs.msg import Float64, String, Float32MultiArray

# Global parameters
wheel_base = 2.26963
lookAhead = 2
current_position = [0.0, 0.0]
circle_flag = 0
flag = "L1"  # Initial flag
human_stop_distance = 14  # Distance to stop for a human
lane_change_distance = 11  # Distance to change lane for a car or cone
lane_change_cooldown = 8  # Cooldown period for lane changes (seconds)
speed = 0.15  # Constant speed
full_path = []
detected_object = ""
x = y = 0
last_lane_change_time = 0



'''def flag_control(detected_object="car", TTC=5):
    global flag

    Tmin = 4

    if detected_object == "person":
        emergency_stop()
    elif (detected_object in ["car", "cone"]) and TTC >= Tmin:
        flag = "L1" if flag == "L2" else "L2"
    elif (detected_object in ["car", "cone"]) and TTC < Tmin:
        emergency_stop()'''

def init_node():
    global cmd_pub, steering_pub, brake_pub

    rospy.init_node("pure_pursuit_control", anonymous=True)

    rospy.Subscriber('/odom', Odometry, calculate_lookAhead_waypoint)
```

```
    rospy.Subscriber('/odom', Odometry, calculate_lookAhead_waypoint)

    cmd_pub = rospy.Publisher('/cmd_vel', Float64, queue_size=10)
    steering_pub = rospy.Publisher('/SteeringAngle', Float64, queue_size=10)
    brake_pub = rospy.Publisher('/brakes', Float64, queue_size=10)
    rospy.Subscriber('/detected_labels', String, manage_object)
    rospy.Subscriber('/lidar_points', Float32MultiArray, manage_lidar)

    rate = rospy.Rate(10)
    rate.sleep()

def calculate_lookAhead_waypoint(odom):
    global current_position, yaw,flag

    # Update current position and orientation
    current_position[0] = odom.pose.pose.position.x
    current_position[1] = odom.pose.pose.position.y
    orientation_q = odom.pose.pose.orientation
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
    (_, _, yaw) = euler_from_quaternion(orientation_list)
    flag_control()
    rospy.loginfo(flag)



    if current_position[0] < 50 and current_position[0] >0 and current_position[1]<20 :
        rospy.loginfo("up")
        waypoints = upper_path2 if flag == "L2" else  U_path
        for point in waypoints:
            if point[0] > (current_position[0] + lookAhead) and point[0] <= (current_position[0] + lookAhead+3):
                calculate_curv_s(point)
                return


    elif current_position[0] >50 :
        rospy.loginfo("left")

        radius = 24 if flag == "L2" else  20


        angle = np.arctan2(current_position[1], current_position[0]-50)
        lookAhead_angle = angle + lookAhead / radius
        lookAhead_point = (50+(radius * np.cos(lookAhead_angle)), 20+(radius * np.sin(lookAhead_angle)))
        rospy.loginfo(lookAhead_point)
        calculate_curv_s(lookAhead_point)


    elif current_position[0]<= 50 and current_position[0] > 0 and current_position[1]>20:
        rospy.loginfo("down")
        waypoints = lower_path2 if flag == "L2" else  L_path
        for point in waypoints:
            if point[0] < (current_position[0] - lookAhead) and point[0] >= (current_position[0] - lookAhead-3):
                calculate_curv_s(point)
                return

    elif current_position[0]< 0 :
        radius = 24 if flag == "L2" else  20
        rospy.loginfo("right")
        angle = np.arctan2(current_position[1]+200, current_position[0])
        lookAhead_angle = angle + lookAhead / radius
        lookAhead_point = (-(radius * np.cos(lookAhead_angle)), (radius * np.sin(lookAhead_angle))-20)
        calculate_curv_s(lookAhead_point)





def calculate_curv_s(point):
    dy = point[1] - current_position[1]
    dx = current_position[0] - point[0]
    local_x = np.cos(yaw) * dy + np.sin(yaw) * dx
    local_y = -np.sin(yaw) * dy + np.cos(yaw) * dx

    curvature = 2 * local_y / (local_x ** 2 + local_y ** 2)
    steering_angle = np.arctan(curvature * wheel_base) * 180 / np.pi
    if abs(steering_angle) <= .5:
        steering_angle = 0

  #  rospy.loginfo(steering_angle)
    steering_pub.publish(Float64(steering_angle))
    cmd_pub.publish(Float64(.2))

def manage_object(msg):
    global detected_object
    detected_object = msg.data

def manage_lidar(msg):
    global x, y
    x = msg.data[0]
```

```python
        y = msg.data[1]

def flag_control():
    global detected_object, x, y, flag, last_lane_change_time

    if detected_object == "person" :#and y < human_stop_distance and abs(x) < 7:  # TUNABLE
        emergency_stop()
    elif detected_object in ["car", "cone"]:
        current_time = rospy.get_time()
        if np.sqrt(x**2 + y**2) <= lane_change_distance and current_time - last_lane_change_time > lane_change_cooldown:  # TUNABLE
            change_lane()
            last_lane_change_time = current_time


def change_lane():
    global flag
    if flag == "L1":
        flag = "L2"
    else:
        flag = "L1"

def emergency_stop():
    cmd_pub.publish(0)
    #steering_pub.publish(0)
    brake_pub.publish(1)
    rospy.sleep(3)
    brake_pub.publish(0)



if __name__ == '__main__':
    try:
        lookAhead = 3

        # Generate the upper and lower straight line
        x_upper = np.linspace(0, 50, 100)
        y_upper = x_upper * 0
        U_path = list(zip(x_upper, y_upper))

        x_lower = x_upper
        y_lower = y_upper
        y_lower[:] = 40
        L_path = list(zip(x_lower, y_lower))

        # Generate the first half circle
```

```python
        # Generate the first half circle
        theta1 = np.linspace(0, np.pi, 100)
        radius = 20
        x_circle1 =  radius * np.cos(theta1)
        y_circle1 =20 - radius * np.sin(theta1)
        R_circle = list(zip(x_circle1, y_circle1))

        # Generate the left half circle
        theta2 = np.linspace(0, np.pi , 100)
        radius = 20
        x_circle2 = 50 + radius * np.cos(theta2)
        y_circle2 = 20 + radius * np.sin(theta2)
        L_circle = list(zip(x_circle2, y_circle2))




        #------------------------------------------------

        #lane 2
        y_upper[:] = -4
        upper_path2 = list(zip(x_upper, y_upper))

        y_lower [:] =  44
        lower_path2 = list(zip(x_lower, y_lower))

        # Combine all paths
        full_path = U_path + L_circle + L_path + R_circle

        init_node()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

7- CSV_Generation:

```python
import rospy
import csv
import math
import numpy as np
import time
from std_msgs.msg import Float64, Int32
from nav_msgs.msg import Odometry
from sensor_msgs.msg import Imu, PointCloud2
from geometry_msgs.msg import Point, Quaternion, Vector3
import sensor_msgs.point_cloud2 as pc

# Initialize global variables
start_time = time.time()
flag = 0
Dict = {}

x_odom_list = []
y_odom_list = []
yaw_odom_list = []
velocity_odom = []
acceleration_odom = []

angular_velocity_x_imu = []
angular_velocity_y_imu = []
angular_velocity_z_imu = []

# Calculate velocity and acceleration
def calculate_velocity_acceleration(prev_x, prev_y, prev_time, curr_x, curr_y, curr_time):
    velocity = math.sqrt((curr_x - prev_x)**2 + (curr_y - prev_y)**2) / (curr_time - prev_time)
    acceleration = (velocity - (math.sqrt((prev_x - prev_x)**2 + (prev_y - prev_y)**2) / prev_time)) / (curr_time - prev_time)
    return velocity, acceleration

# Calculate mean and RMS
def calculate_mean_rms(data_list):
    mean_value = np.mean(data_list)
    rms_value = np.sqrt(np.mean(np.square(data_list)))
    return mean_value, rms_value

# Odometry callback
def odom_callback(msg):
    global x_odom_list, y_odom_list, yaw_odom_list, velocity_odom, acceleration_odom, Dict, start_time
    x_odom = msg.pose.pose.position.x
    y_odom = msg.pose.pose.position.y
    yaw_odom = msg.pose.pose.orientation.z
    curr_time = time.time() - start_time

    if x_odom_list and y_odom_list:
        prev_x = x_odom_list[-1]
        prev_y = y_odom_list[-1]
```

```
            prev_y = y_odom_list[-1]
            prev_time = curr_time - 1
            velocity, acceleration = calculate_velocity_acceleration(prev_x, prev_y, prev_time, x_odom, y_odom, curr_time
            velocity_odom.append(velocity)
            acceleration_odom.append(acceleration)
        else:
            velocity_odom.append(0)
            acceleration_odom.append(0)

        x_odom_list.append(x_odom)
        y_odom_list.append(y_odom)
        yaw_odom_list.append(yaw_odom)

        mean_x_odom, rms_x_odom = calculate_mean_rms(x_odom_list)
        mean_y_odom, rms_y_odom = calculate_mean_rms(y_odom_list)
        mean_yaw_odom, rms_yaw_odom = calculate_mean_rms(yaw_odom_list)

        Dict.update({
            "positions_x_odom": x_odom,
            "positions_y_odom": y_odom,
            "velocity": velocity_odom[-1],
            "acceleration": acceleration_odom[-1],
            "yaw_odom": yaw_odom,
            "Time_Sec": curr_time,
            "mean_x_odom": mean_x_odom,
            "rms_x_odom": rms_x_odom,
            "mean_y_odom": mean_y_odom,
            "rms_y_odom": rms_y_odom,
            "mean_yaw_odom": mean_yaw_odom,
            "rms_yaw_odom": rms_yaw_odom
        })

        CSV_SAVE()

    # IMU callback
    def imu_callback(msg):
        global angular_velocity_x_imu, angular_velocity_y_imu, angular_velocity_z_imu, Dict
        angular_velocity_x = msg.angular_velocity.x
        angular_velocity_y = msg.angular_velocity.y
        angular_velocity_z = msg.angular_velocity.z
        angular_velocity_x_imu.append(angular_velocity_x)
        angular_velocity_y_imu.append(angular_velocity_y)
        angular_velocity_z_imu.append(angular_velocity_z)

        Dict.update({
            "angular_velocity_x_imu": angular_velocity_x,
            "angular_velocity_y_imu": angular_velocity_y,
            "angular_velocity_z_imu": angular_velocity_z
        })
```

```
    CSV_SAVE()

# Command velocity callback
def cmd_vel_callback(msg):
    velocity_cmd = msg.data
    Dict.update({"velocity_cmd": velocity_cmd})
    CSV_SAVE()

# Steering angle callback
def steering_angle_callback(msg):
    steering_angle = msg.data
    Dict.update({"steering_angle": steering_angle})
    CSV_SAVE()


#-------------------------------------------------------------------
#              ADD Ridar Reading
#--------------------------------------------------------------------
# Point cloud callback (commented out)
# def point_cloud_callback(msg):
#     x_points = []
#     y_points = []
#     z_points = []
#     for point in pc.read_points(msg, field_names=("x", "y", "z"), skip_nans=True):
#         x_points.append(point[0])
#         y_points.append(point[1])
#         z_points.append(point[2])
#     Dict.update({"x_points": x_points, "y_points": y_points, "z_points": z_points})
#     CSV_SAVE()

def listener():
    rospy.init_node('listen', anonymous=True)
    rospy.Subscriber("odom", Odometry, odom_callback)
    rospy.Subscriber("imu", Imu, imu_callback)
    rospy.Subscriber("cmd_vel", Float64, cmd_vel_callback)
    rospy.Subscriber("SteeringAngle", Float64, steering_angle_callback)

    # subscriber of lidar
    # rospy.Subscriber("velodyne_points", PointCloud2, point_cloud_callback)
    rospy.spin()

def CSV_SAVE():
    global flag
    with open("/home/eslam/catkin_workspace/src/t2/scripts/Dimensions31.csv", mode="a") as csvfile:
        fieldnames = [
            "positions_x_odom", "positions_y_odom", "velocity", "acceleration",
            "yaw_odom", "Time_Sec", "angular_velocity_x_imu", "angular_velocity_y_imu",
            "angular_velocity_z_imu", "velocity_cmd", "steering_angle",
            "mean_x_odom", "rms_x_odom", "mean_y_odom", "rms_y_odom", "mean_yaw_odom", "rms_yaw_odom"
```

```

def CSV_SAVE():
    global flag
    with open("/home/eslam/catkin_workspace/src/t2/scripts/Dimensions31.csv", mode="a") as csvfile:
        fieldnames = [
            "positions_x_odom", "positions_y_odom", "velocity", "acceleration",
            "yaw_odom", "Time_Sec", "angular_velocity_x_imu", "angular_velocity_y_imu",
            "angular_velocity_z_imu", "velocity_cmd", "steering_angle",
            "mean_x_odom", "rms_x_odom", "mean_y_odom", "rms_y_odom", "mean_yaw_odom", "rms_yaw_odom"
            # Delecrtion of lidar point
            #, "x_points", "y_points", "z_points"
        ]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        if flag == 0:
            writer.writeheader()
            flag = 1
        writer.writerow(Dict)

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

Matlab:

1- Draw Graphs:

```
% Define the file paths
csvFilePath = '/home/eslam/catkin_workspace/src/t2/scripts/Dimensions33.csv';
saveDir = '/home/eslam/catkin_workspace/src/t2/scripts/Graphs3';

% Create the directory if it doesn't exist
if ~exist(saveDir, 'dir')
    mkdir(saveDir);
end

% Read the CSV file
data = readtable(csvFilePath);

% Extract data
time = data.Time_Sec;
positions_x_odom = data.positions_x_odom;
positions_y_odom = data.positions_y_odom;
yaw_odom = data.yaw_odom;
velocity = data.velocity;
acceleration = data.acceleration;
rms_x_odom = data.rms_x_odom;
rms_y_odom = data.rms_y_odom;
rms_yaw_odom = data.rms_yaw_odom;
velocity_cmd = data.velocity_cmd;
steering_angle = data.steering_angle;

% Function to save plots
savePlot = @(fig, name) saveas(fig, fullfile(saveDir, name), 'png');

% Plot and save graph for x and y that out from odom
fig = figure;
plot(positions_x_odom, positions_y_odom);
title('X and Y from Odometry');
xlabel('X from Odometry');
ylabel('Y from Odometry');
grid on;
savePlot(fig, 'xy_odom.png');
close(fig);

% Plot and save graph for x out from odom with time
fig = figure;
plot(time, positions_x_odom);
title('X from Odometry over Time');
xlabel('Time (s)');
ylabel('X from Odometry');
grid on;
savePlot(fig, 'x_odom_time.png');
close(fig);
```

```
% Plot and save graph for y out from odom with time
fig = figure;
plot(time, positions_y_odom);
title('Y from Odometry over Time');
xlabel('Time (s)');
ylabel('Y from Odometry');
grid on;
savePlot(fig, 'y_odom_time.png');
close(fig);

% Plot and save graph for yaw out from odom with time
fig = figure;
plot(time, yaw_odom);
title('Yaw from Odometry over Time');
xlabel('Time (s)');
ylabel('Yaw from Odometry');
grid on;
savePlot(fig, 'yaw_odom_time.png');
close(fig);

% Plot and save graph for velocity with time
fig = figure;
plot(time, velocity);
title('Velocity over Time');
xlabel('Time (s)');
ylabel('Velocity');
grid on;
savePlot(fig, 'velocity_time.png');
close(fig);

% Plot and save graph for acceleration with time
fig = figure;
plot(time, acceleration);
title('Acceleration over Time');
xlabel('Time (s)');
ylabel('Acceleration');
grid on;
savePlot(fig, 'acceleration_time.png');
close(fig);

% Plot and save graph for RMS_x with time
fig = figure;
plot(time, rms_x_odom);
title('RMS X from Odometry over Time');
xlabel('Time (s)');
ylabel('RMS X from Odometry');
grid on;
savePlot(fig, 'rms_x_odom_time.png');
close(fig);
```

```
fig = figure;
plot(time, rms_x_odom);
title('RMS X from Odometry over Time');
xlabel('Time (s)');
ylabel('RMS X from Odometry');
grid on;
savePlot(fig, 'rms_x_odom_time.png');
close(fig);

% Plot and save graph for RMS_y with time
fig = figure;
plot(time, rms_y_odom);
title('RMS Y from Odometry over Time');
xlabel('Time (s)');
ylabel('RMS Y from Odometry');
grid on;
savePlot(fig, 'rms_y_odom_time.png');
close(fig);

% Plot and save graph for RMS_yaw with time
fig = figure;
plot(time, rms_yaw_odom);
title('RMS Yaw from Odometry over Time');
xlabel('Time (s)');
ylabel('RMS Yaw from Odometry');
grid on;
savePlot(fig, 'rms_yaw_odom_time.png');
close(fig);

% Plot and save graph for velocity_cmd with time
fig = figure;
plot(time, velocity_cmd);
title('Commanded Velocity over Time');
xlabel('Time (s)');
ylabel('Commanded Velocity');
grid on;
savePlot(fig, 'velocity_cmd_time.png');
close(fig);

% Plot and save graph for steering_angle with time
fig = figure;
plot(time, steering_angle);
title('Steering Angle over Time');
xlabel('Time (s)');
ylabel('Steering Angle');
grid on;
savePlot(fig, 'steering_angle_time.png');
close(fig);
```

2- Filter and noise

```
clear;
clc;

% Specify the file name
filename = '/home/eslam/catkin_workspace/src/t2/scripts/Dimensions33.csv';

% Read the data from the CSV file
data = readtable(filename);

columnData_X = data.positions_x_odom;
columnData_X_clean = columnData_X(~isnan(columnData_X));

columnData_Y = data.positions_y_odom;
columnData_Y_clean = columnData_Y(~isnan(columnData_Y));

columnData_yaw = data.yaw_odom;
columnData_yaw_clean = columnData_yaw(~isnan(columnData_yaw));

columnData_time = data.Time_Sec;
columnData_time_clean = columnData_time(~isnan(columnData_time));

data_rms_x = data.rms_x_odom;
data_rms_x_clean = data_rms_x(~isnan(data_rms_x));

data_rms_y = data.rms_y_odom;
data_rms_y_clean = data_rms_y(~isnan(data_rms_y));

data_rms_yaw = data.rms_yaw_odom;
data_rms_yaw_clean = data_rms_yaw(~isnan(data_rms_yaw));

xPointsM= awgn(columnData_X_clean, 30.78, "measured");     % old = 0.0313
yPointsM= awgn(columnData_Y_clean, 25.9, "measured");      % old = 46.3
yawPointsM= awgn(columnData_yaw_clean,0.075,"measured");



% Apply Noise to x y yaw and store it in a New CSV called NoisyOdom.csv for line path--------

A = [ 1 0 0;
      0 1 0;
      0 0 1];

B = [0 1 1;
     0 1 1;
     1 0 0;];

H = [1;
     0;
     0];          % Observation matrix
```

```
Q = 0.629*eye(3);     % Process noise covariance to be applied to the whole matrix [2 x 2]

R = 10*eye(3);           % Measurement noise covariance old 0.1

x = [0;
     0;
     0];        % Initial state estimate for x, y directions and yaw.

P = eye(3);         % Initial error covariance to be large number


columnData_YN = yPointsM;
filtered_yPoints = zeros(size(columnData_YN));
for i = 1:length(columnData_YN)
    % Predictionstep
    x = A.*x;                          % Predicted state estimate
    P = A.*P.*A' + Q;                % Predicted error covariance
    % P = A.*P. *A' + Q;                 % Predicted error covariance

    % Update step
    K = P.*H'/((H.*P).*H' + R);           % Kalman gain
    % K = eye(2).*H'/(H.*eye(2).*H' + R);  % Kalman gain (using identity matrix for P)

    x = x + K.*(columnData_YN(i) - H.*x); % Updated state estimate
    P = P - K.*(H.*P);              % Updated error covariance  (use TIMES (.) for elementwise multiplication.)

    filtered_yPoints(i) = x(1);    % Store filtered position
end

columnData_XN = xPointsM;
filtered_xPoints = zeros(size(columnData_XN));
for i = 1:length(columnData_XN)
    % Predictionstep
    x = A.*x;                          % Predicted state estimate
    P = A.*P.*A' + Q;                % Predicted error covariance
    % P = A.*P. *A' + Q;                 % Predicted error covariance

    % Update step
    K = P.*H'/((H.*P).*H' + R);           % Kalman gain
    % K = eye(2).*H'/(H.*eye(2).*H' + R);  % Kalman gain (using identity matrix for P)

    x = x + K.*(columnData_XN(i) - H.*x); % Updated state estimate
    P = P - K.*(H.*P);              % Updated error covariance  (use TIMES (.) for elementwise multiplication.)

    filtered_xPoints(i) = x(1);    % Store filtered position
end

columnData_YawN = yawPointsM;
```

```
columnData_YawN = yawPointsM;
filtered_yawPoints = zeros(size(columnData_YawN));
for i = 1:length(columnData_YawN)
    % Predictionstep
    x = A.*x;                          % Predicted state estimate
    P = A.*P.*A' + Q;                % Predicted error covariance
    % P = A.*P. *A' + Q;                   % Predicted error covariance

    % Update step
    K = P.*H'/((H.*P).*H' + R);            % Kalman gain
    % K = eye(2).*H'/(H.*eye(2).*H' + R);  % Kalman gain (using identity matrix for P)

    x = x + K.*(columnData_YawN(i) - H.*x); % Updated state estimate
    P = P - K.*(H.*P);                 % Updated error covariance  (use TIMES (.) for elementwise multiplication.)

    filtered_yawPoints(i) = x(1);     % Store filtered position
end


%{
%true value plot
plot(columnData_X_clean,columnData_Y_clean , '-b');
xlabel('odom_x');
ylabel('odom_y');
grid on;
hold on;
%}

%noise plot
plot(xPointsM,yPointsM, '-r');
xlabel('noised x');
ylabel('noised y');
grid on;
hold on;

%filter plot
plot(filtered_xPoints,filtered_yPoints, '-g');
xlabel('filtered x');
ylabel('filtered y');
grid on;
hold on;

legend ('noisy signal','filtered signal')

overall_RMS_x = mean(data_rms_x_clean)
overall_RMS_y = mean(data_rms_y_clean)
overall_RMS_yaw = mean(data_rms_yaw_clean)

```


```
hold on;

%filter plot
plot(filtered_xPoints,filtered_yPoints, '-g');
xlabel('filtered x');
ylabel('filtered y');
grid on;
hold on;

legend ('noisy signal','filtered signal')

overall_RMS_x = mean(data_rms_x_clean)
overall_RMS_y = mean(data_rms_y_clean)
overall_RMS_yaw = mean(data_rms_yaw_clean)
```


```


new_data = table(columnData_X_clean, columnData_Y_clean, columnData_yaw_clean, xPointsM, yPointsM, yawPointsM, filtered_xPoints, filtered_yPoints, filtered_yawPoints, ...
                 'VariableNames', {'x_odom', 'y_odom', 'yaw_odom', 'x_noisy', 'y_noisy', 'yaw_noisy', 'x_filtered', 'y_filtered', 'yaw_filtered'});

new_file = '/home/eslam/catkin_workspace/src/t2/scripts/change_lane_path_odom_noise_filter.csv';

writetable(new_data, new_file);
```

Performance analysis

Analyze the system's performance on each track, while providing graphs that validates the performance of your system, discussing any challenges encountered and how they were addressed.

Path1 (Stright line):

RMS Values

overall_RMS_x = 0.0814
overall_RMS_y = 30.7836
overall_RMS_yaw = 0.0014

Strengths

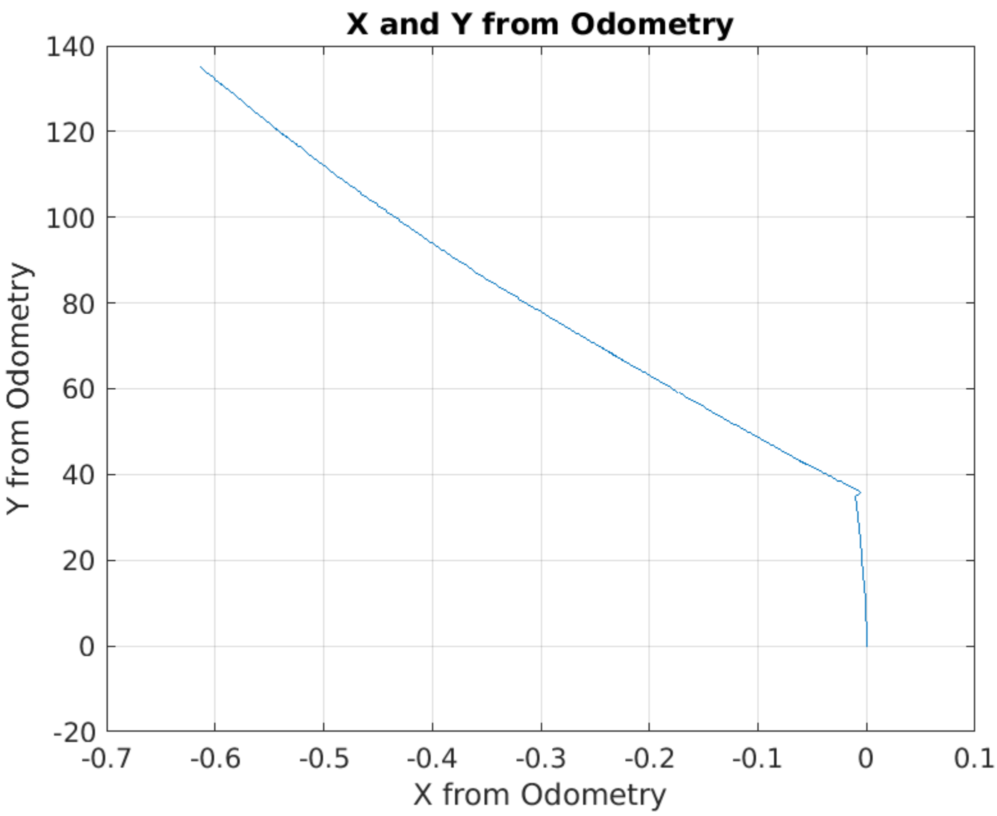
- **Real-Time Object Detection:** The system can detect objects and react in real-time, ensuring safety by stopping the vehicle when a human is detected.
- **Simple Control Logic:** The control logic is straightforward, making it easy to understand and implement.
- **Pure Pursuit Algorithm:** Using the Pure Pursuit algorithm allows for smooth and efficient path following.

Weaknesses

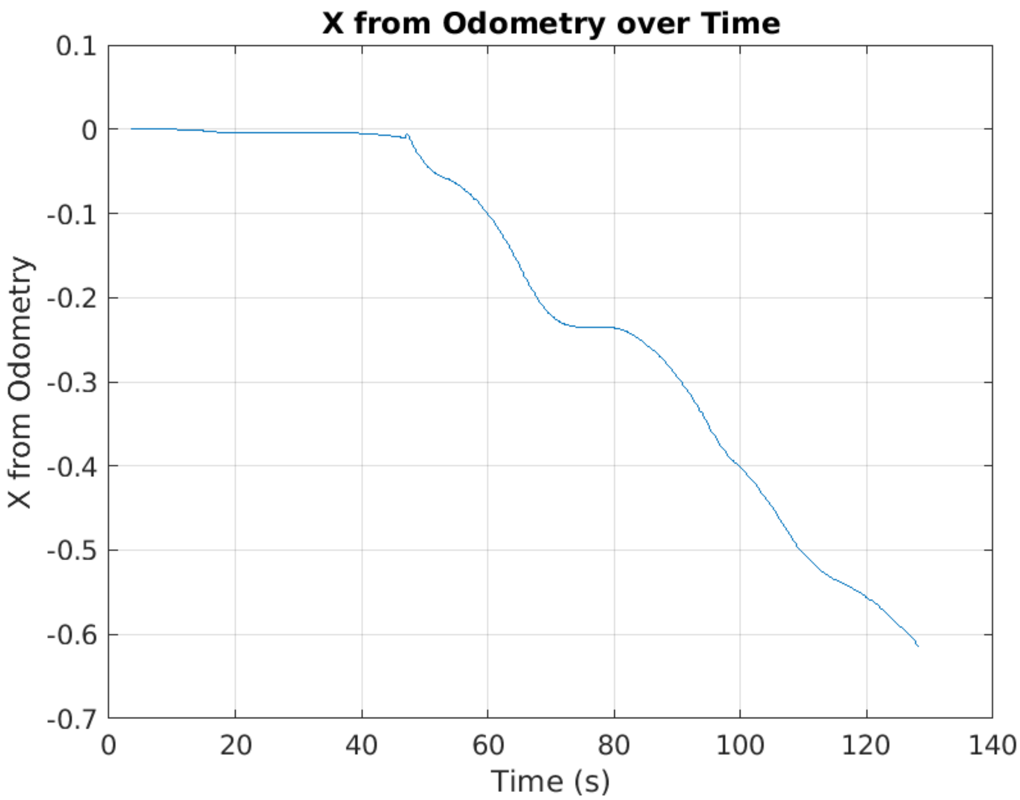
- **Limited Manoeuvrability:** The current implementation focuses on straight-line movement with limited handling of complex scenarios like lane changes or turns.
- **Fixed Parameters:** The system uses fixed parameters for look-ahead distance and velocity, which may not be optimal for all situations.
- **Basic Obstacle Handling:** The obstacle handling mechanism is basic, primarily focusing on emergency stops without more sophisticated avoidance strategies.

Graphs

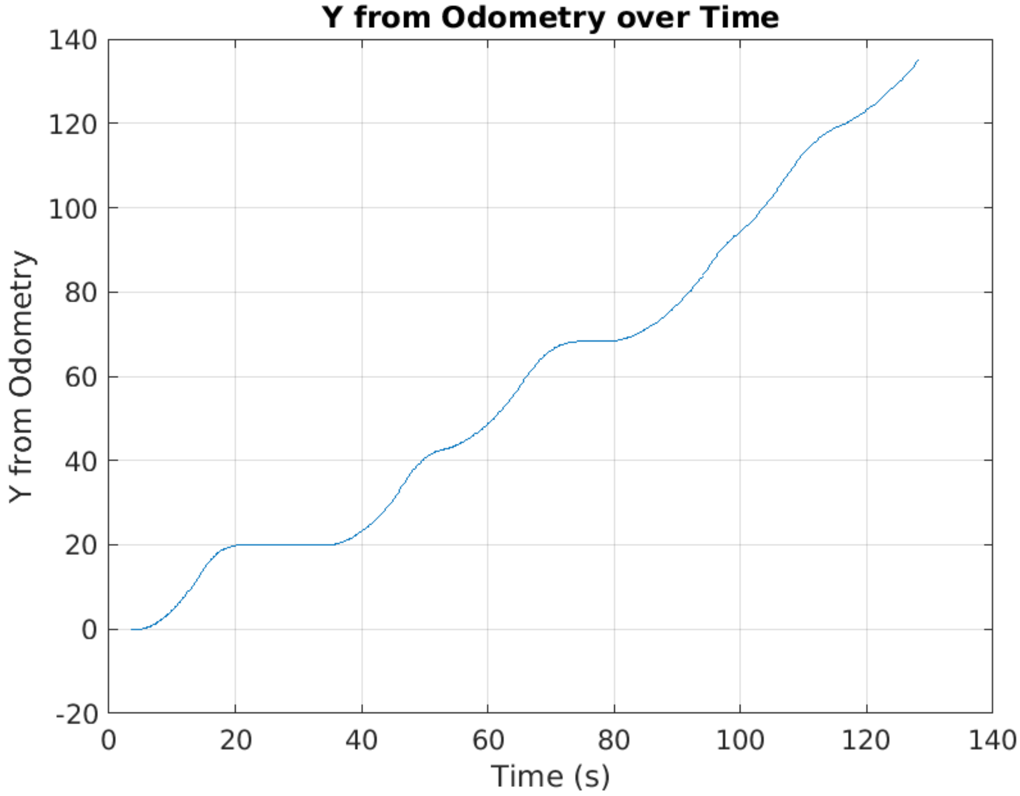
X vs Y



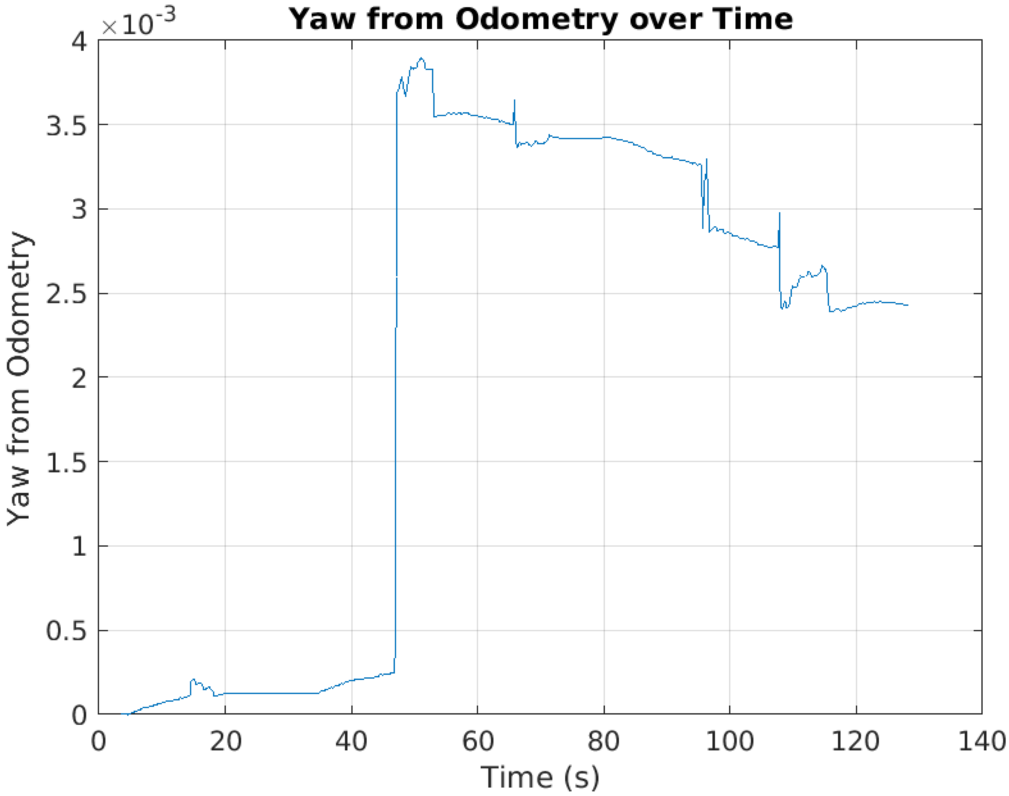
X vs Time


X from Odometry over Time
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
0
20
40
60
80
100
120
140
Time (s)
X from Odometry

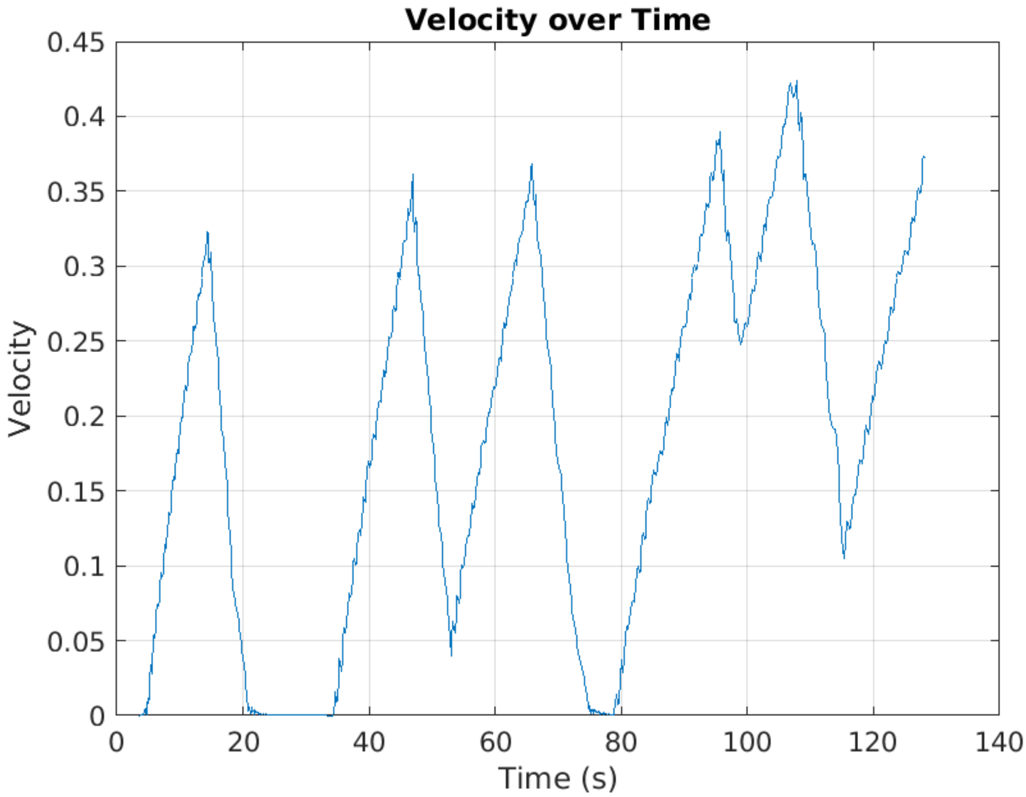
Y vs Time


Y from Odometry over Time
140
120
100
80
60
40
20
0
-20
0
20
40
60
80
100
120
140
Time (s)
Y from Odometry

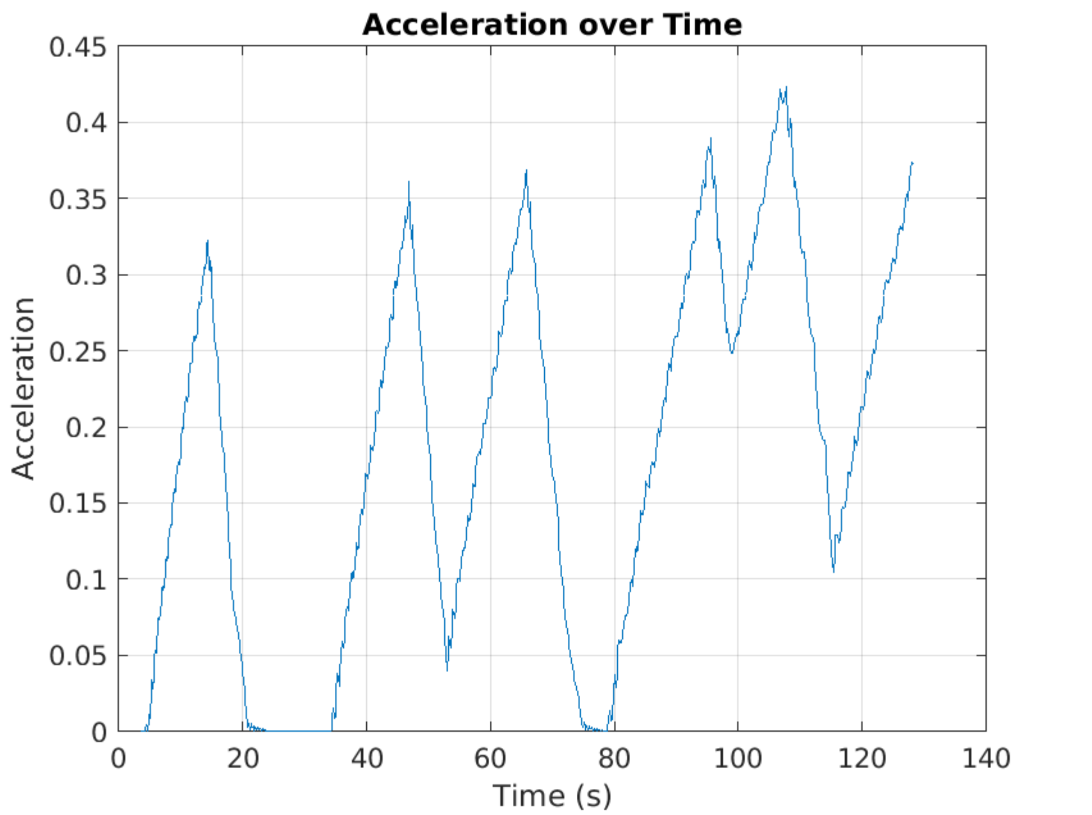
Yaw vs Time



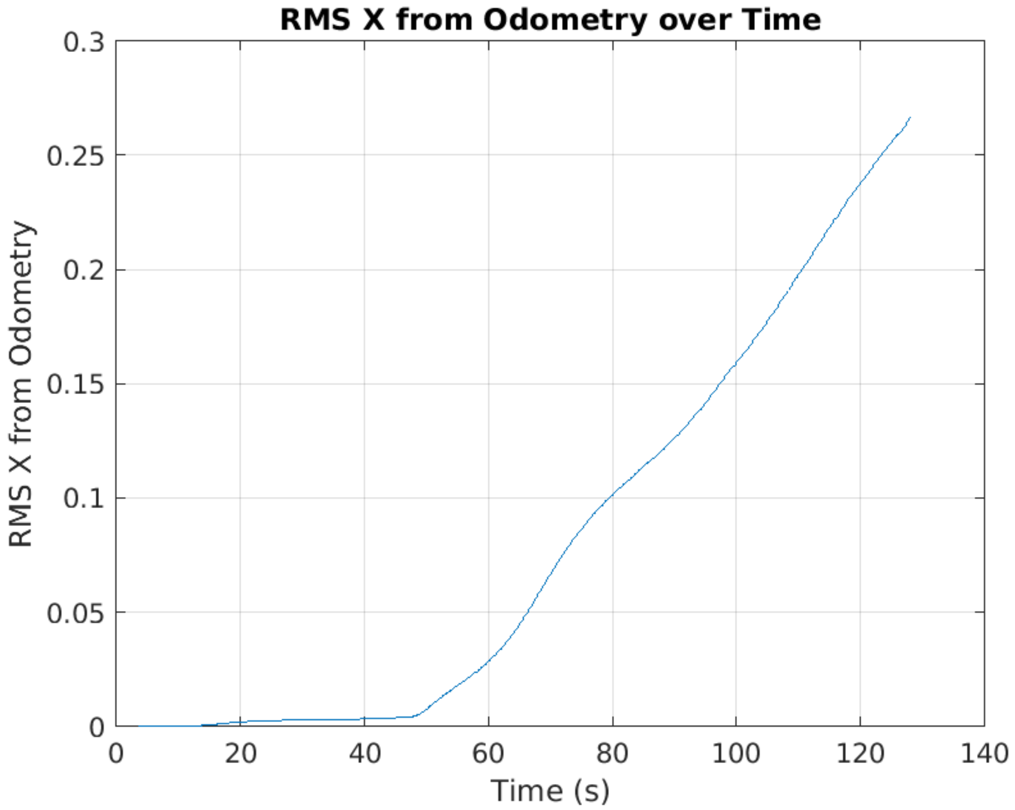
Velocity vs Time



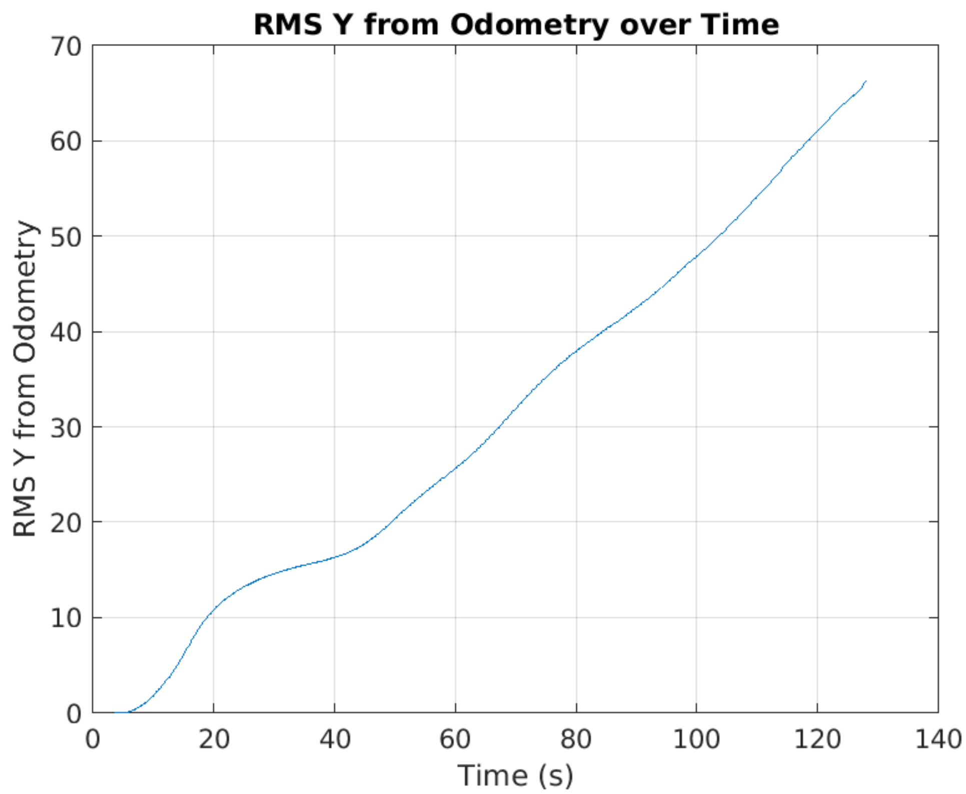
Acceleration vs Time



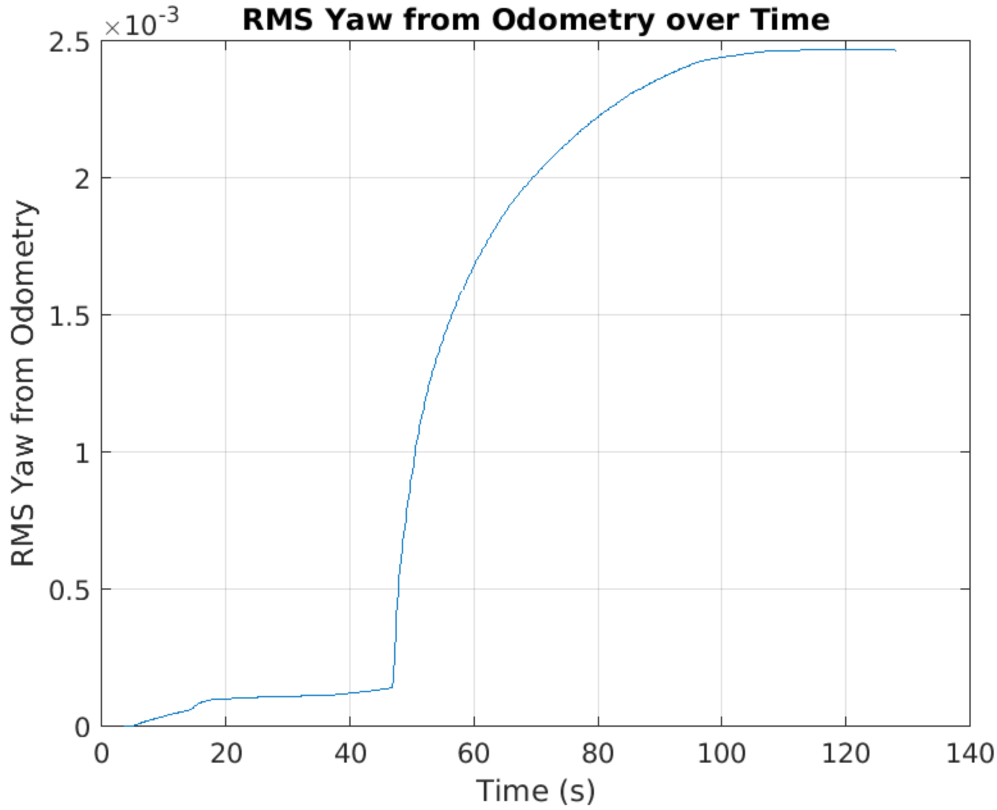
RMS_X vs Time



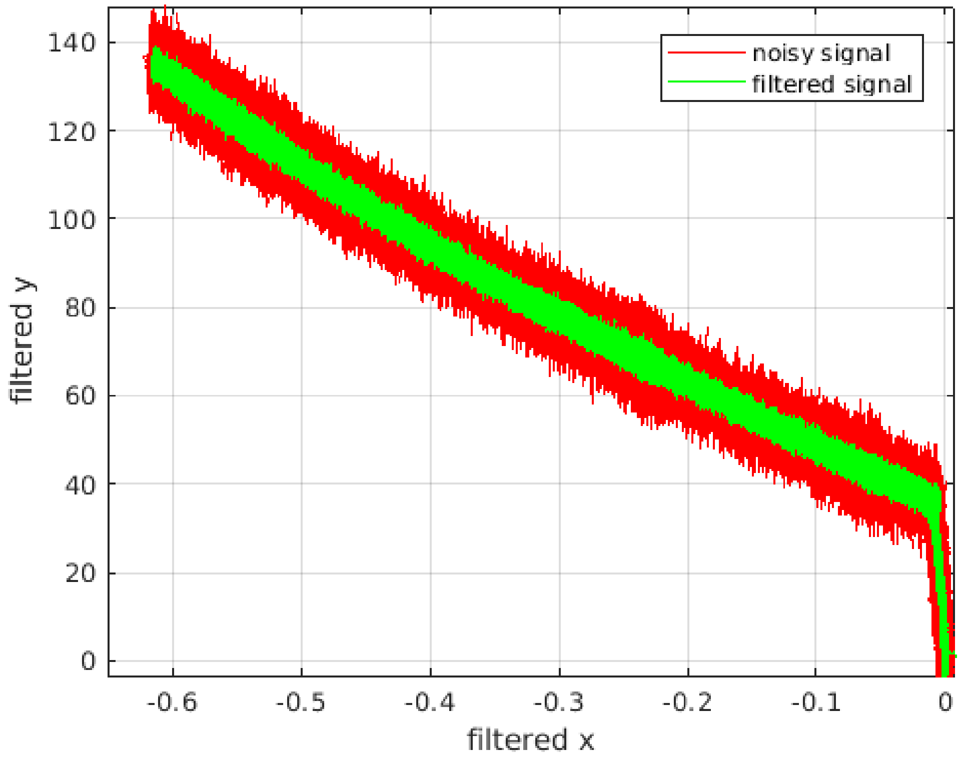
RMS_Y vs Time



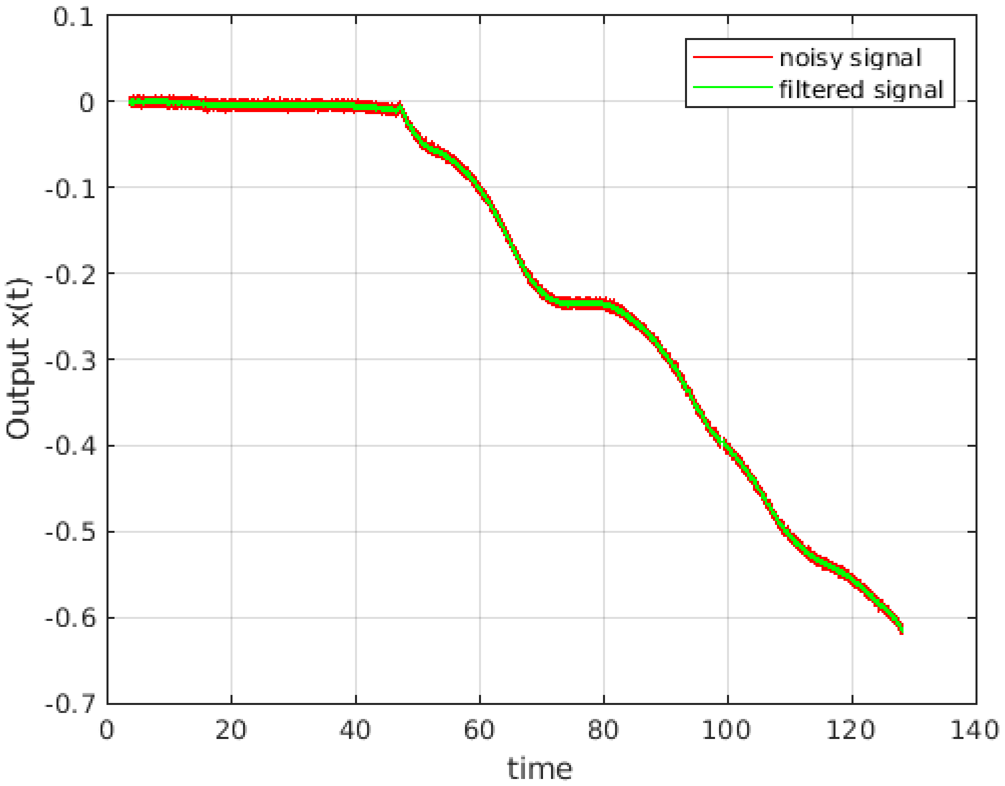
RMS_YAW vs Time



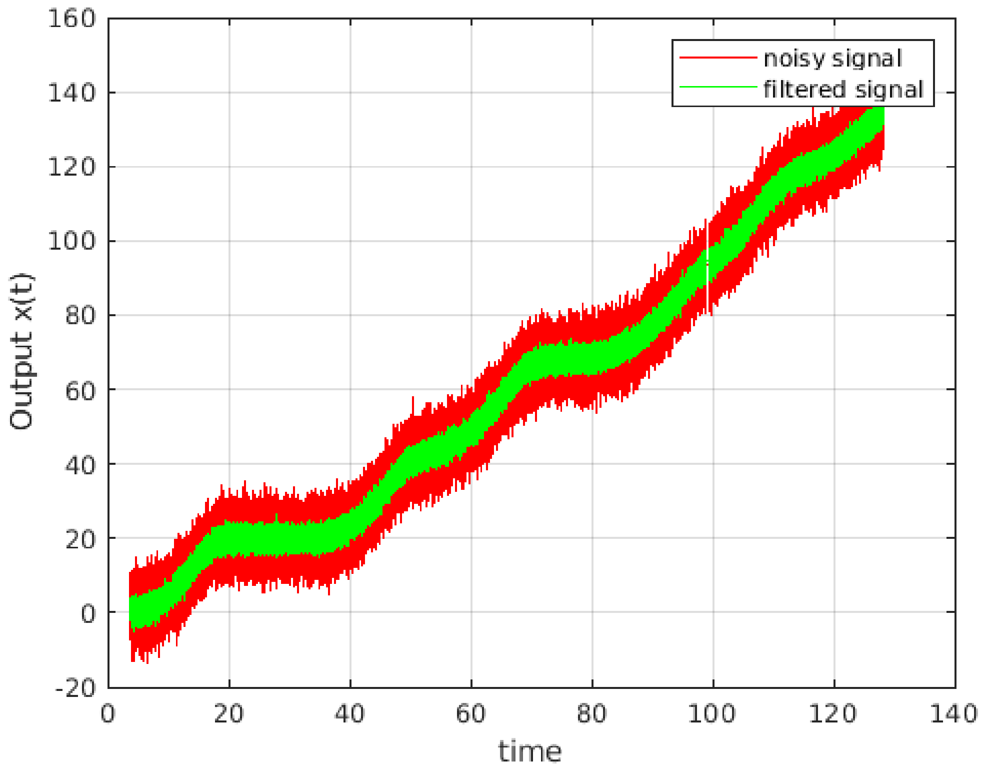
X vs Y filtered



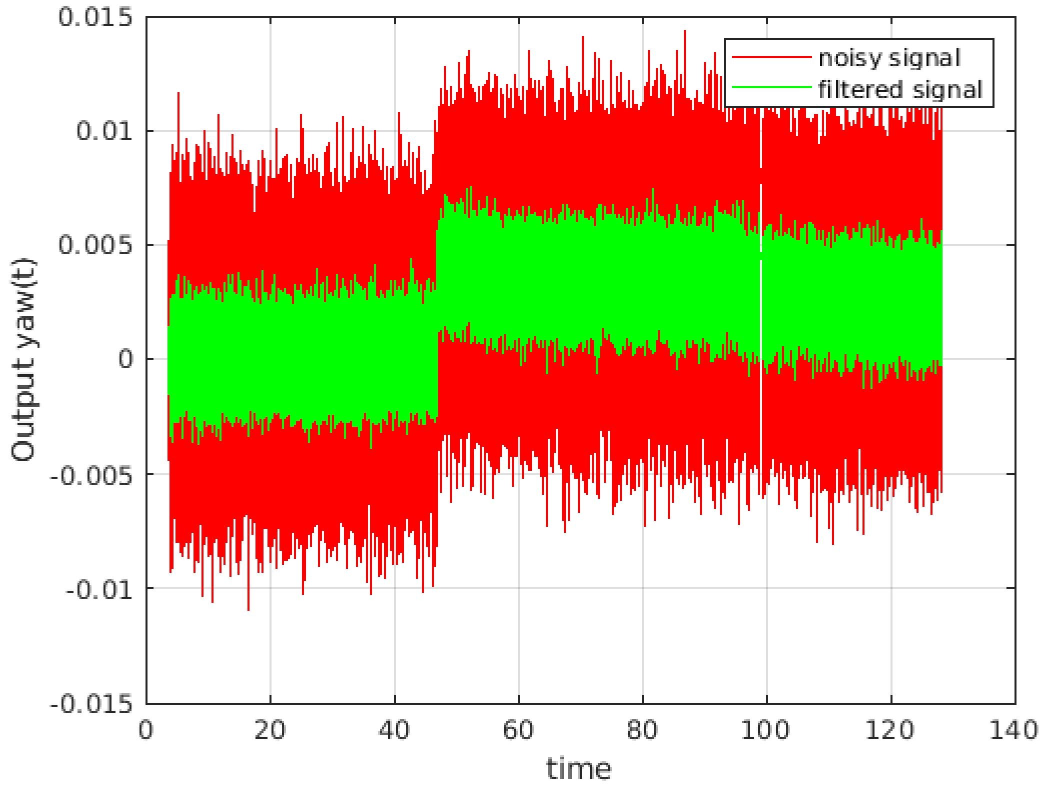
X vs Time filtered



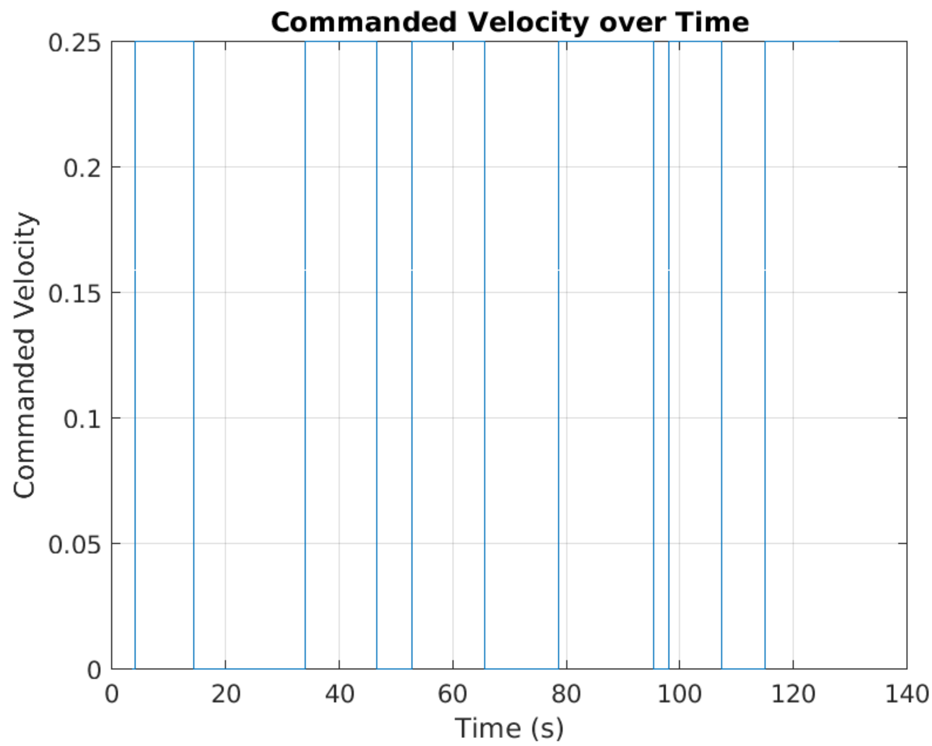
Y vs Time filtered



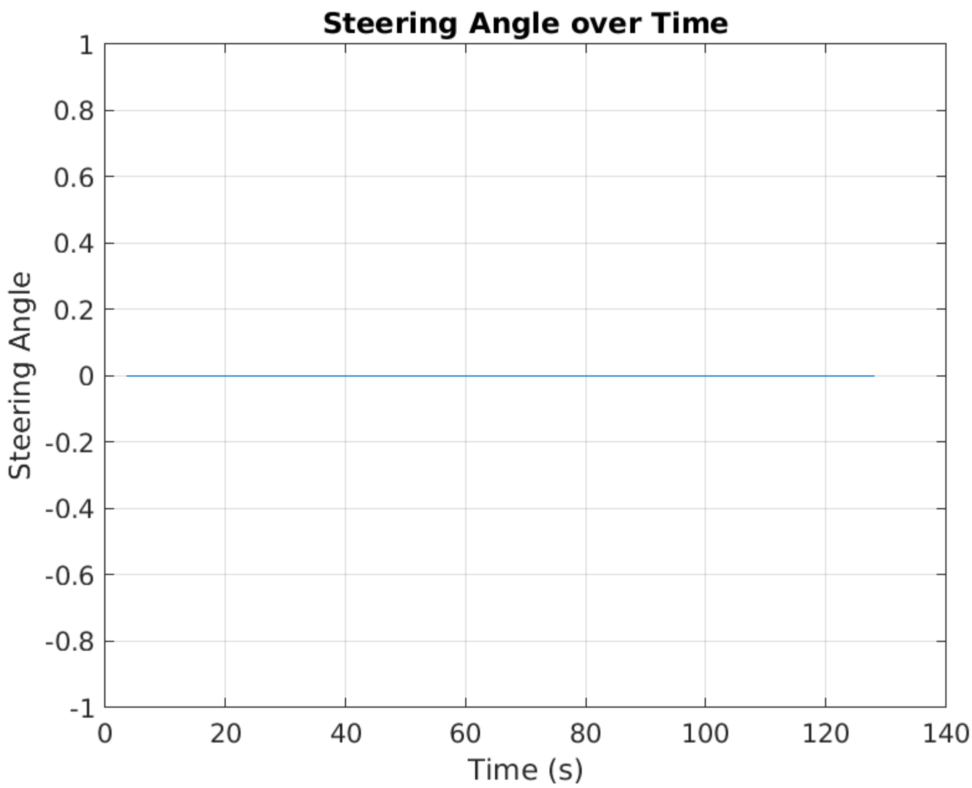
Yaw vs Time filtered



velocity_cmd vs time



steering_angle vs Time



Path2 (Stright line with a change lane):

RMA values

overall_RMS_x = 1.3080

overall_RMS_y = 23.0330

overall_RMS_yaw = 0.0540

Strengths

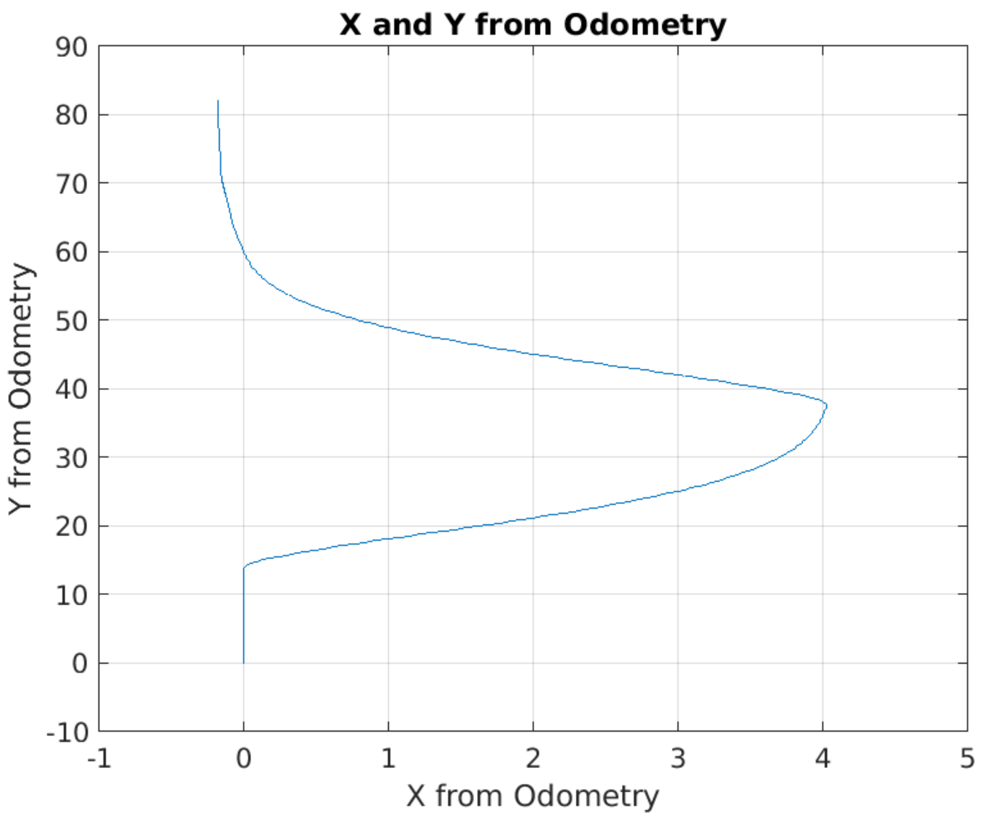
- **Real-Time Object Detection and Response:** The system can detect objects in real-time and respond appropriately, ensuring safety.
- **Lane Change Capability:** The vehicle can perform lane changes to avoid obstacles, improving manoeuvrability.
- **Pure Pursuit Algorithm:** The use of the Pure Pursuit algorithm allows for smooth path following.

Weaknesses

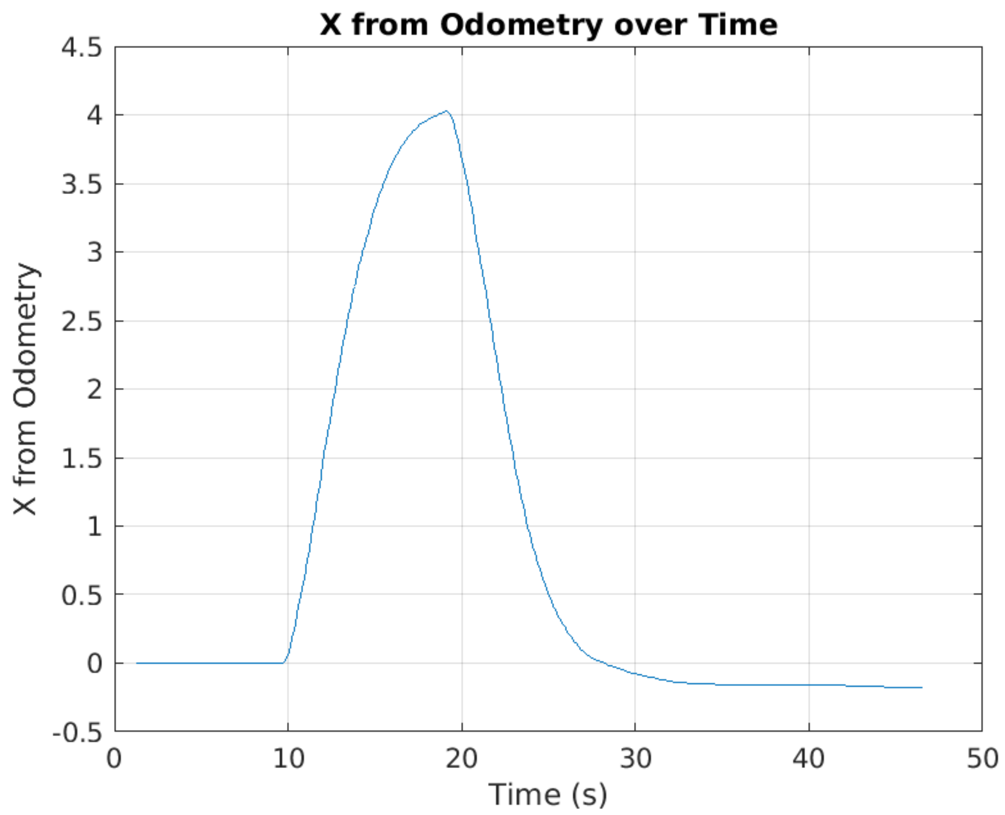
- **Limited Obstacle Handling:** The system primarily focuses on emergency stops and lane changes, lacking more advanced obstacle avoidance strategies.
- **Fixed Parameters:** The use of fixed parameters for look-ahead distance, speed, and distances for stopping and lane changes may not be optimal for all scenarios.
- **Simplified Path Planning:** The path planning is based on predefined waypoints, which may not be sufficient for more complex environments.

Graphs

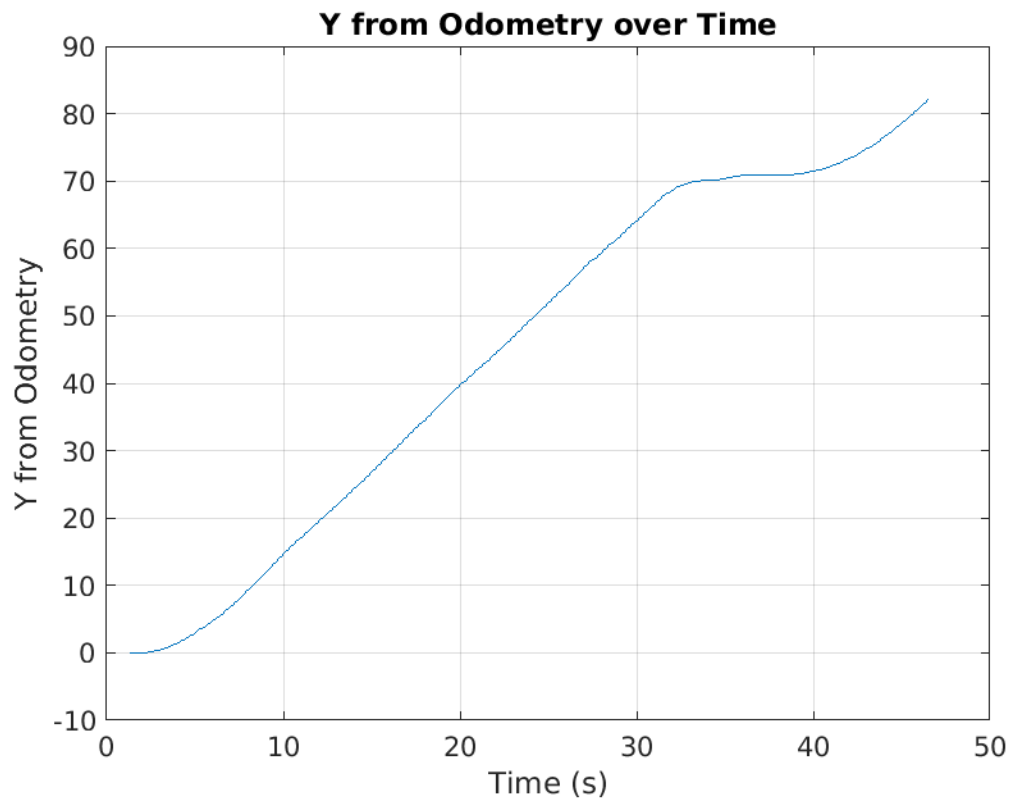
X vs Y



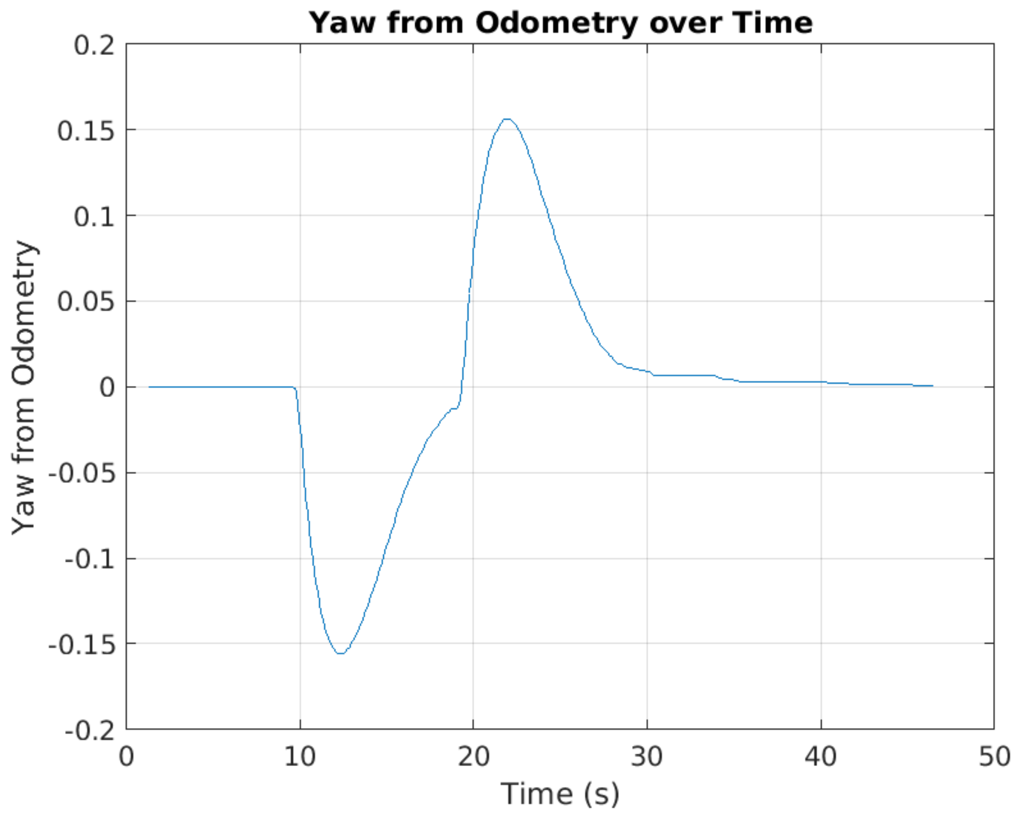
X vs Time


X from Odometry over Time
X from Odometry
Time (s)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
0
10
20
30
40
50

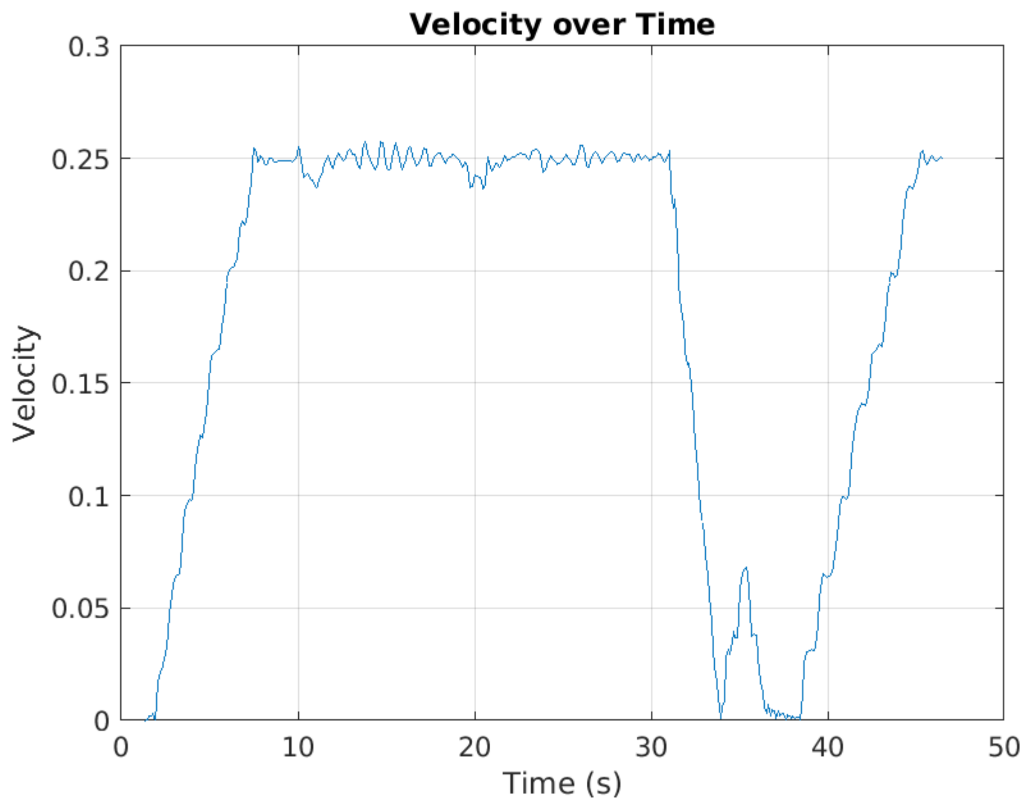
Y vs Time


Y from Odometry over Time
Y from Odometry
Time (s)
90
80
70
60
50
40
30
20
10
0
-10
0
10
20
30
40
50

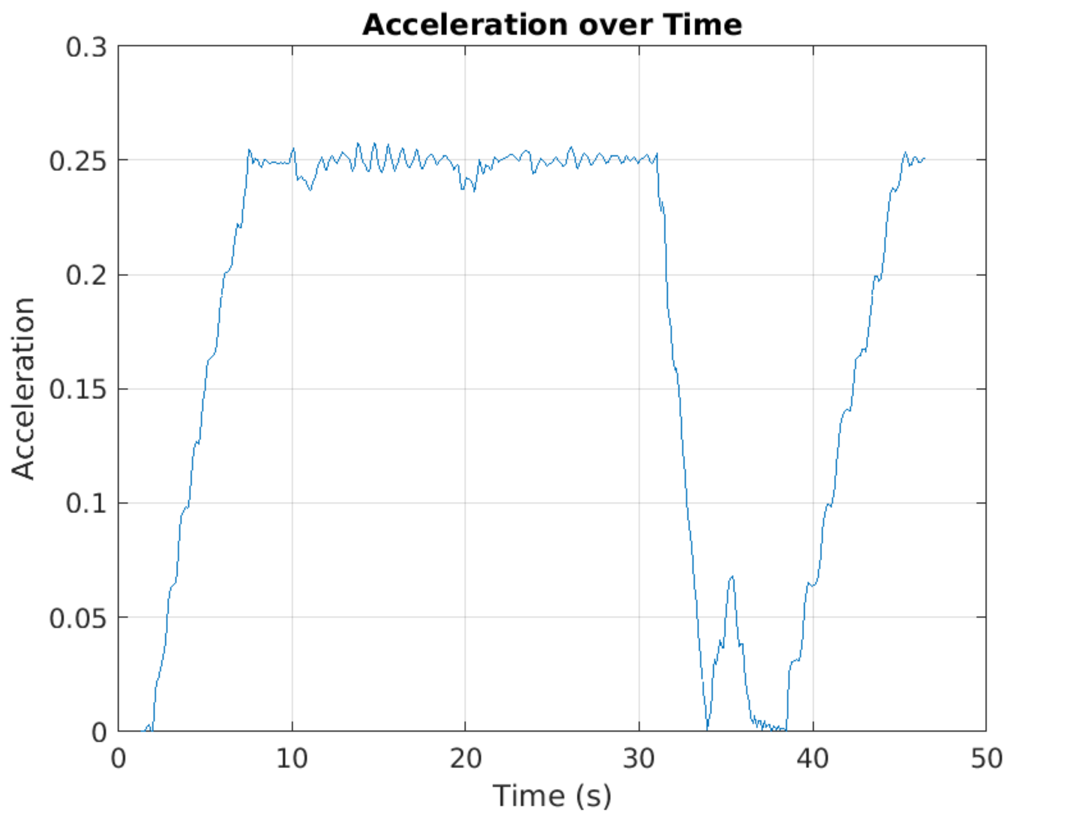
Yaw vs Time



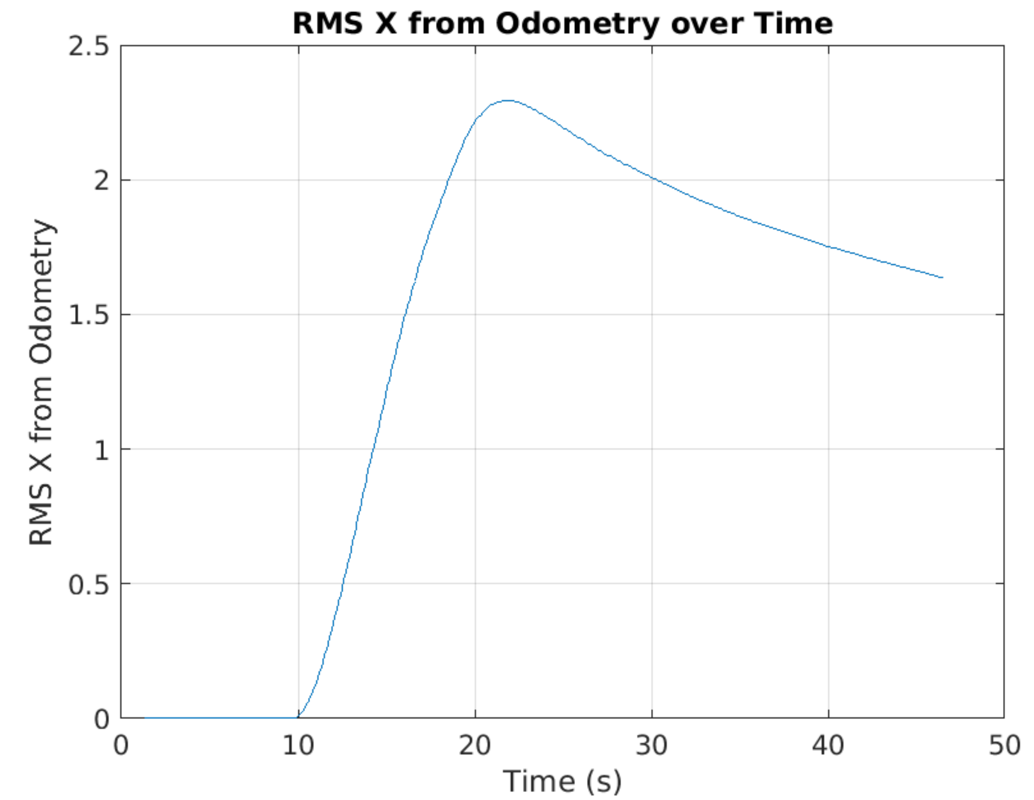
Velocity vs Time



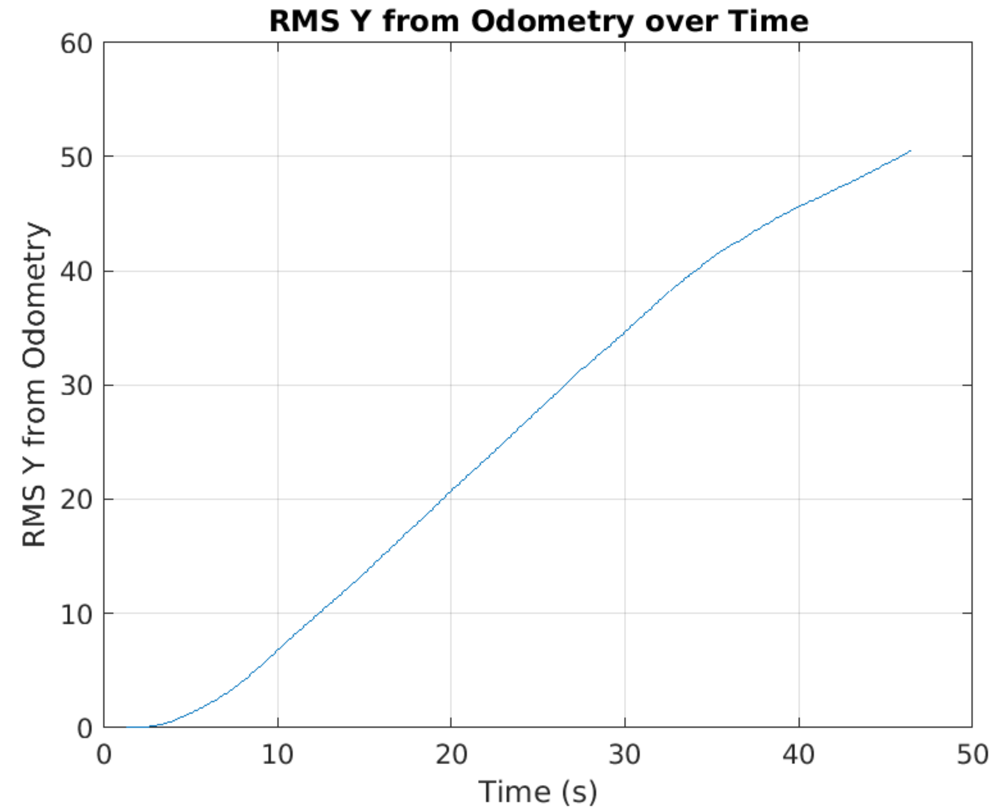
Acceleration vs Time



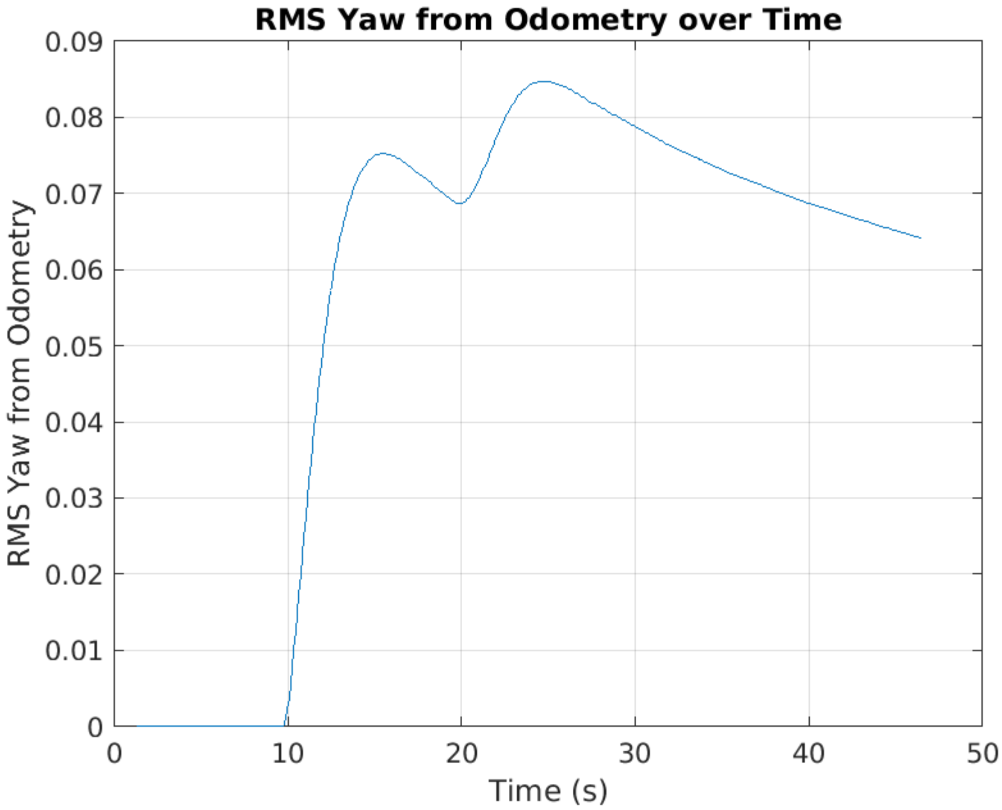
RMS_X vs Time


RMS X from Odometry over Time
RMS X from Odometry
Time (s)
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50

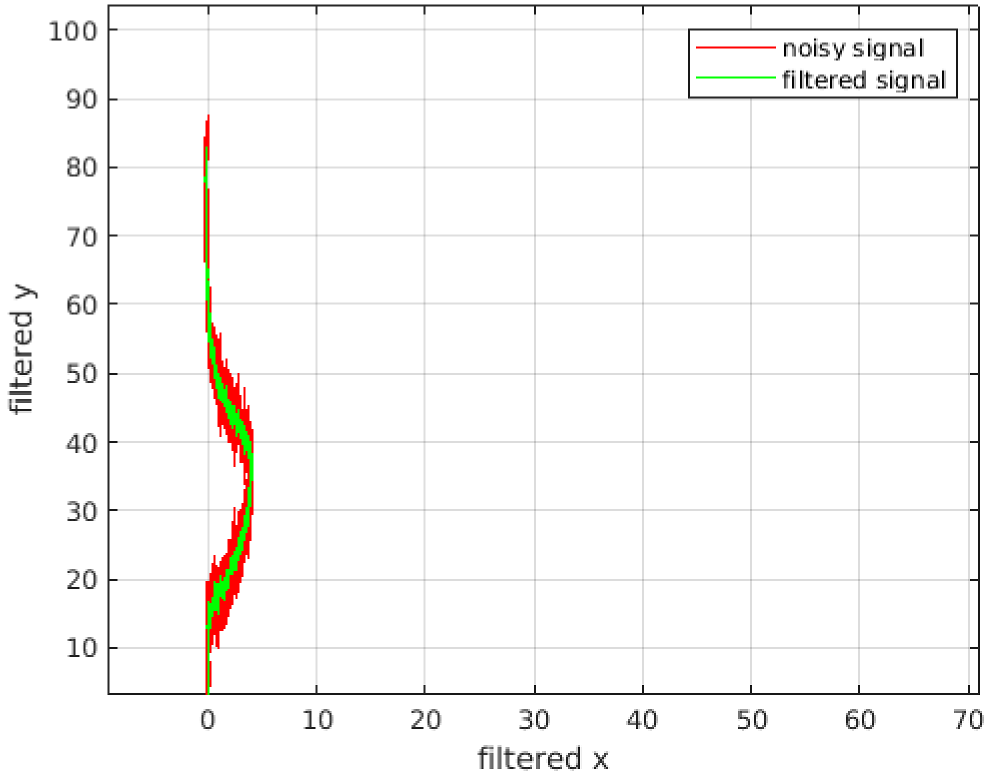
RMS_Y vs Time


RMS Y from Odometry over Time
RMS Y from Odometry
Time (s)
60
50
40
30
20
10
0
0
10
20
30
40
50

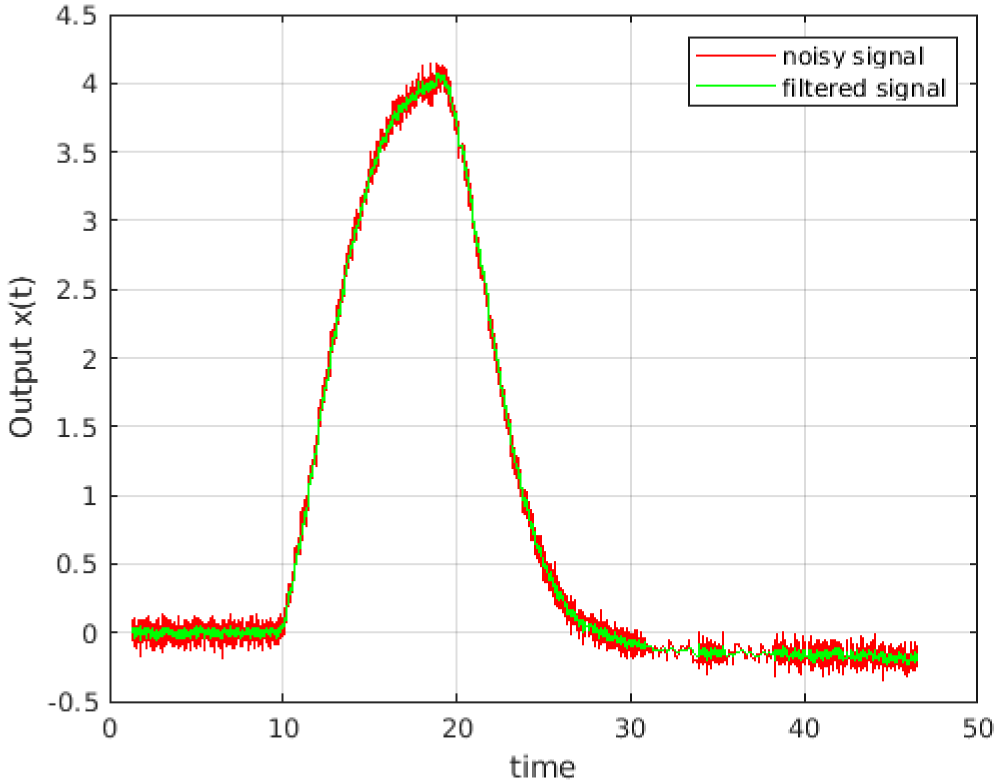
RMS_YAW vs Time



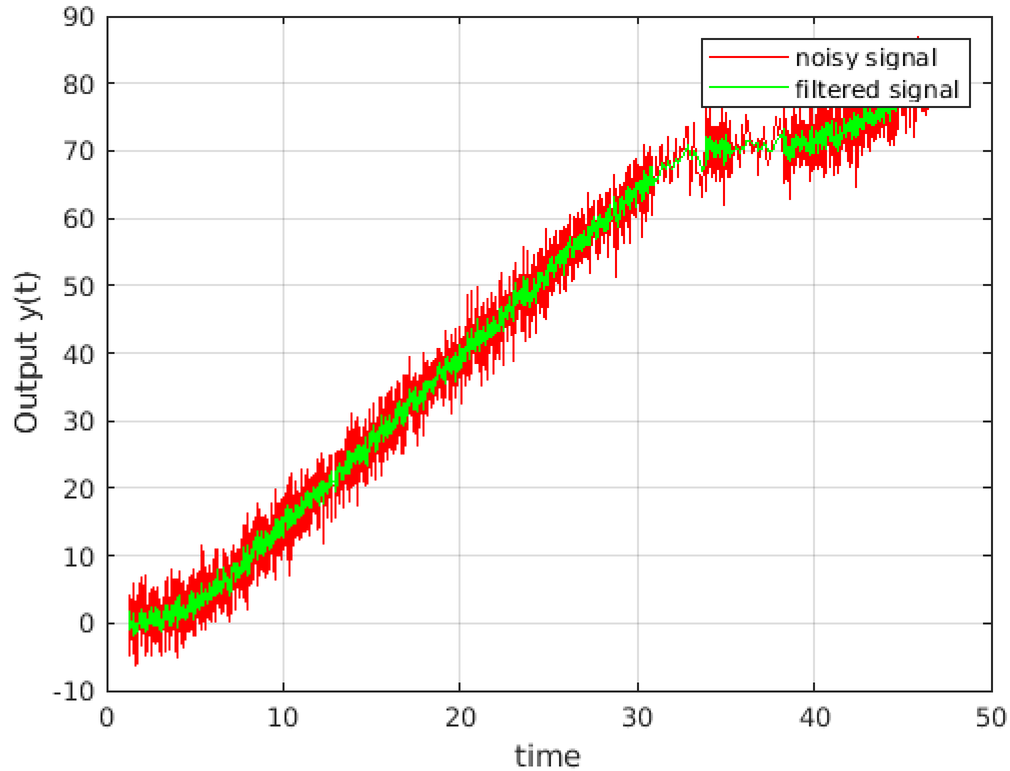
X vs Y filtered



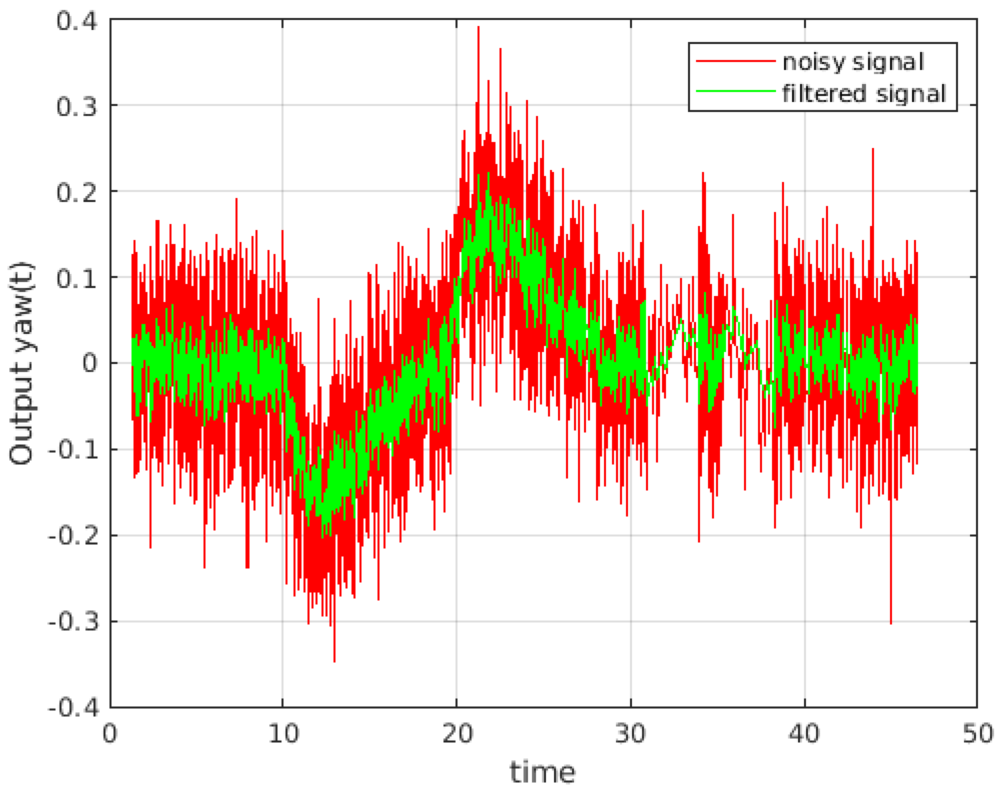
X vs Time filtered



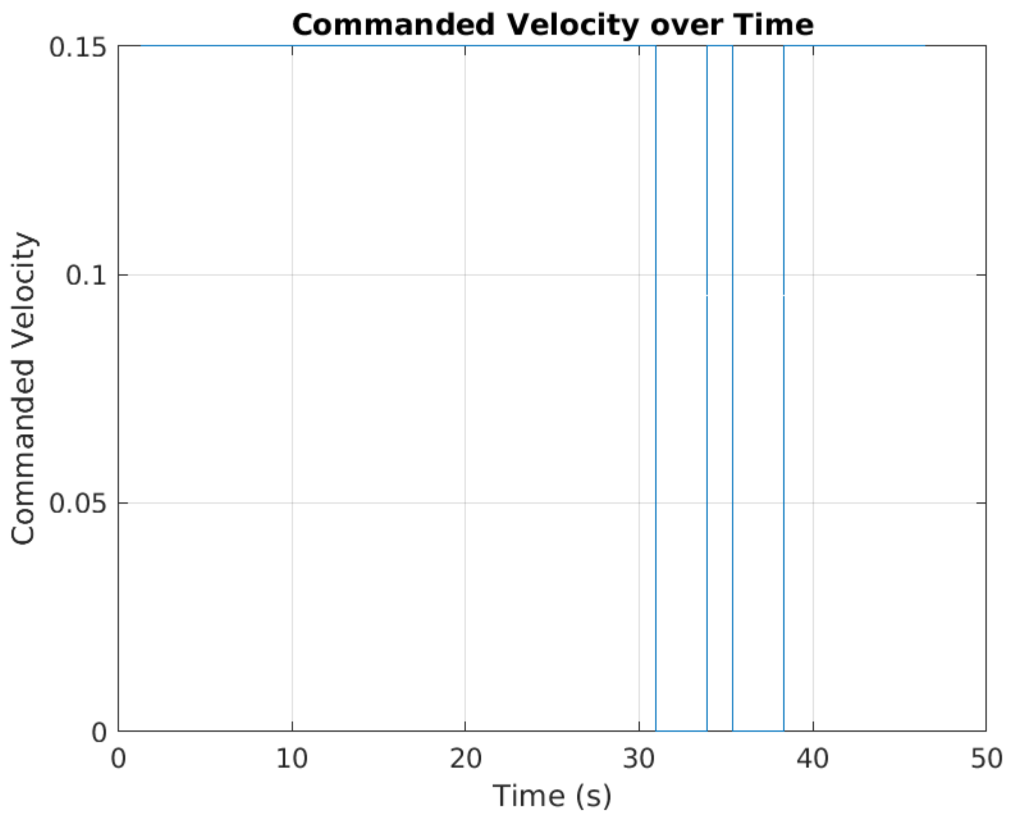
Y vs Time filtered



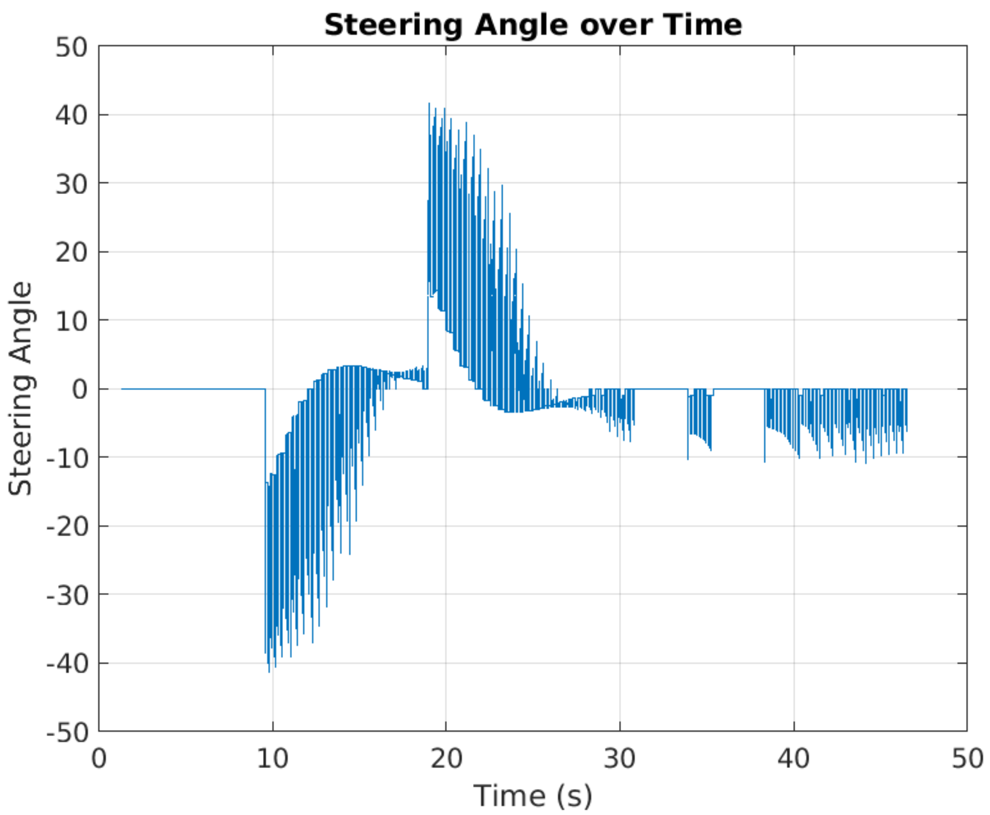
Yaw vs Time filtered



velocity_cmd vs time



steering_angle vs Time



Path3 (Circular path):

RMS values

overall_RMS_x = 8.6372
overall_RMS_y = 7.0777
overall_RMS_yaw = 0.5487

Strengths

- **Real-Time Object Detection and Response:** The system can detect objects in real-time and respond appropriately, ensuring safety.
- **Lane Change Capability:** The vehicle can perform lane changes to avoid obstacles, improving manoeuvrability.
- **Pure Pursuit Algorithm:** The use of the Pure Pursuit algorithm allows for smooth path following.

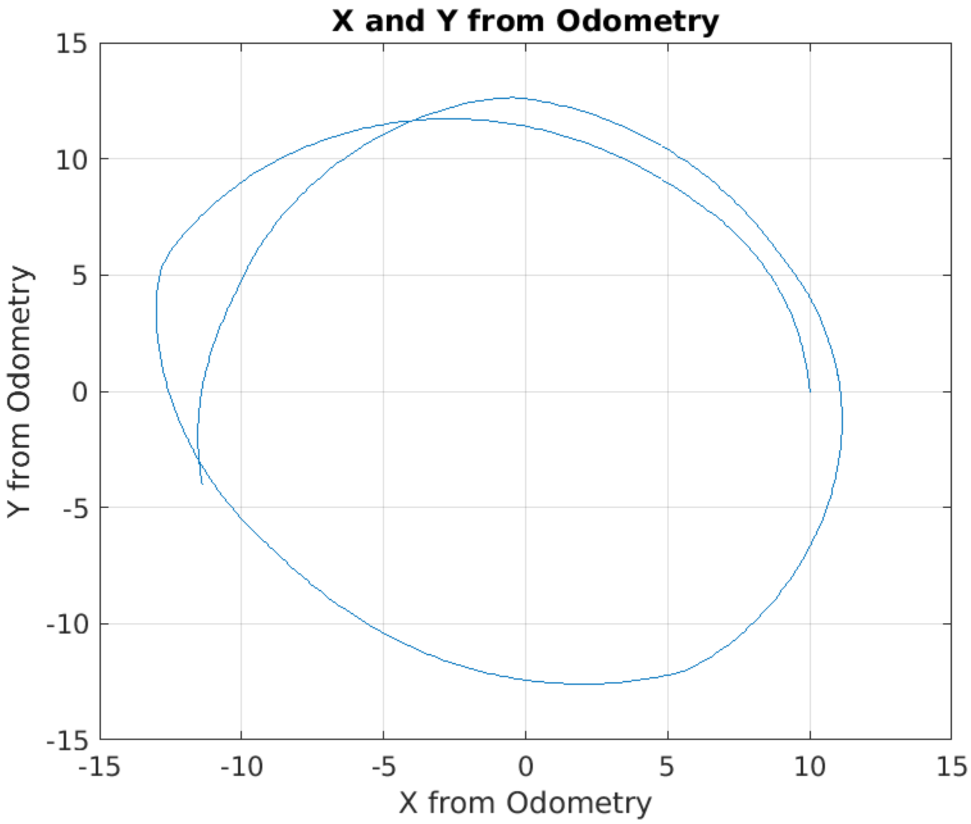
Weaknesses

- **Limited Obstacle Handling:** The system primarily focuses on emergency stops and lane changes, lacking more advanced obstacle avoidance strategies.
- **Fixed Parameters:** The use of fixed parameters for look-ahead distance, speed, and distances for stopping and lane changes may not be optimal for all scenarios.

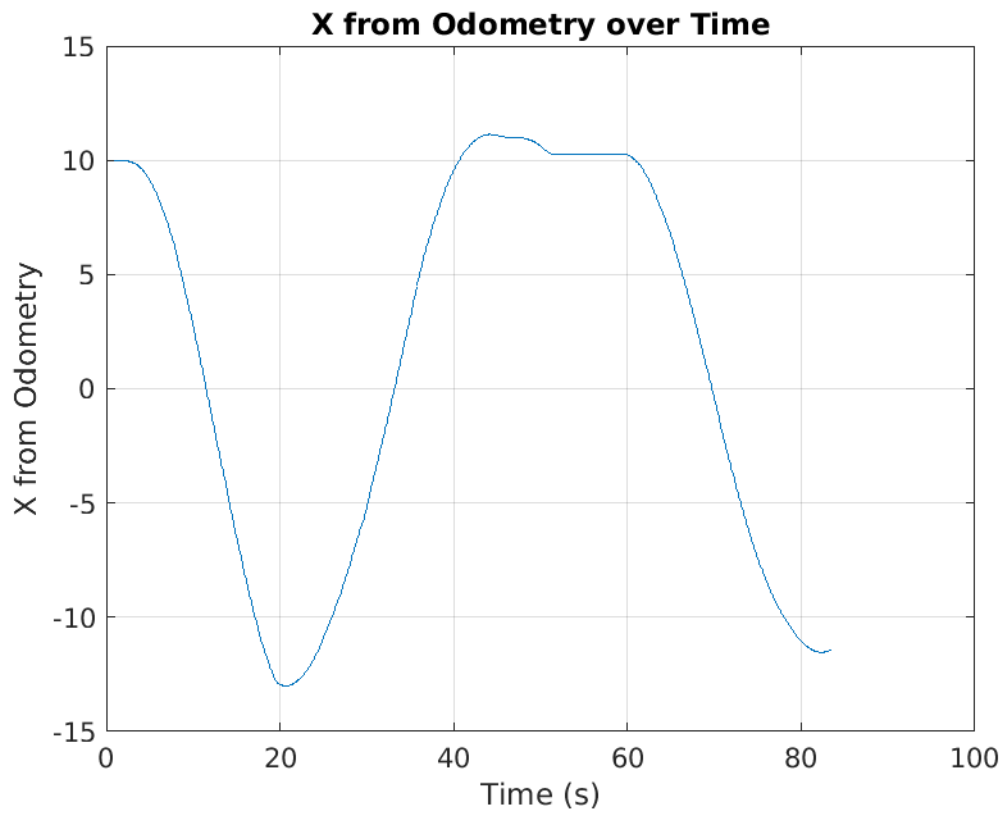
- **Simplified Path Planning:** The path planning is based on predefined waypoints, which may not be sufficient for more complex environments.

Graphs

X vs Y

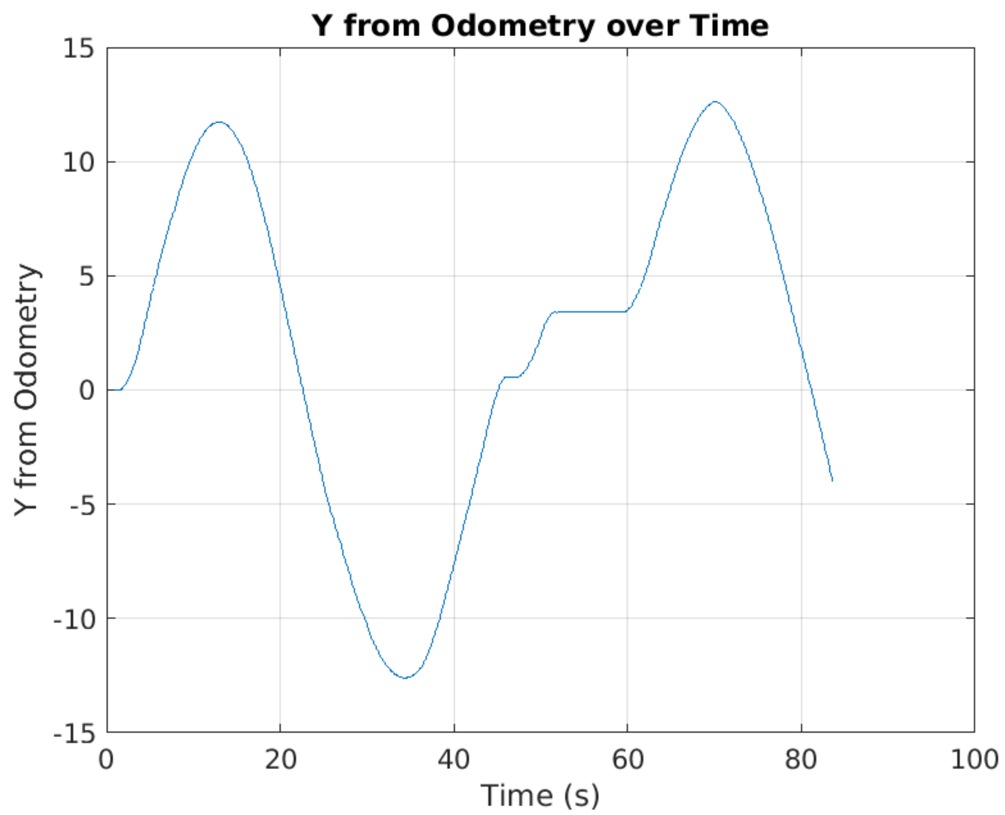


X vs Time



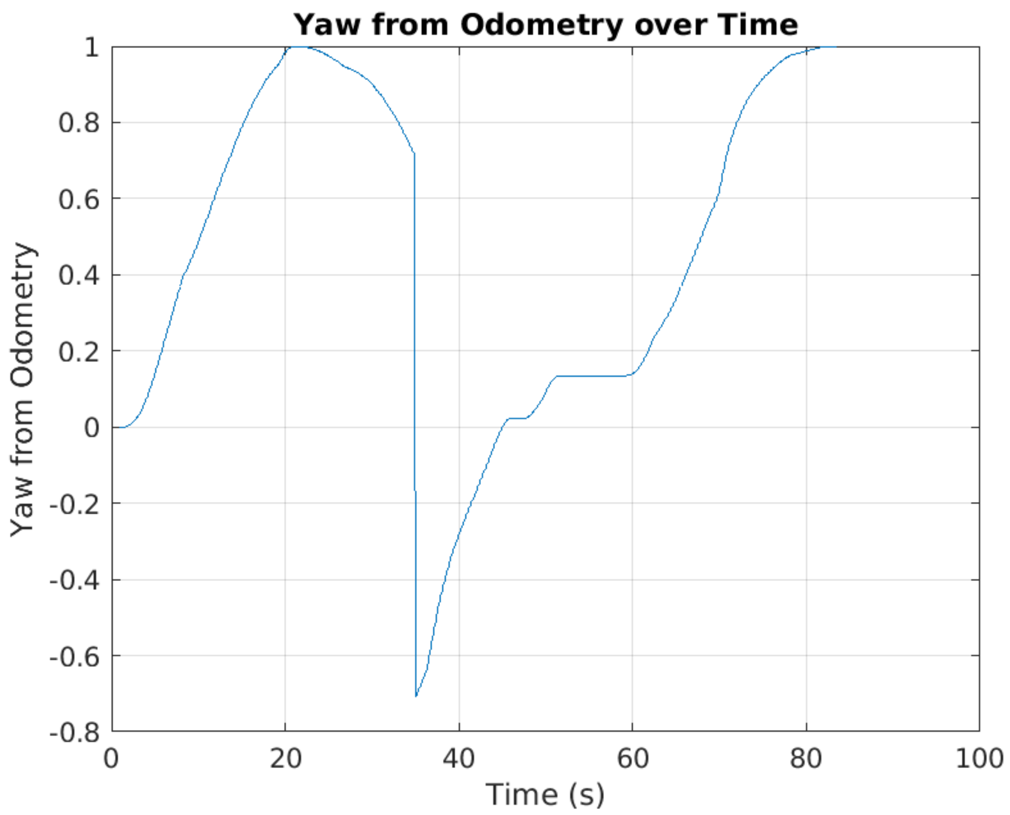
X from Odometry over Time
X from Odometry
Time (s)
15
10
5
0
-5
-10
-15
0
20
40
60
80
100

Y vs Time

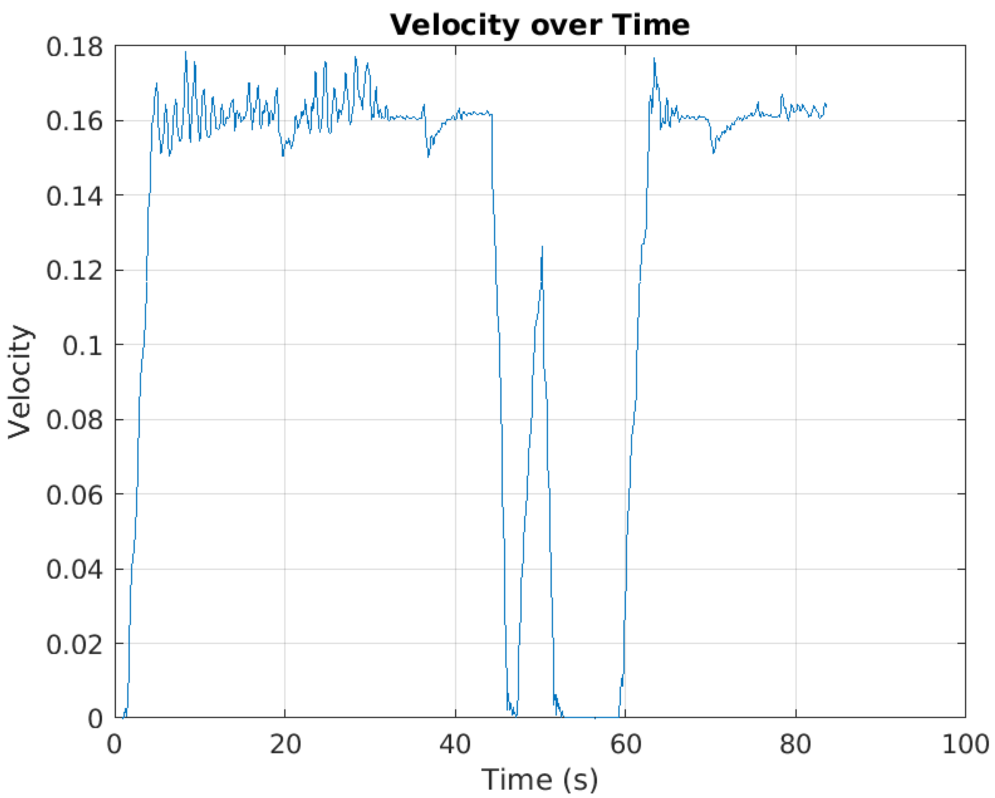


Y from Odometry over Time
Y from Odometry
Time (s)
15
10
5
0
-5
-10
-15
0
20
40
60
80
100

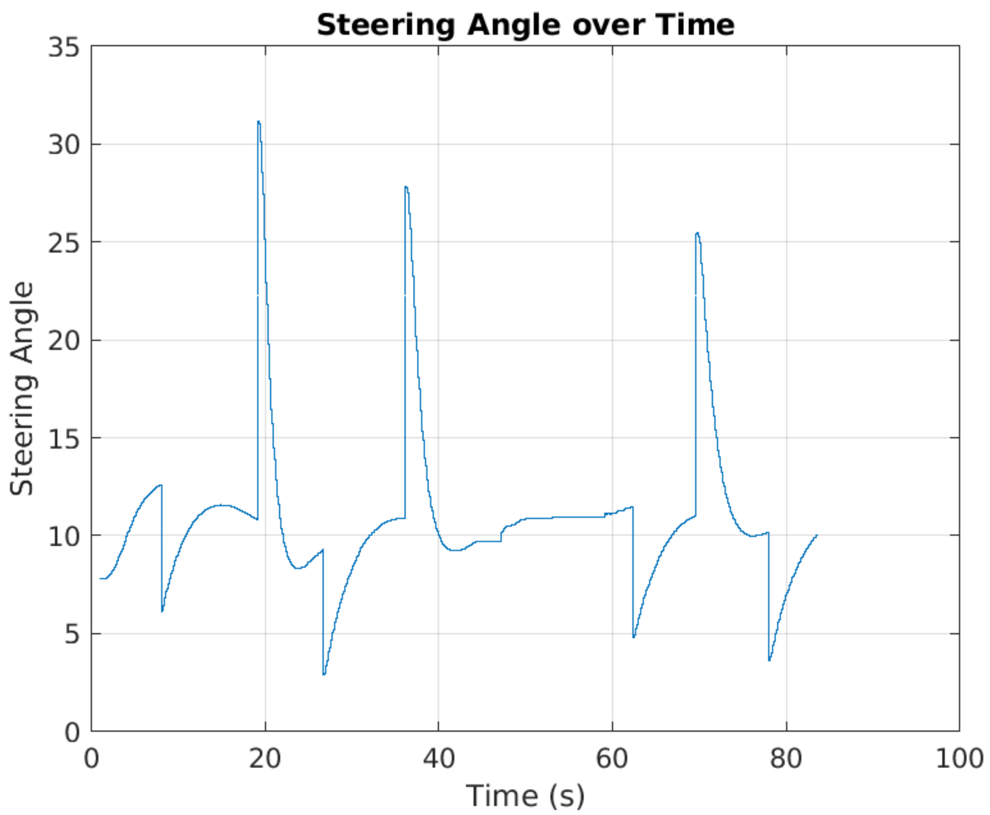
Yaw vs Time



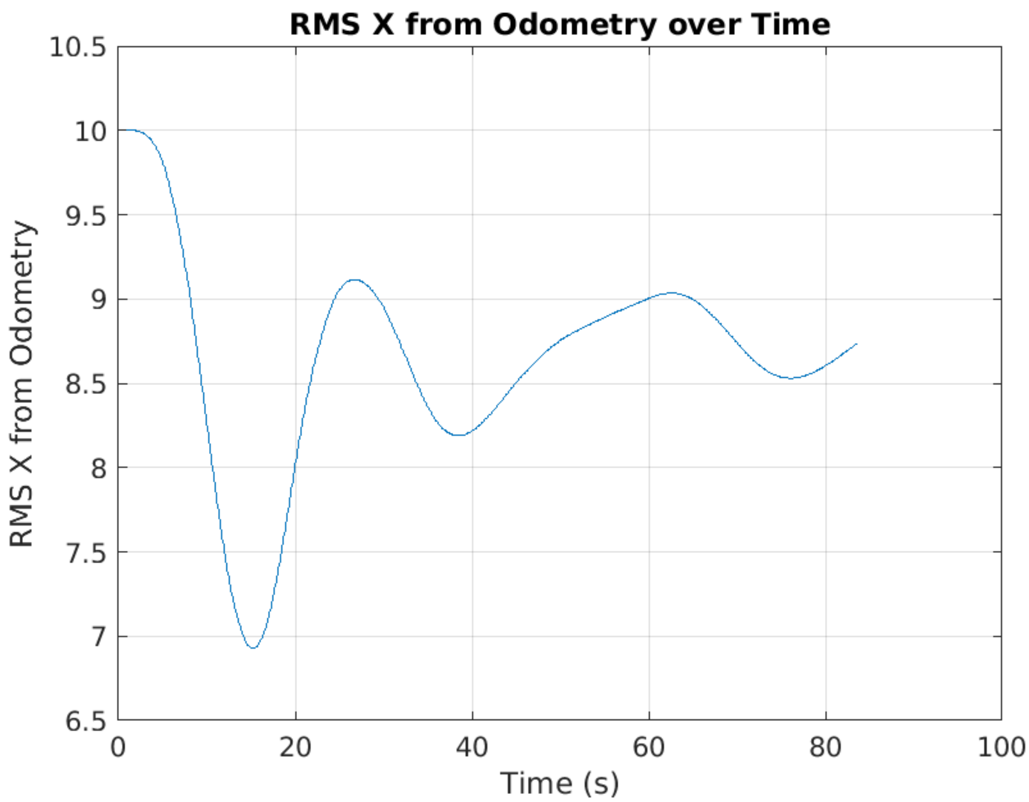
Velocity vs Time



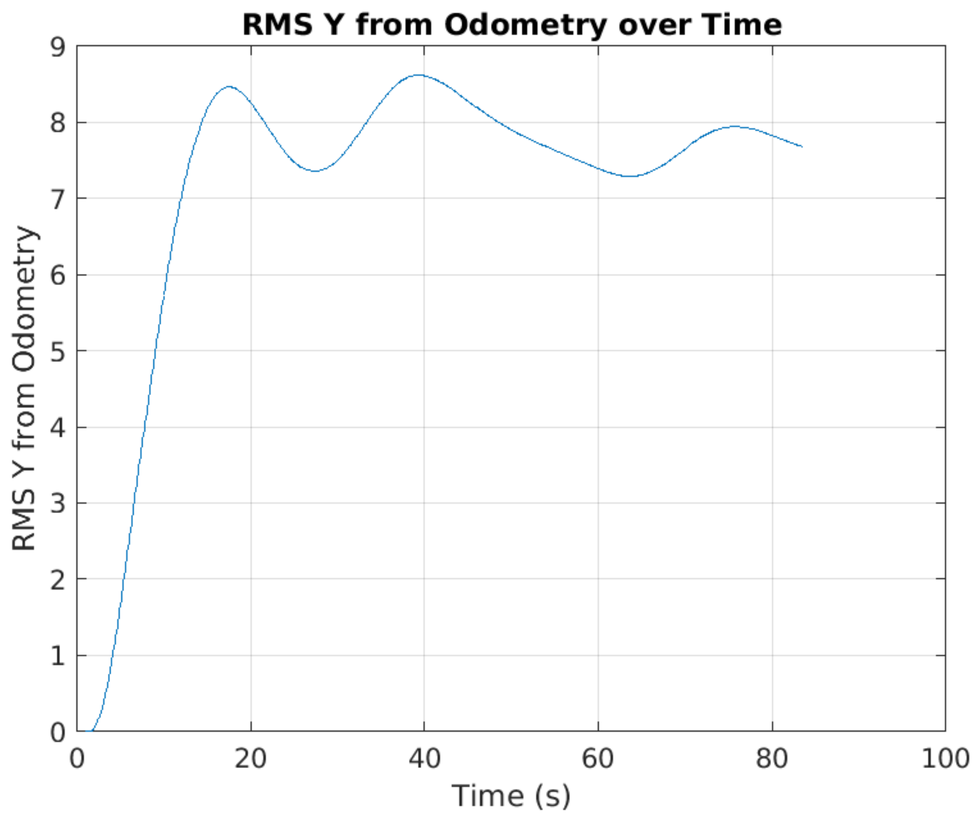
Acceleration vs Time



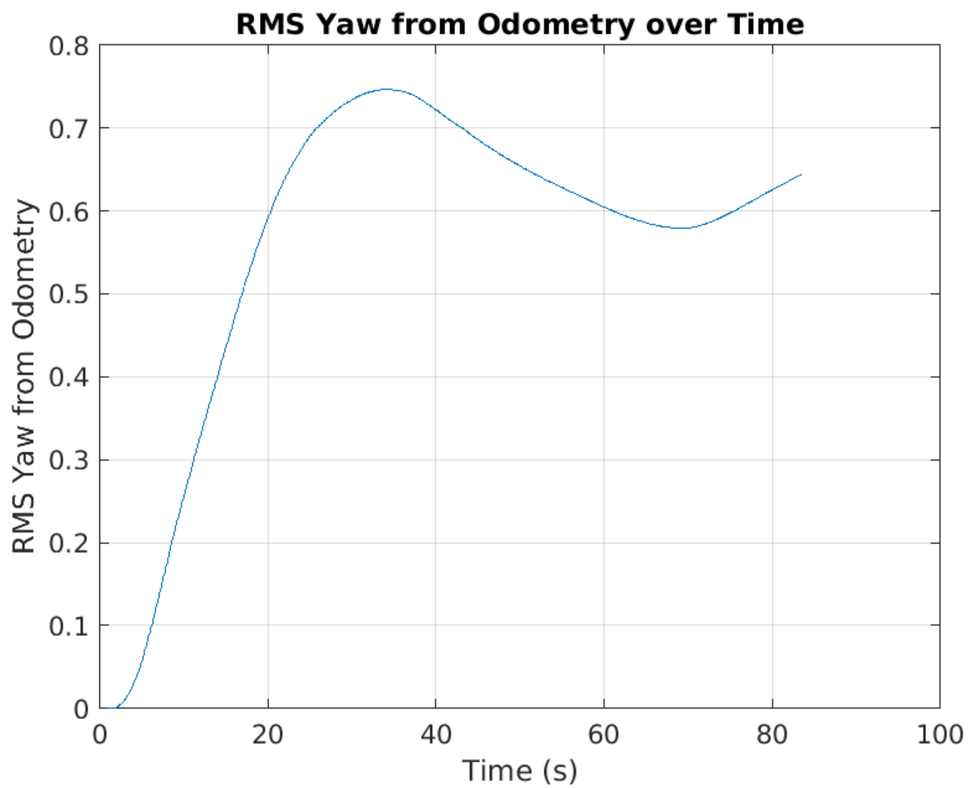
RMS_X vs Time



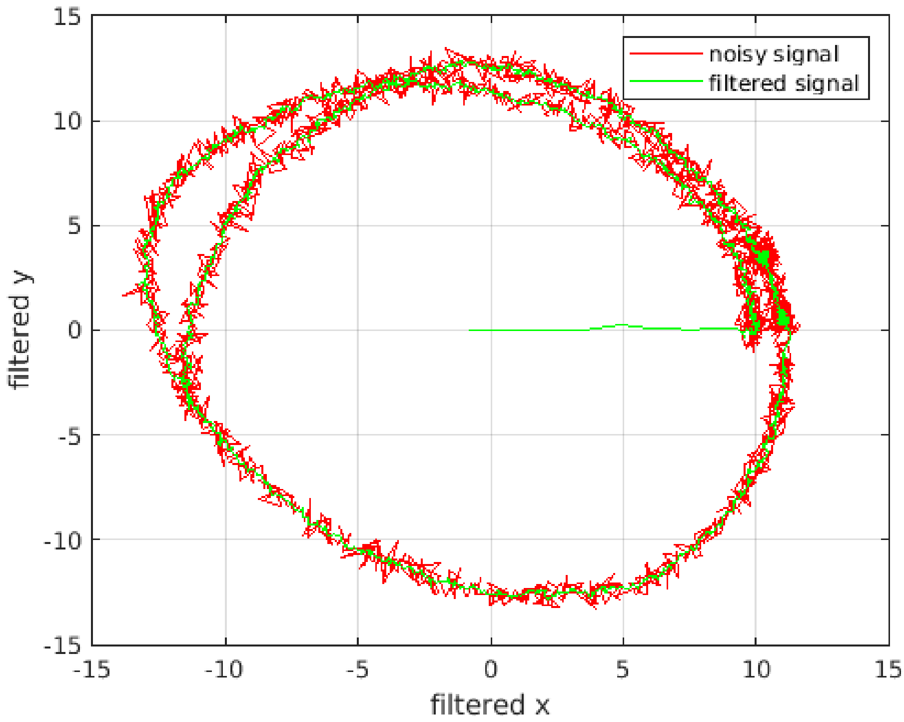
RMS_Y vs Time



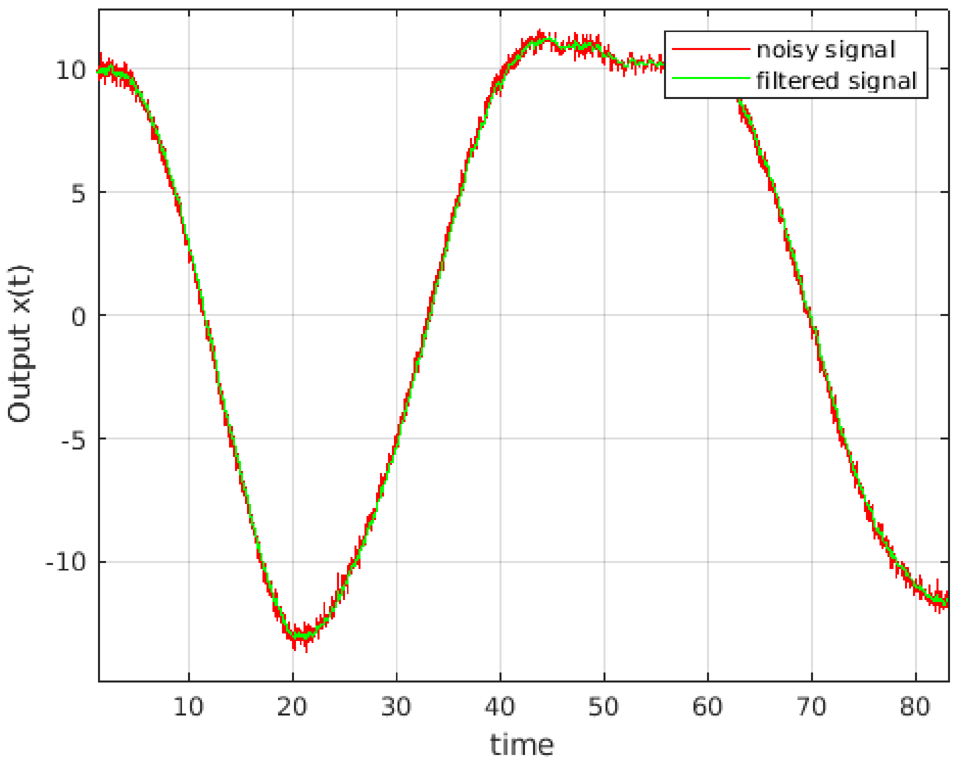
RMS_YAW vs Time



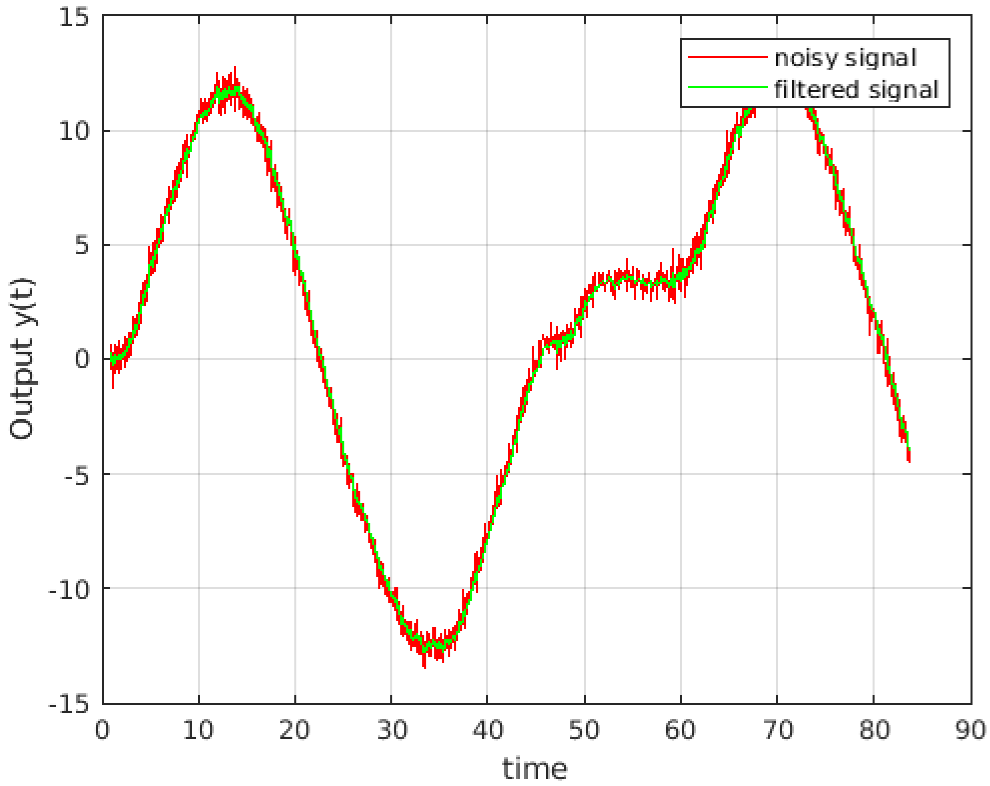
X vs Y filtered



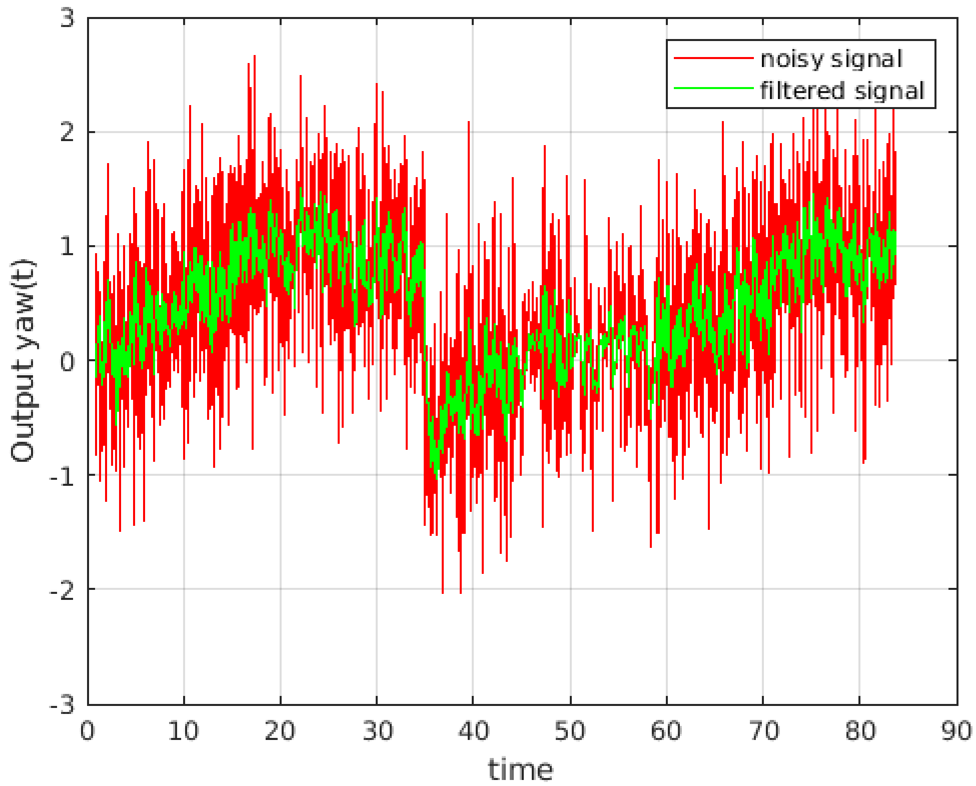
X vs Time filtered



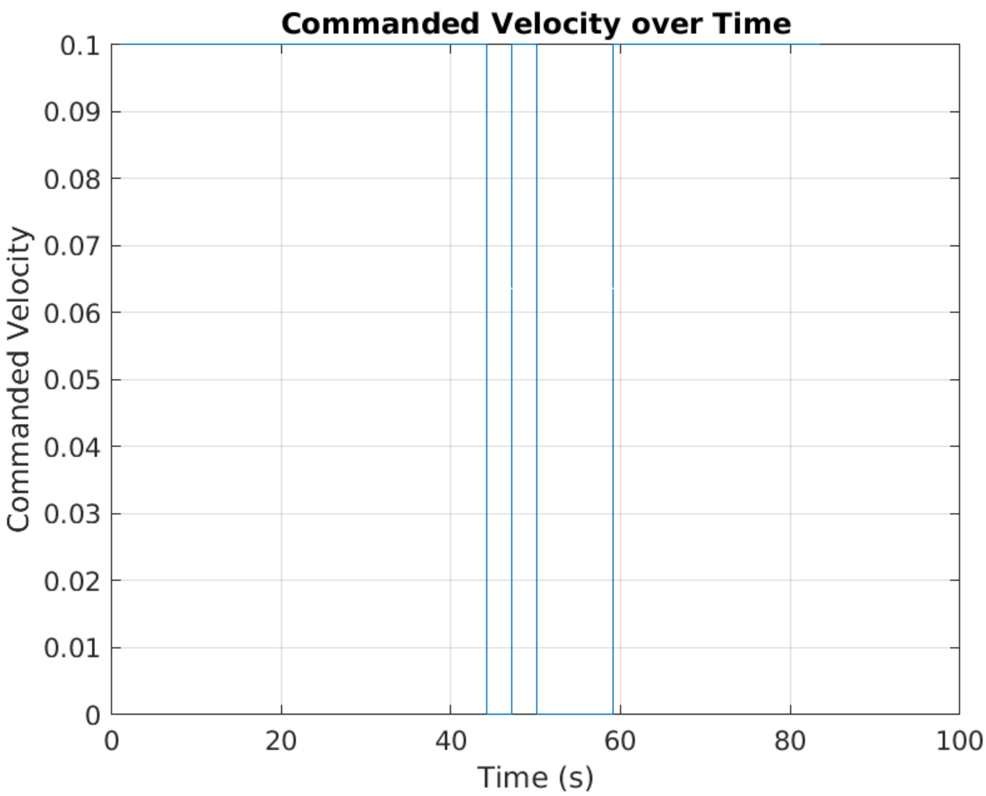
Y vs Time filtered



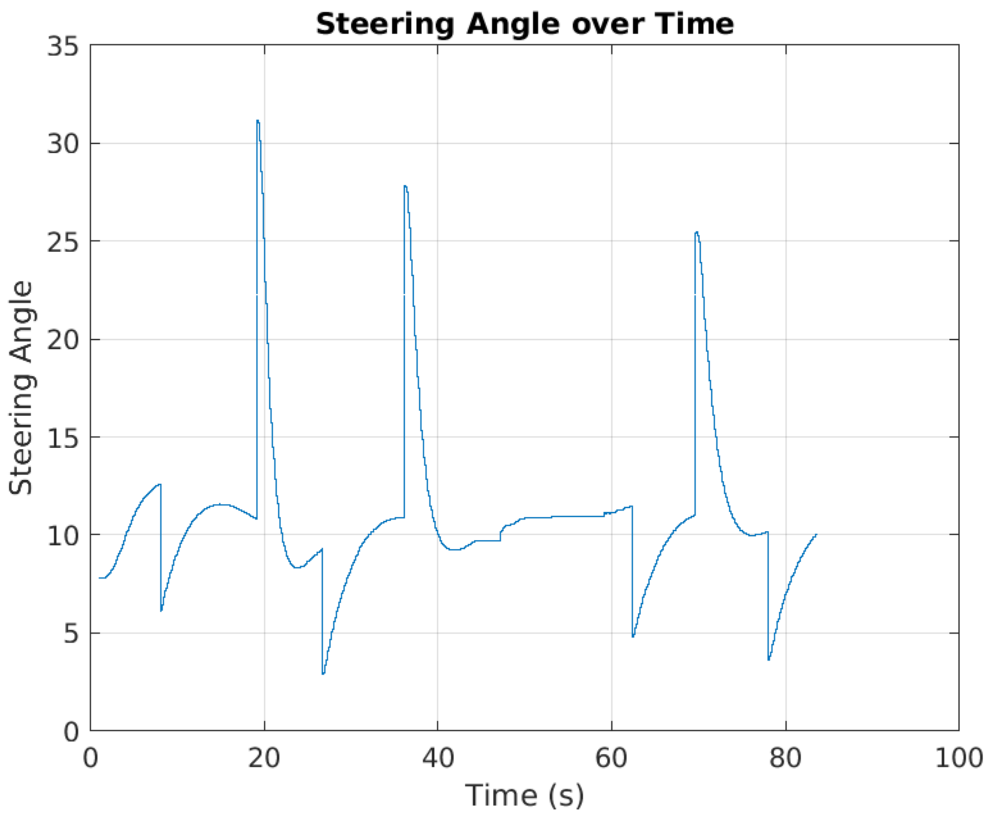
Yaw vs Time filtered



velocity_cmd vs time



steering_angle vs Time



Custom Track:

RMS Values:

overall_RMS_x = 35.6080
overall_RMS_y =16.6426
overall_RMS_yaw =0.6911

Strengths

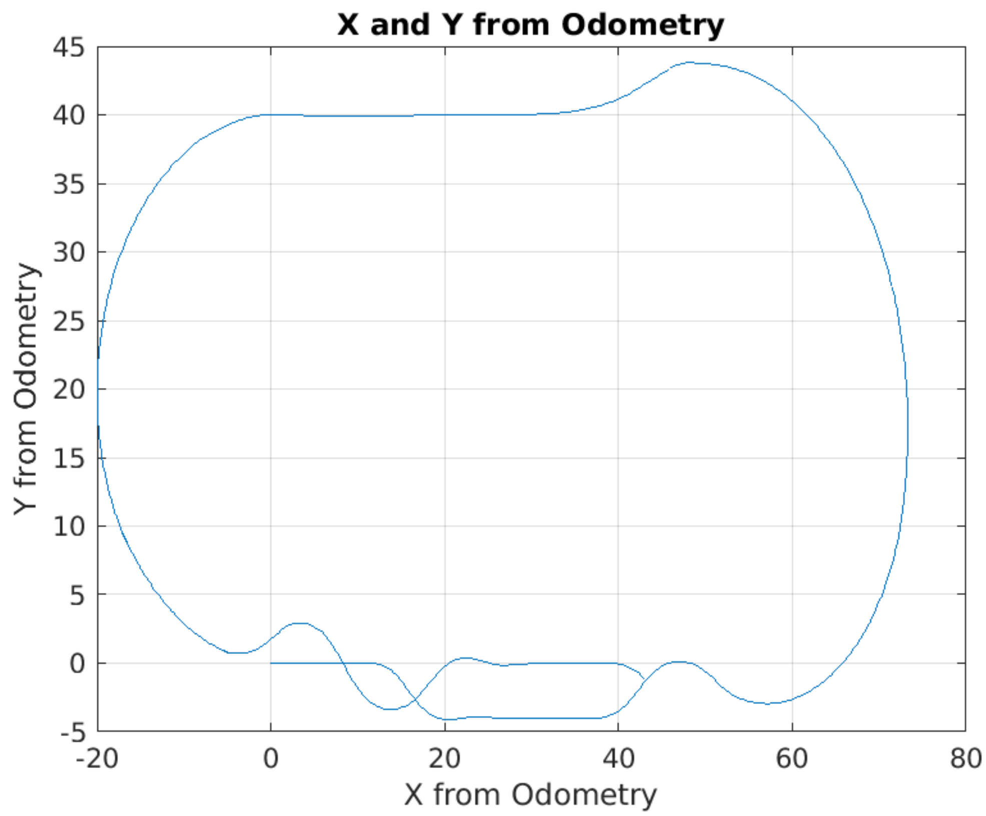
- **Real-Time Object Detection and Response:** The system can detect objects in real-time and respond appropriately, ensuring safety.
- **Lane Change Capability:** The vehicle can perform lane changes to avoid obstacles, improving manoeuvrability.
- **Pure Pursuit Algorithm:** The use of the Pure Pursuit algorithm allows for smooth path following.

Weaknesses

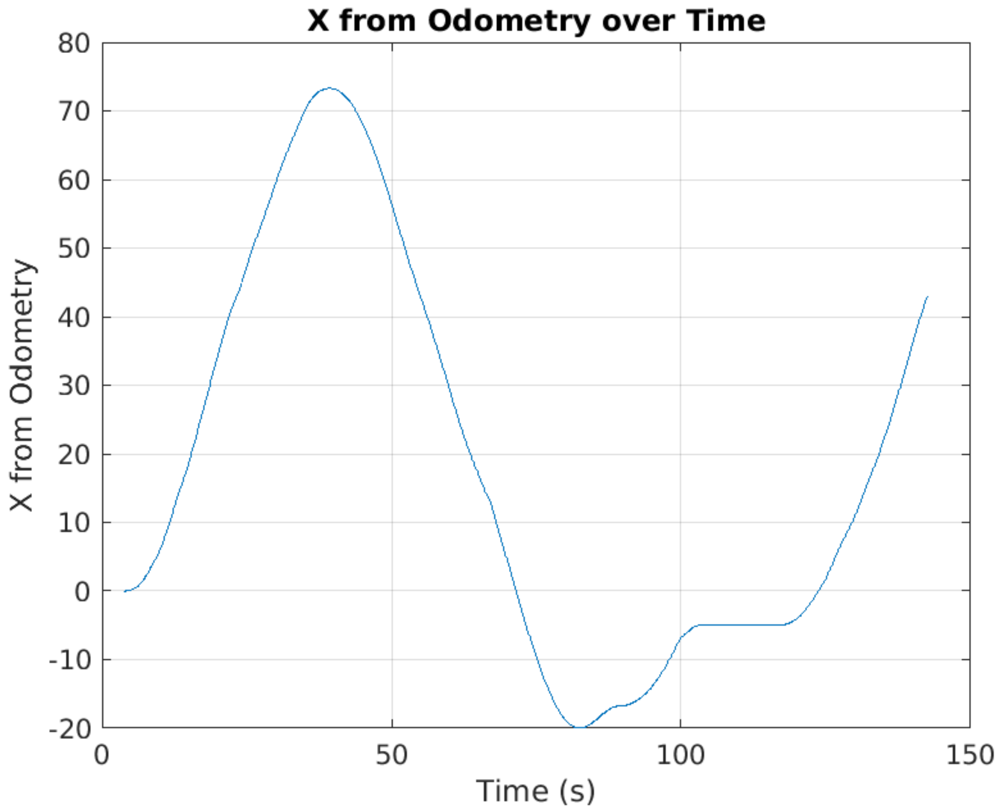
- **Limited Obstacle Handling:** The system primarily focuses on emergency stops and lane changes, lacking more advanced obstacle avoidance strategies.
- **Fixed Parameters:** The use of fixed parameters for look-ahead distance, speed, and distances for stopping and lane changes may not be optimal for all scenarios.
- **Simplified Path Planning:** The path planning is based on predefined waypoints, which may not be sufficient for more complex environments.

Graphs:

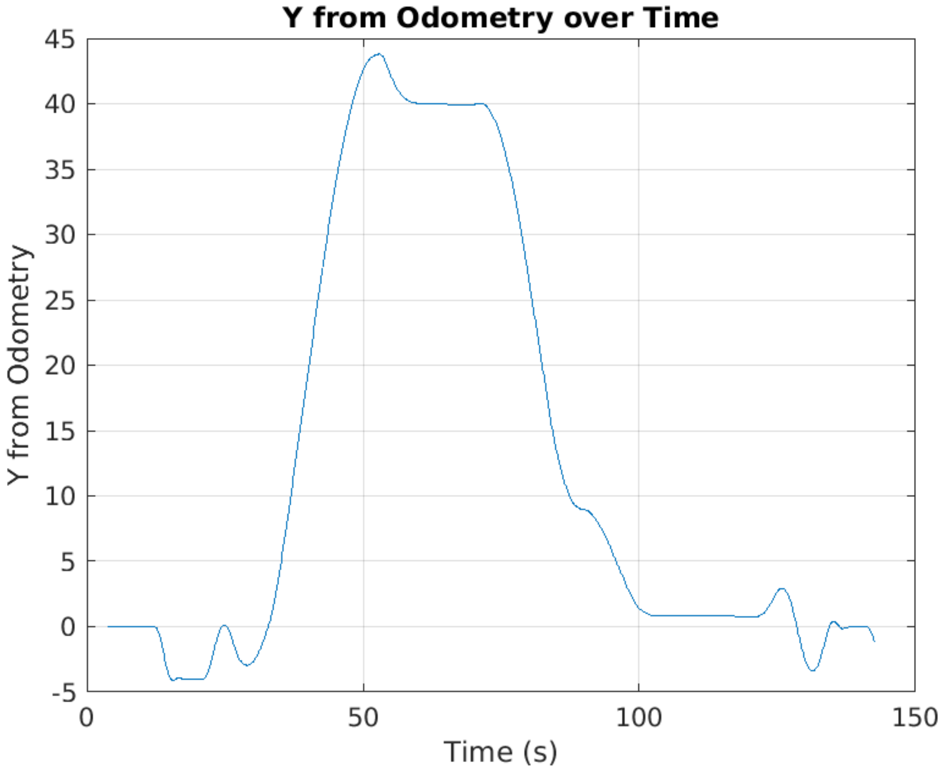
X vs Y


X and Y from Odometry
Y from Odometry
X from Odometry
45
40
35
30
25
20
15
10
5
0
-5
-20
0
20
40
60
80

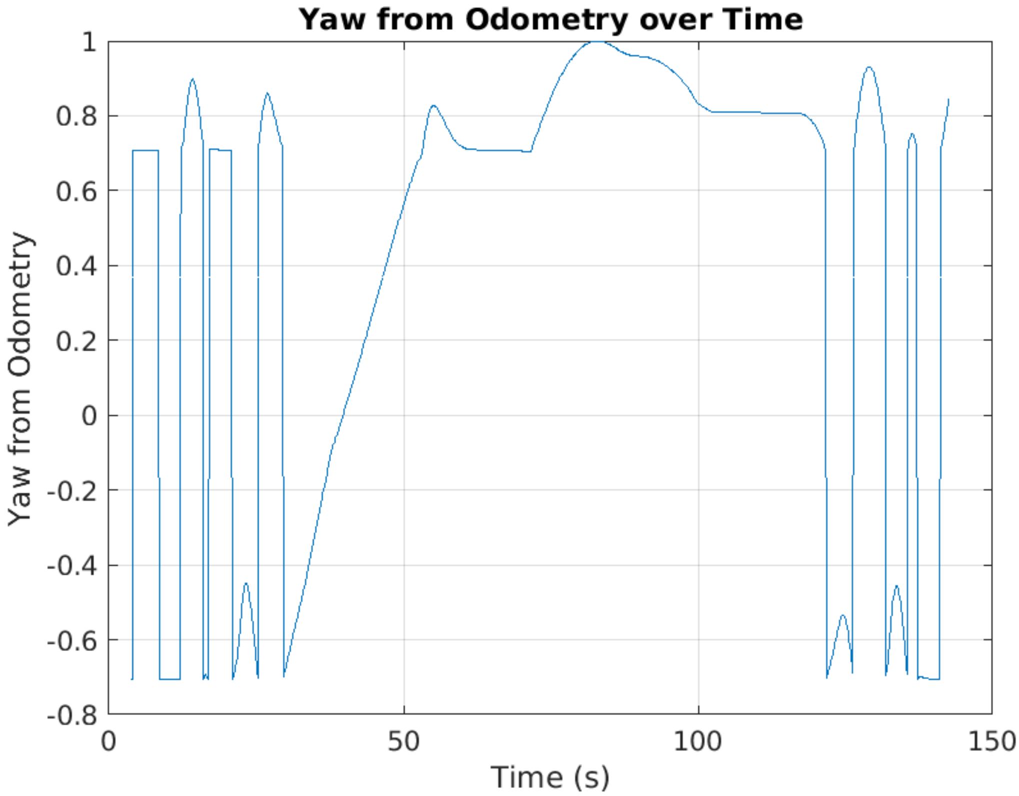
X vs Time


X from Odometry over Time
X from Odometry
Time (s)
80
70
60
50
40
30
20
10
0
-10
-20
0
50
100
150

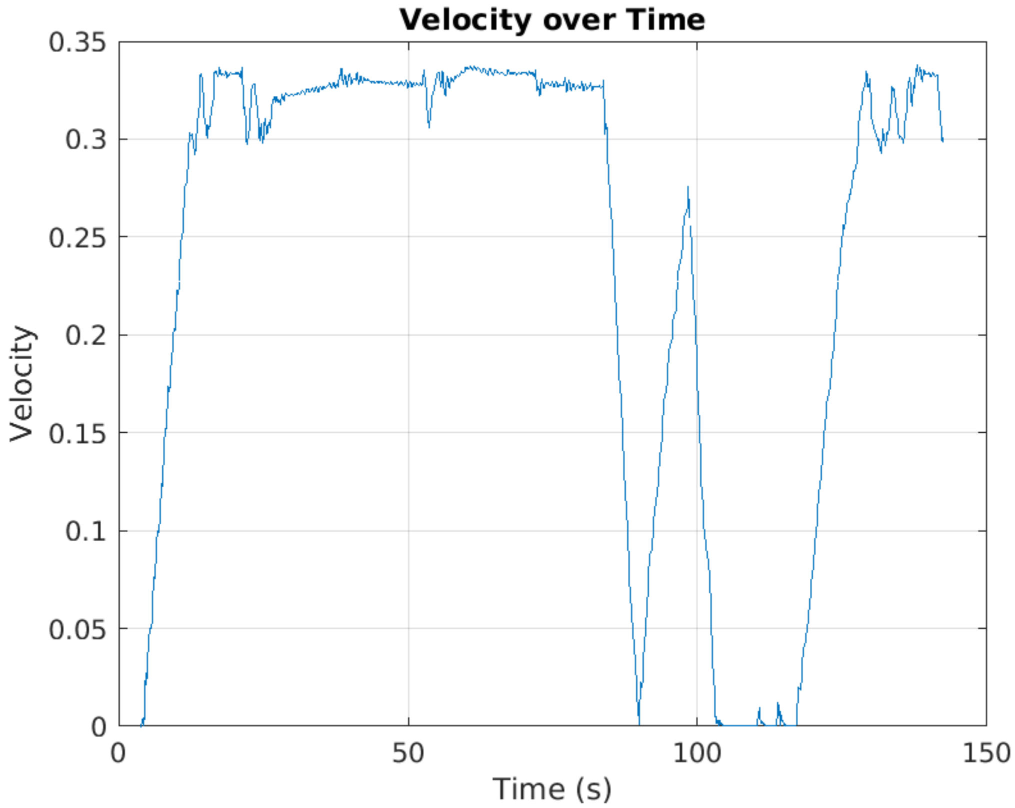
Y vs Time



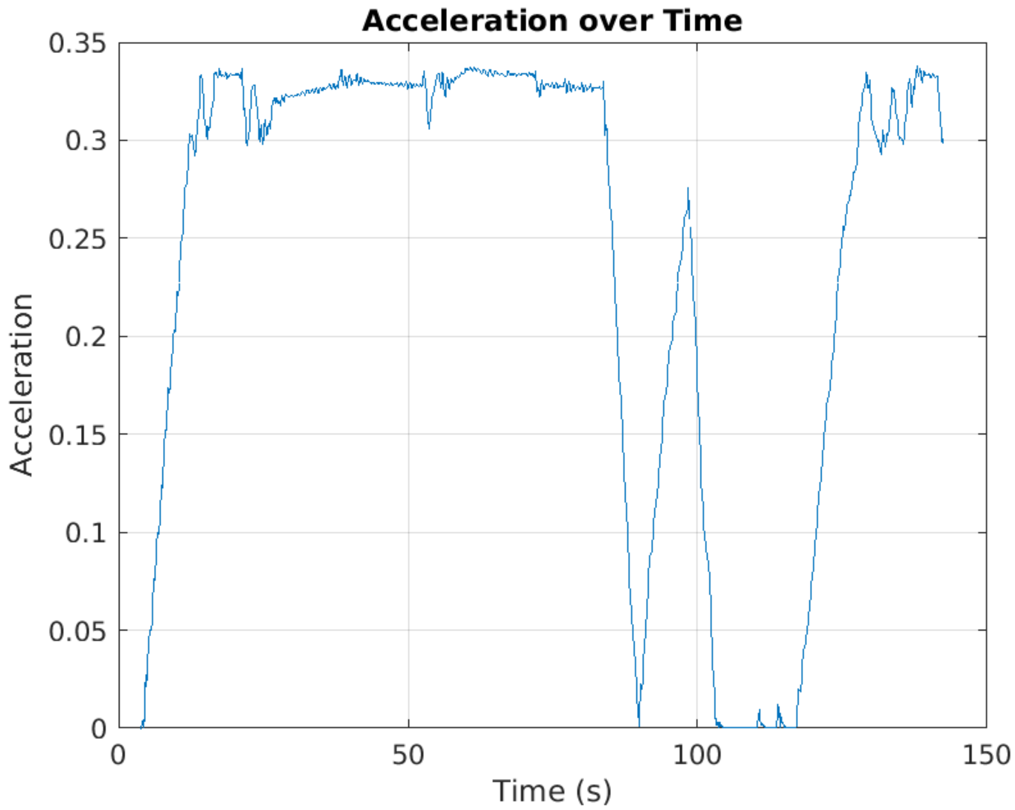
Yaw vs Time



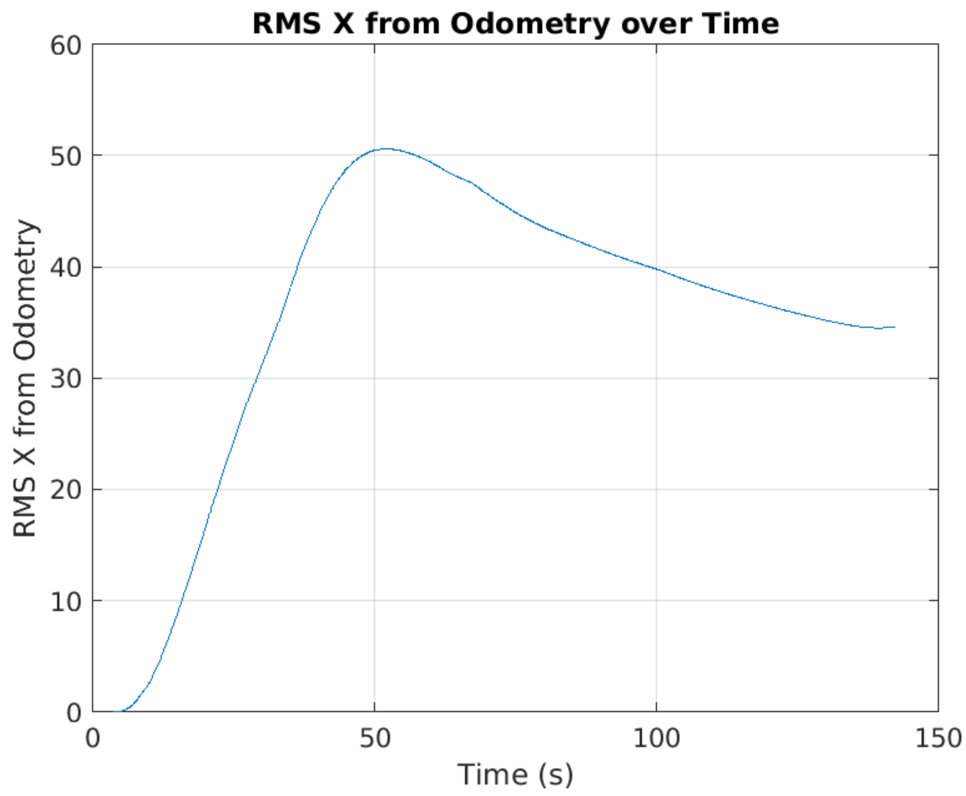
Velocity vs Time



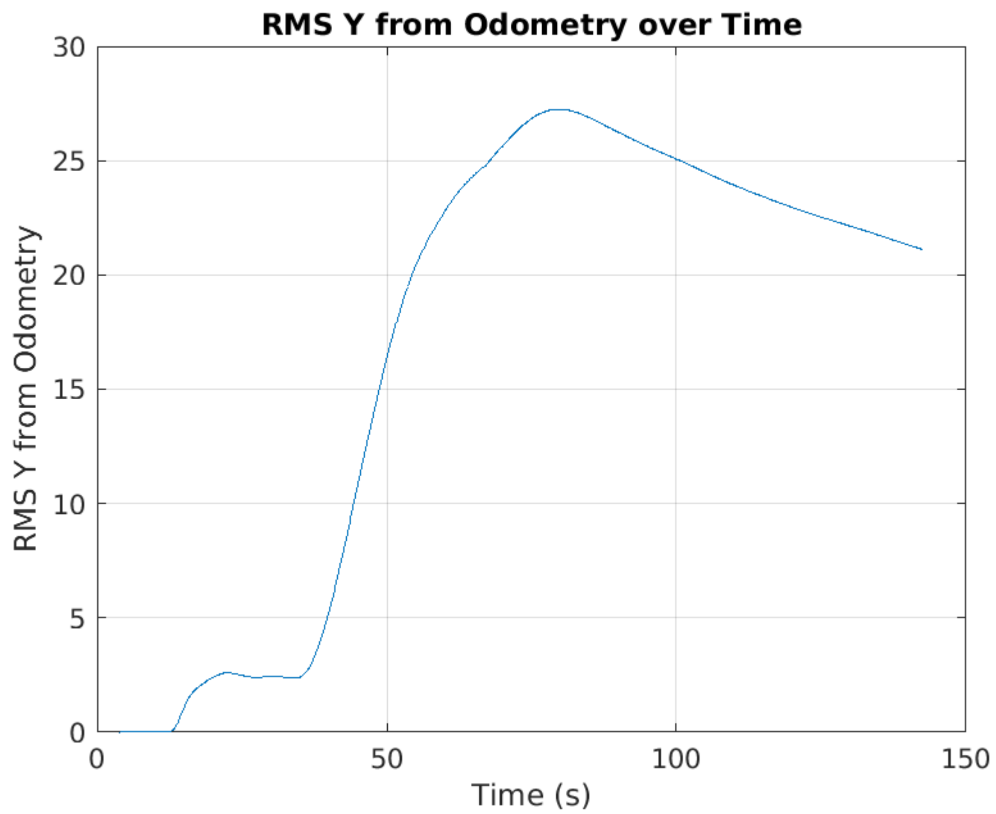
Acceleration vs Time



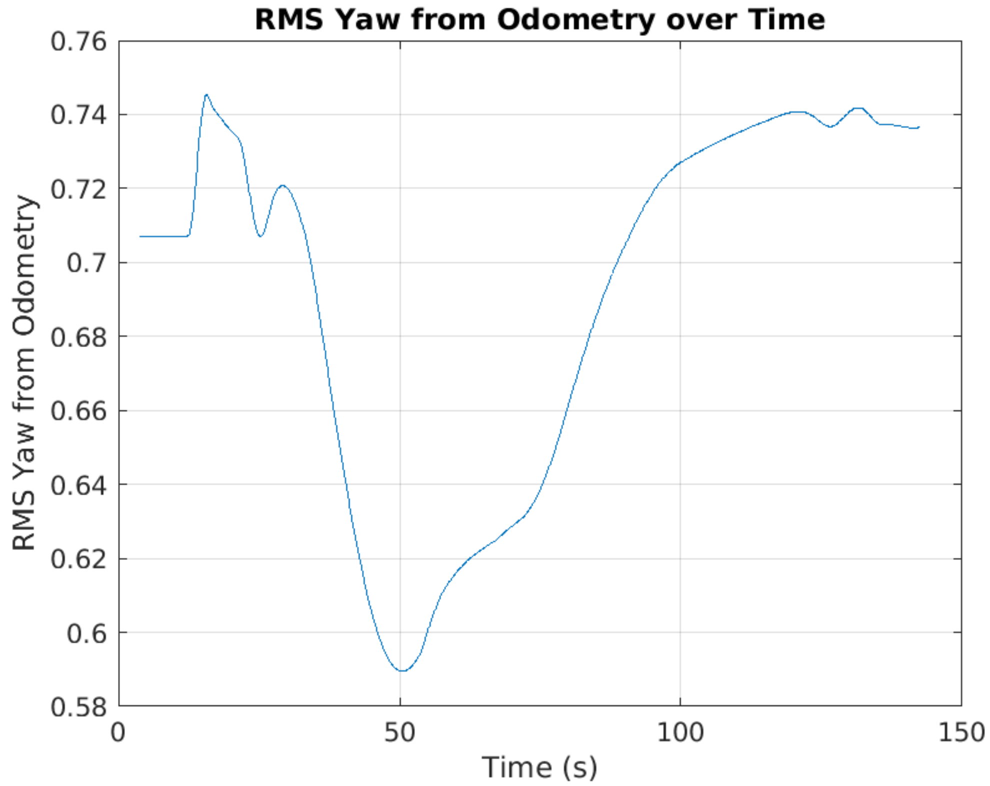
RMS_X vs Time



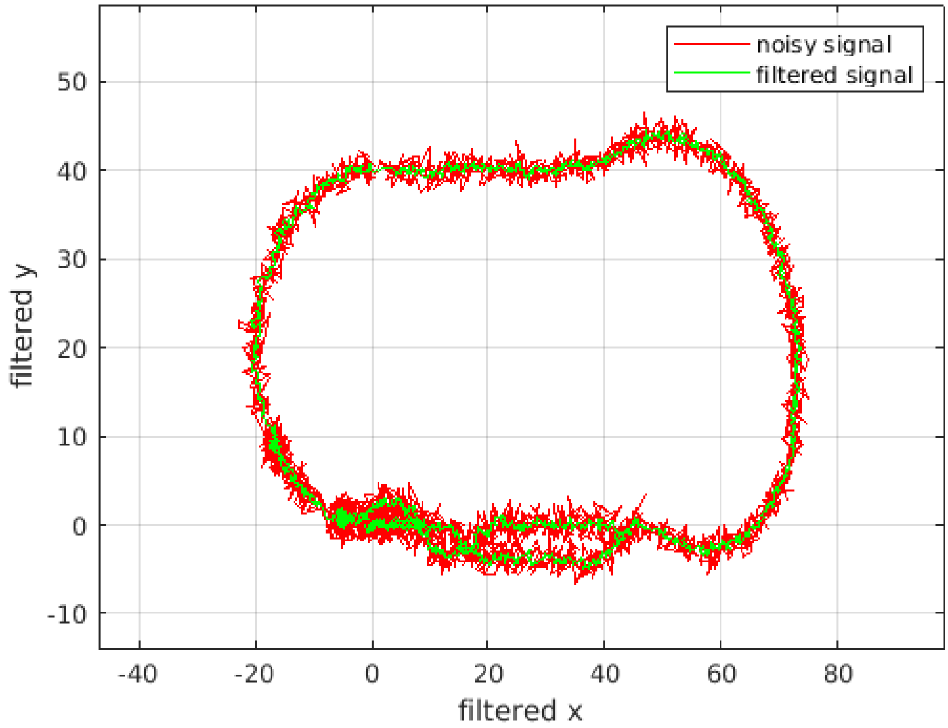
RMS_Y vs Time



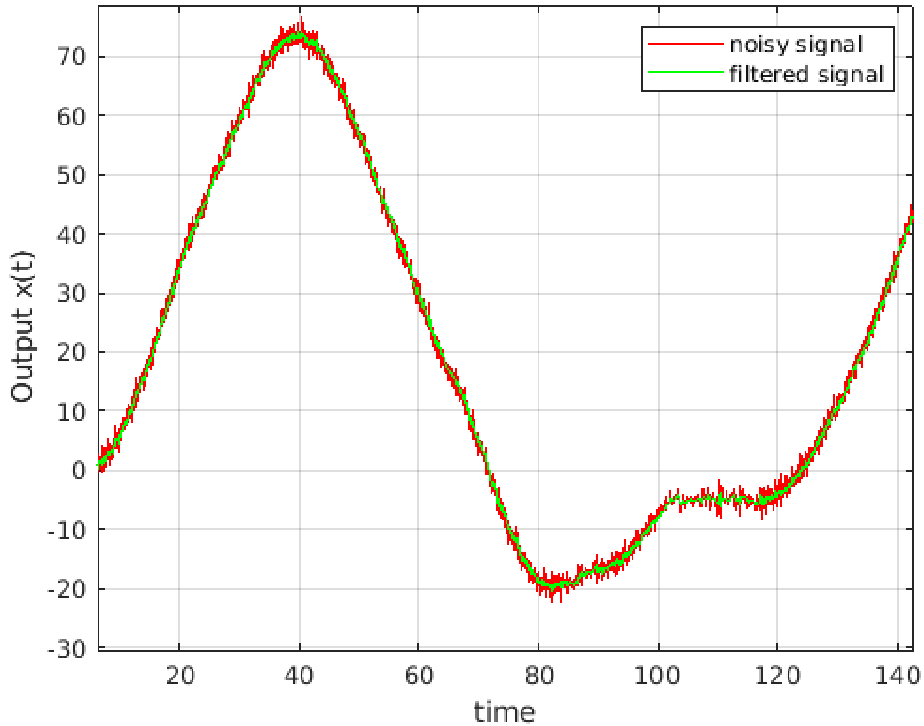
RMS_YAW vs Time



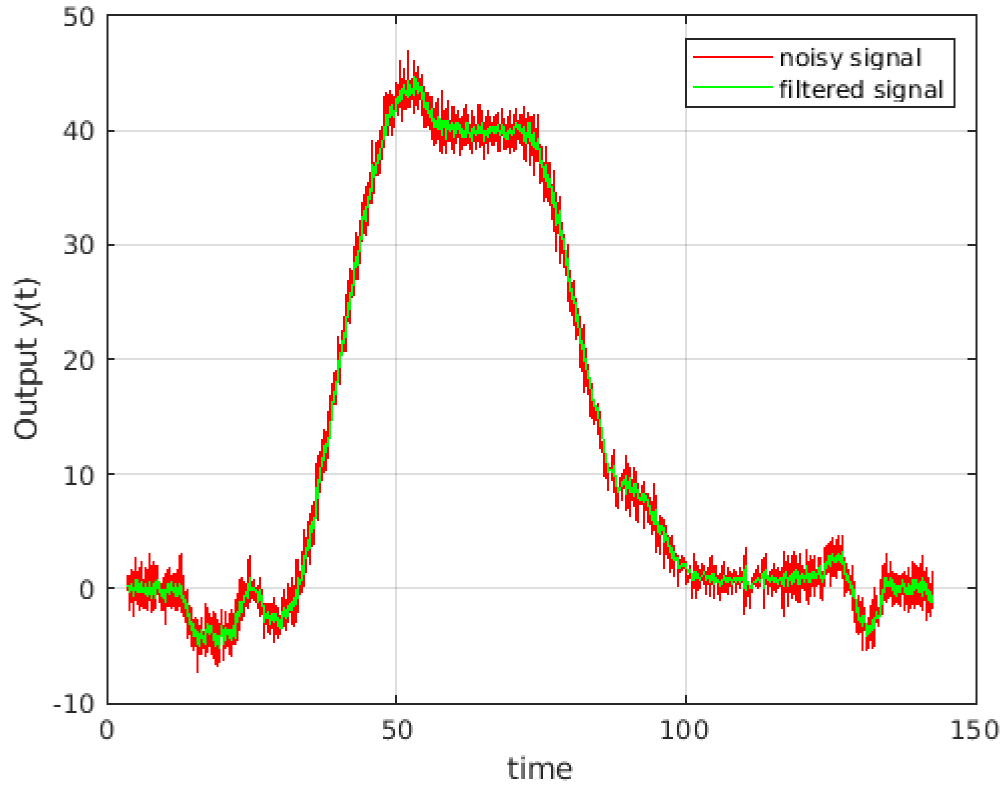
X vs Y filtered



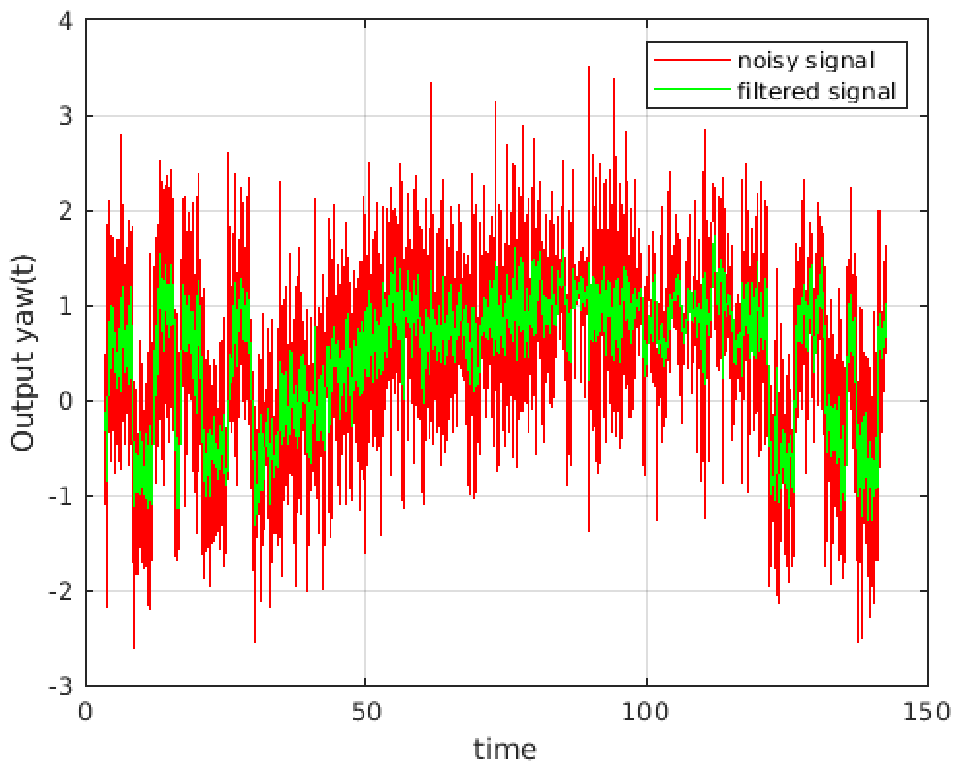
X vs Time filtered



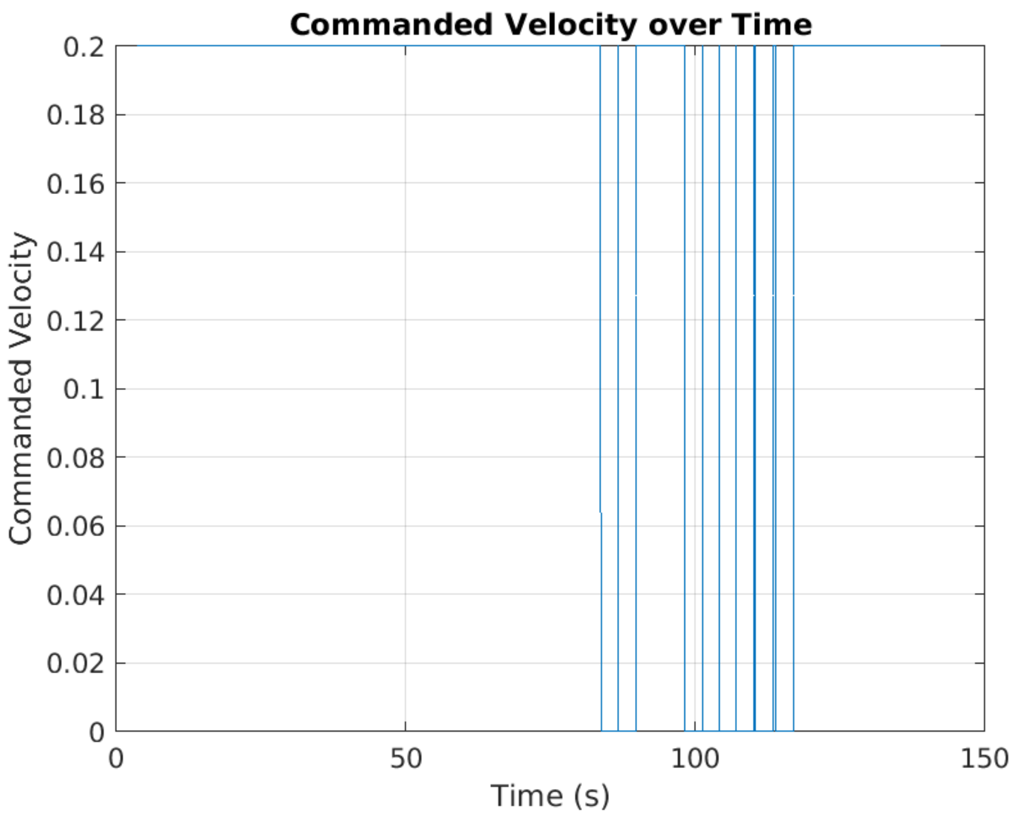
Y vs Time filtered



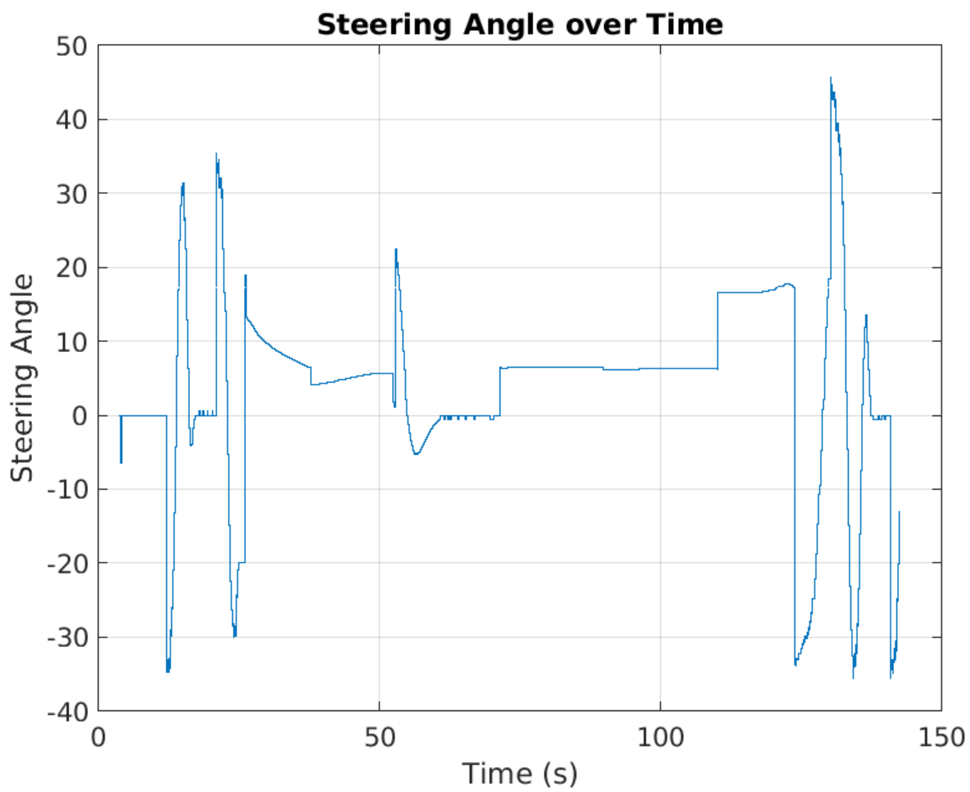
YAW vs Time filtered



velocity_cmd vs time



steering_angle vs Time



Conclusion

Provide a conclusion of the whole project what you have learned so far.

- Phase 1: Object Creation and Simulation

Initially, we learned to create objects in CoppeliaSim, including adding joints and physical geometries. We then integrated these objects into RViz and Gazebo, enabling us to visualize and simulate robotic behaviors effectively.

- Phase 2: Control and Navigation Algorithms

We explored various control algorithms such as PID, Pure Pursuit, and Stanley, understanding their applications and appropriate use cases. Additionally, we delved into navigation algorithms, including filtering techniques and SLAM (Simultaneous Localization and Mapping), and learned when to apply each method. This phase enhanced our understanding of navigation and path planning concepts essential for autonomous vehicles.

- Phase 3: Mathematical Foundations and Applications

In the final phase, we focused on the application of mathematical principles in coding and algorithm development. We examined the mathematical expressions relevant to vehicle dynamics and their impact on the behavior of an autonomous car. This mathematical insight proved crucial in refining our control and navigation strategies.

Overall, this project has provided us with a robust foundation in robotic simulation, control algorithms, navigation techniques, and the mathematical underpinnings of autonomous systems.

Note:

https://drive.google.com/drive/folders/1a43vUpi7CoiALsGSzRfYnTOBFUl-1UiY?usp=drive_link

you will find here our vision model and paths and cones and all file