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:

- Specific details for handling circular paths will follow similar principles, adapted to the unique characteristics of circular navigation.

5. **Custom Track:**

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.)

- **Tools and Packages**

- Simulation Tools: RViz, CoppeliaSim
- ROS Packages: `pc2`, `cv_bridge`, `std_msgs`, `nav_msgs`, `sensor_msgs`, `geometry_msgs`

- **Libraries**

- 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: Apply noise and appropriate filtering techniques to sensor data.
2. System Initialization: Ensure all systems, including the vehicle, are operational at the beginning of the track.
3. Algorithm Implementation: Implement Pure Pursuit algorithms to guide the vehicle.
4. Look-Ahead Point Calculation: Use look-ahead point algorithms to 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 is used to generate target points. By applying the straight-line equation, the vehicle moves towards the next point, adjusting its path using the look-ahead parameter.

Path 2: Straight Line with Lane Change

The vehicle moves in a straight line, using its camera to detect humans and LiDAR to measure distance. Upon detecting a human, the vehicle executes 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 the navigation of circular routes by using the appropriate circular equations.

This formalized approach highlights the structured methodology and comprehensive application of algorithms and tools in the project.

Custom Track:

Tracks Description

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

Initial Challenges:

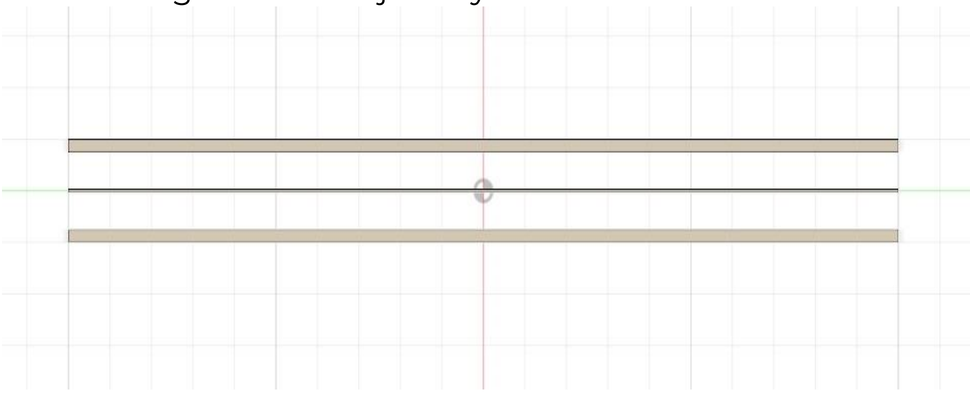
The primary challenge involved familiarizing ourselves with CoppeliaSim. This included defining physical dimensions, modifying object geometries, configuring vehicle parameters, and ensuring accurate movement in both circular paths and custom tracks.

Path Design:

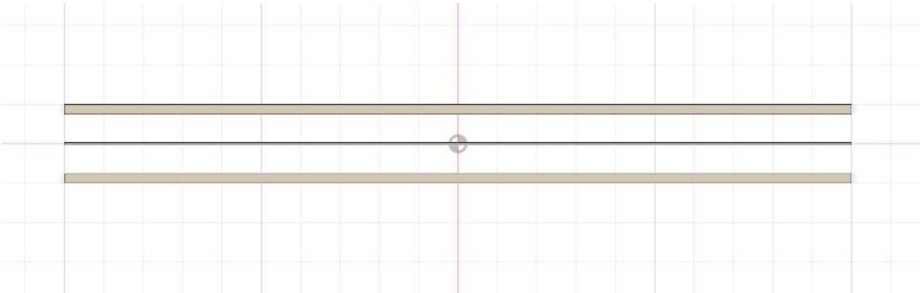
We designed our paths using Fusion 360, converting them into objects that could be imported and utilized in CoppeliaSim for simulation.

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.

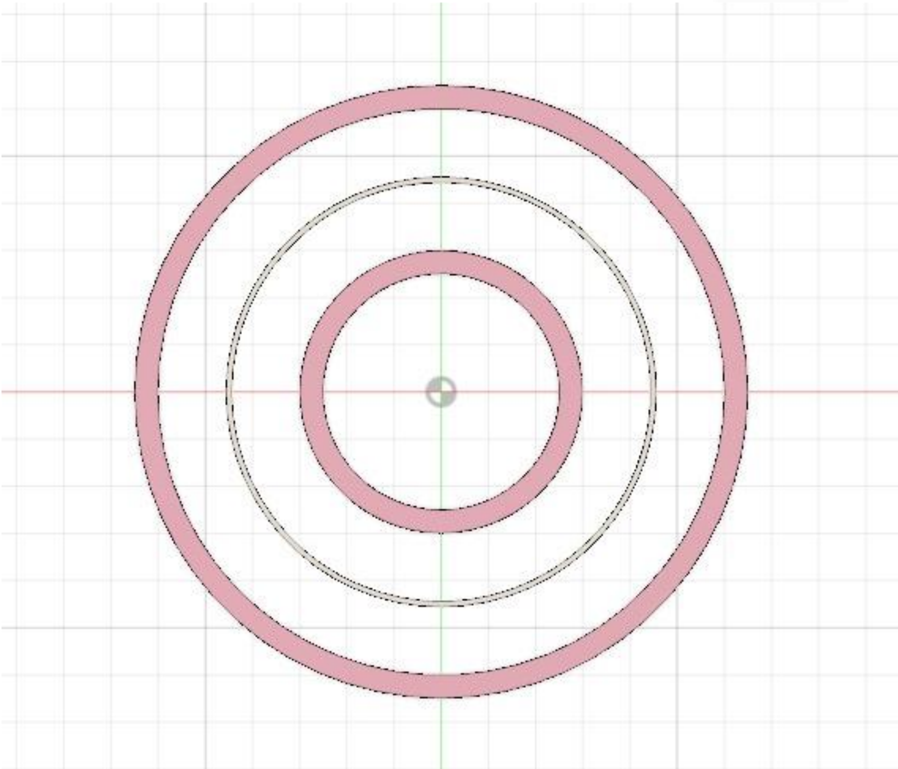


2. Path 2: Straight Line with Lane Changes:
   - Implementation: The vehicle moves in a straight line but includes algorithms to detect obstacles like slow vehicles or cones. Upon detection, the vehicle executes a lane change.
   - Challenges: Developing a robust algorithm to decide when to change lanes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

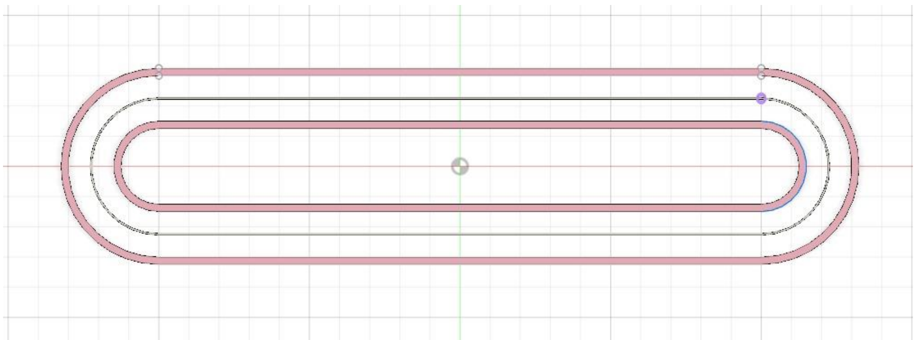


3. Path 3: Circular Path:

- 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 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.

- **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.

- **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.

- **Custom Track**

- **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"
```

```
    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

```
#!/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)

```python
#!/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
```

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
```

```python
    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
```


```python
    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
```


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)
```

```
    lidar_sub = rospy.Subscriber('/lidar_points', Float32MultiArray, manage_lidar)

    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:
        path_flag = 1

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

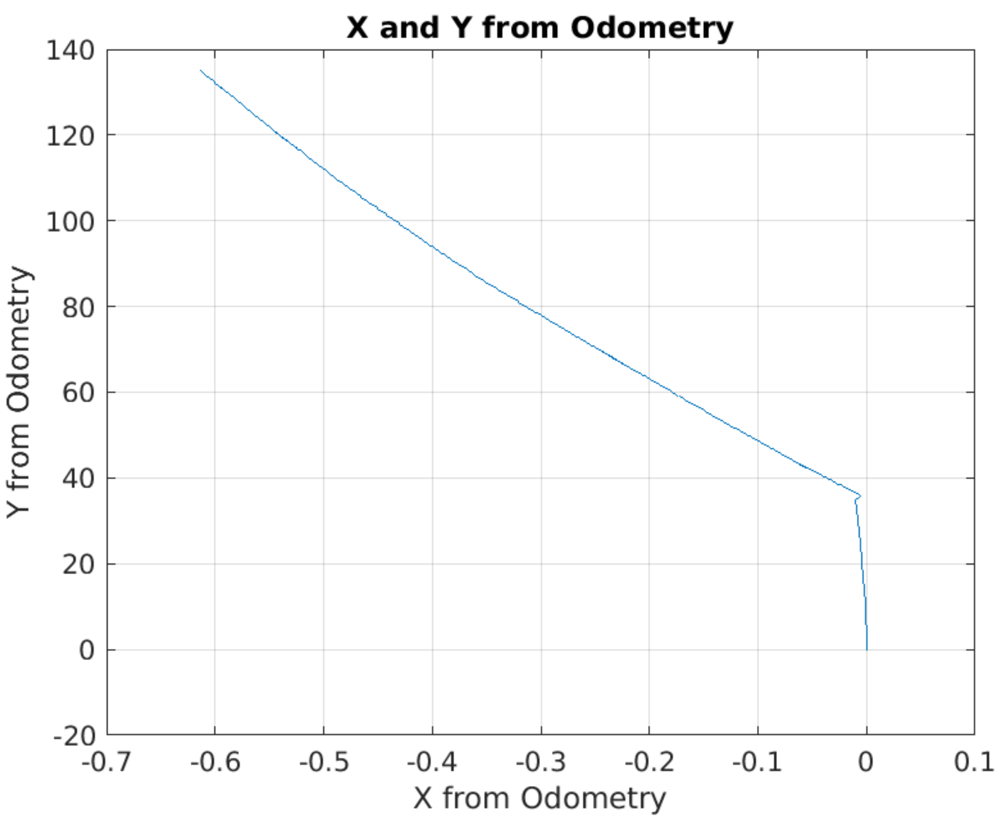
Custom path

Performance analysis

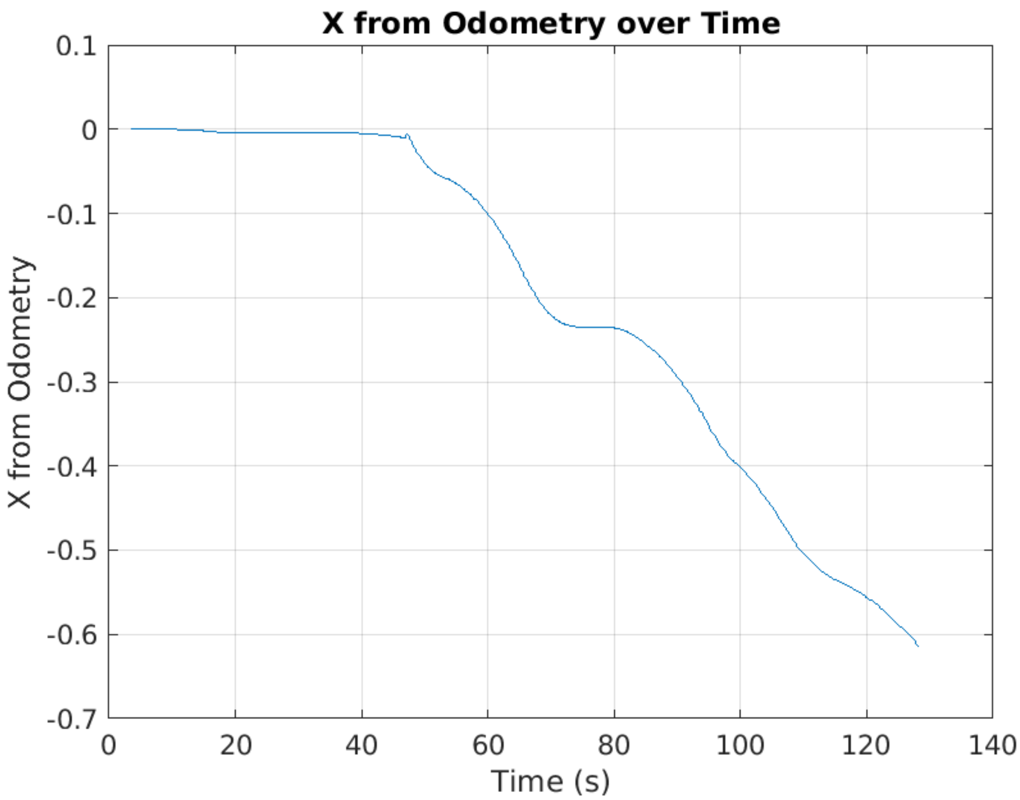
Analyze the system's performance on each track, while providing graphs that validates the performance of your system, discussing any challenges encountered and how they were addressed.

Path1 (Stright line):

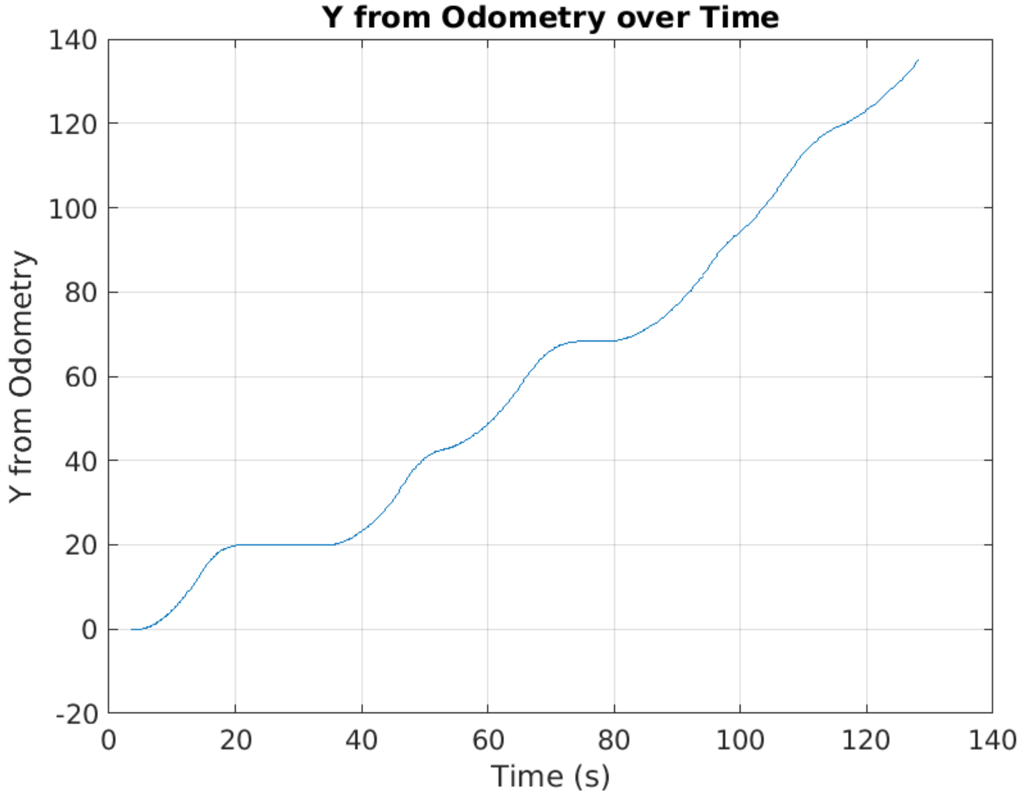
X vs Y



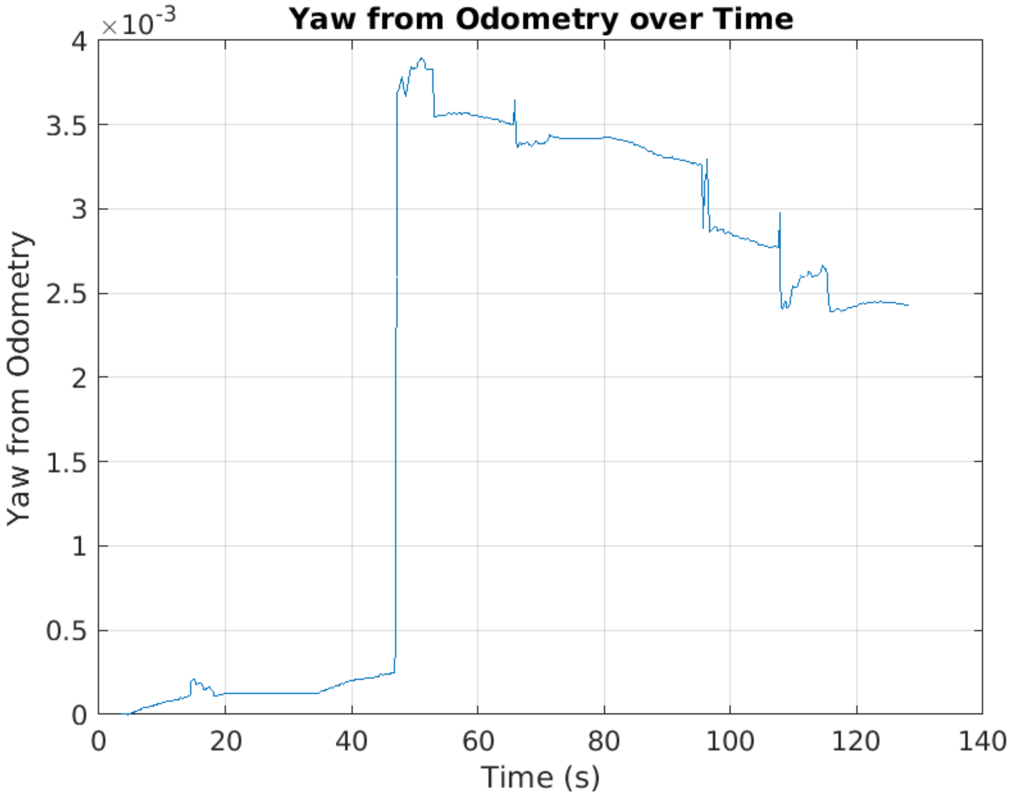
X vs Time


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

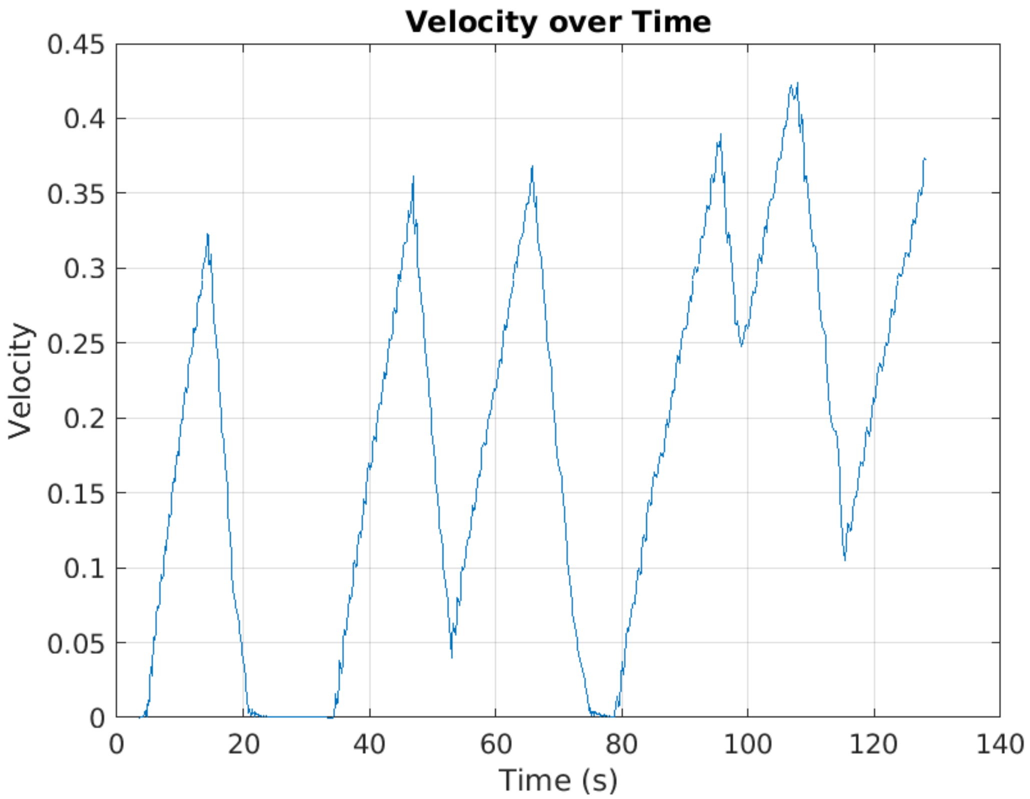
Y vs Time


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

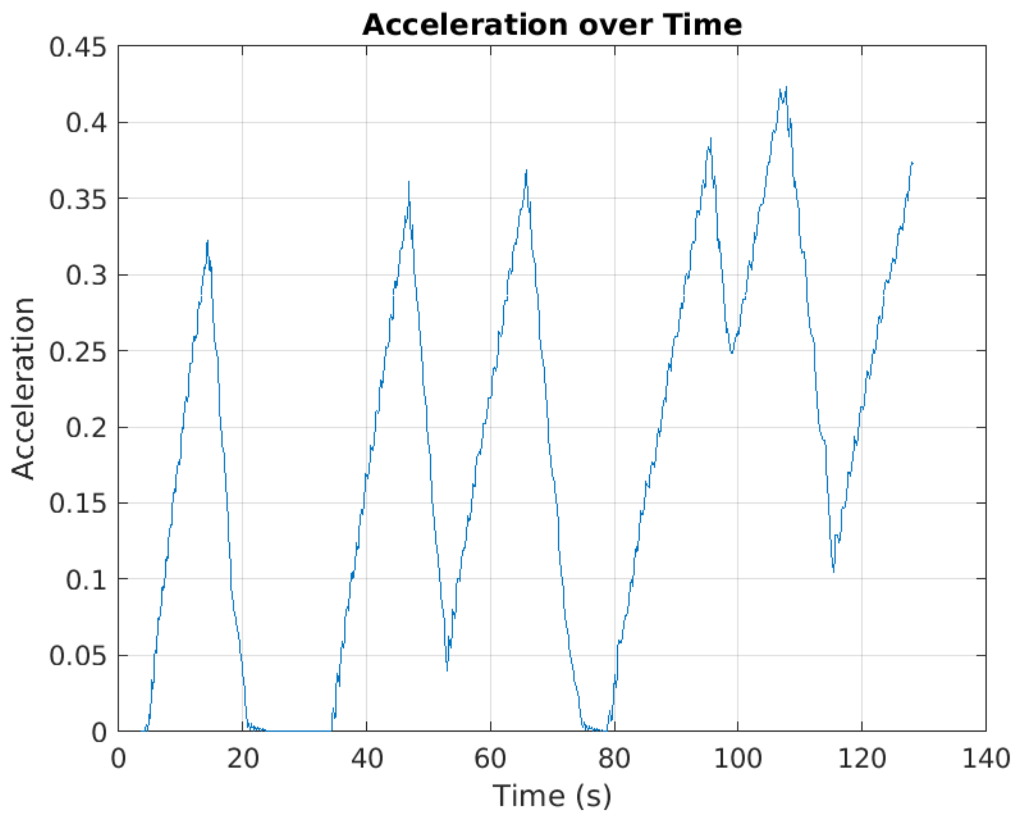
Yaw vs Time



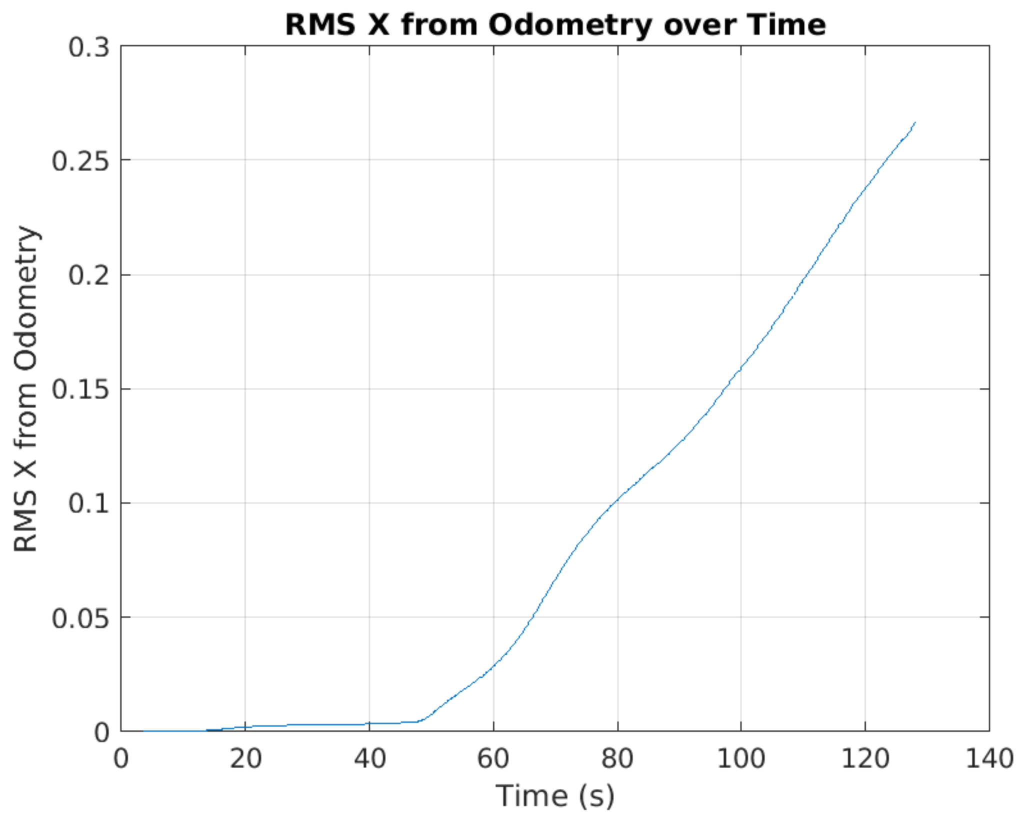
Velocity vs Time



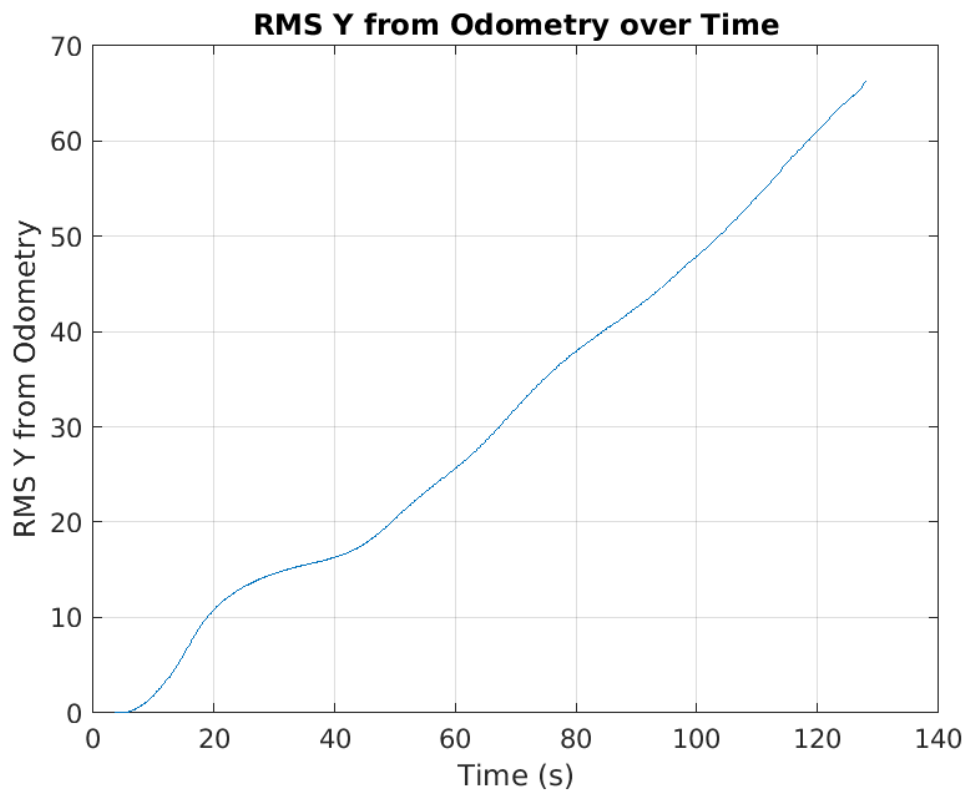
Acceleration vs Time



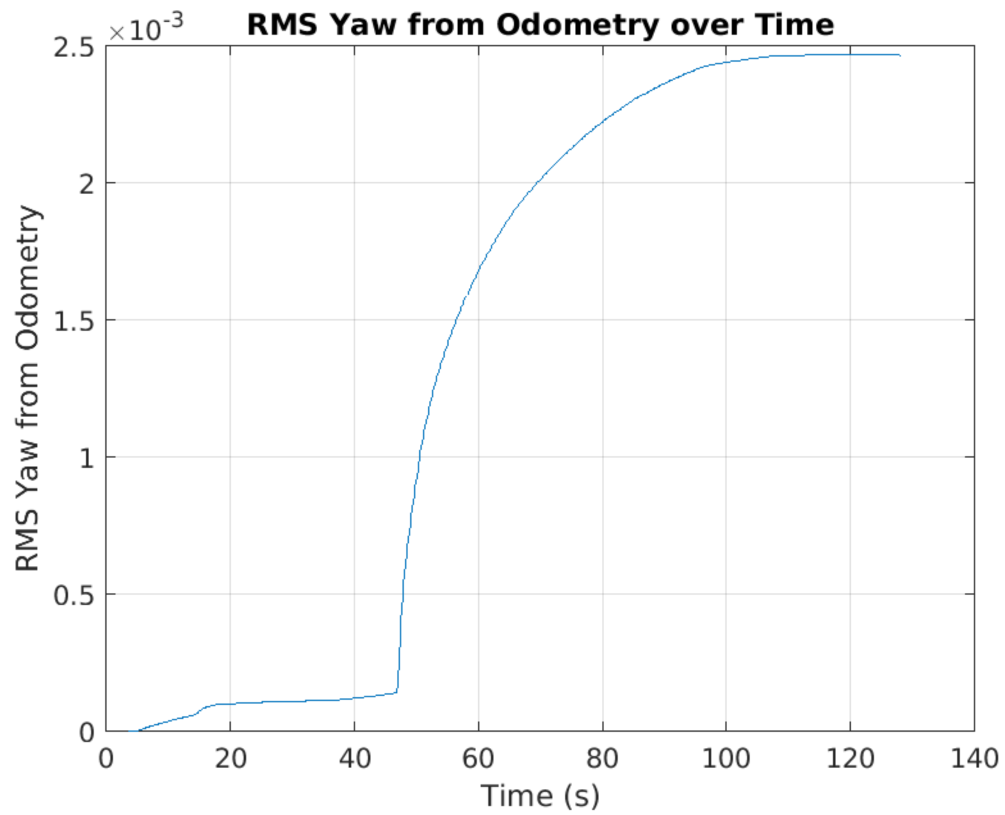
RMS_X vs Time



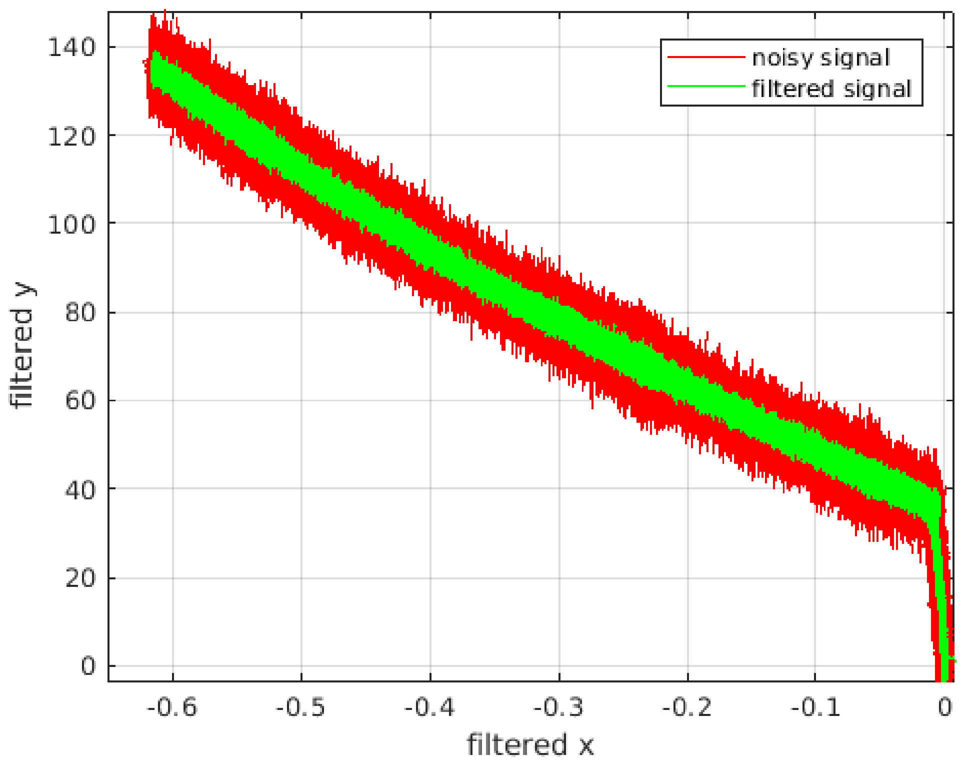
RMS_Y vs Time



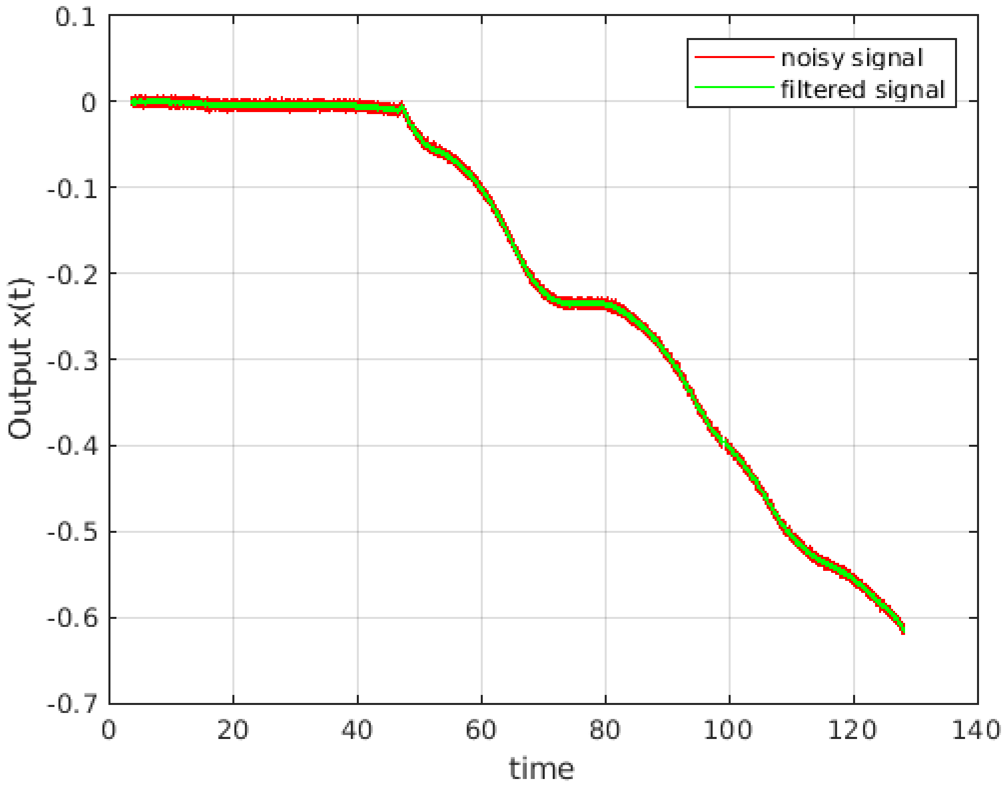
RMS_YAW vs Time



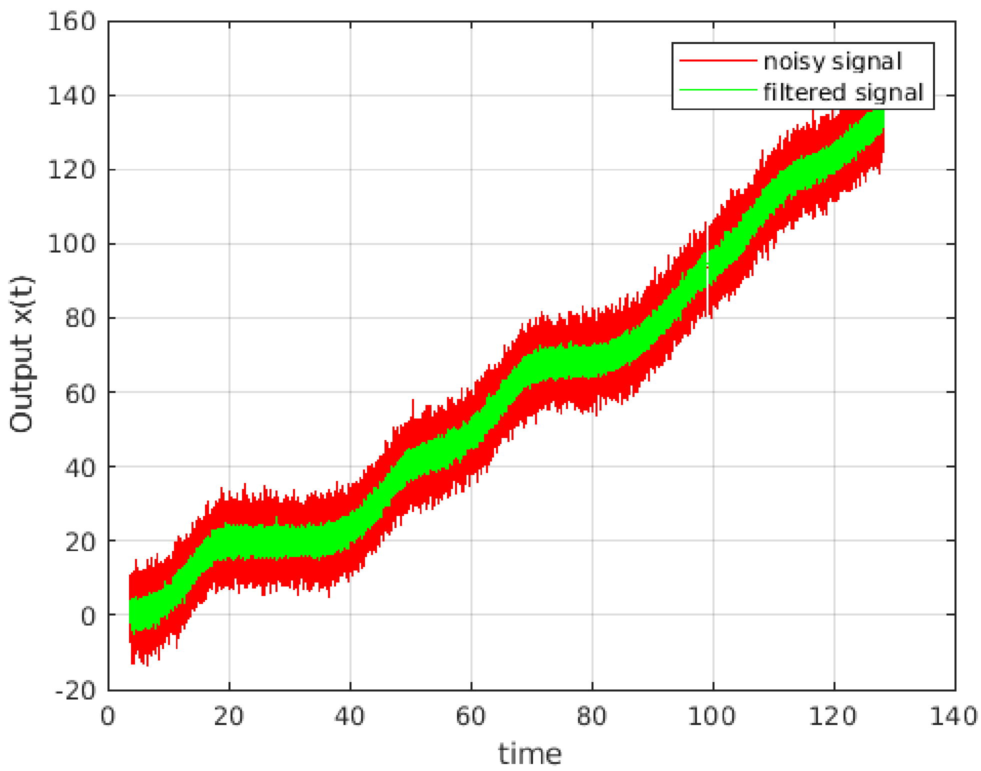
X vs Y filtered



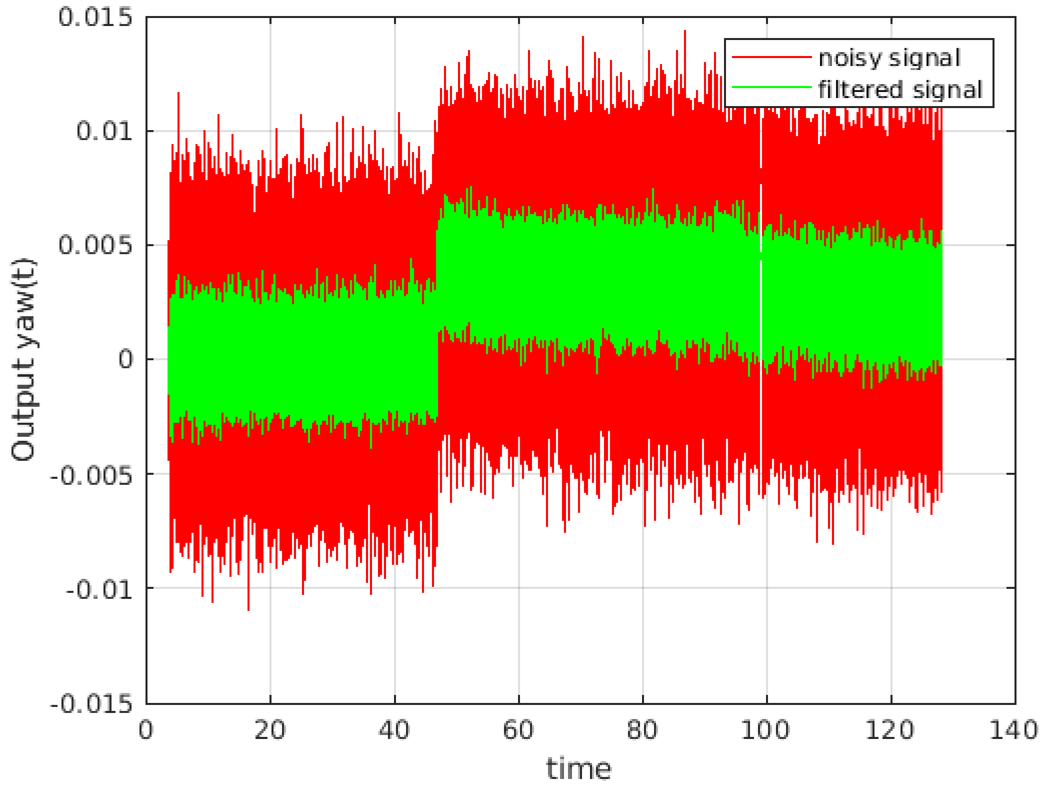
X vs Time filtered



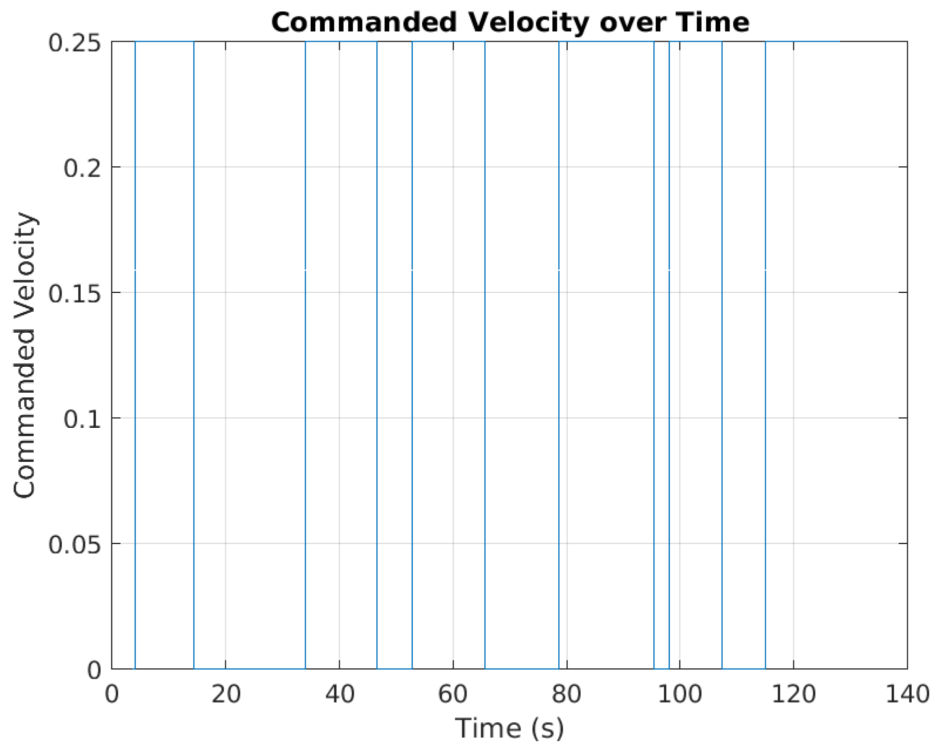
Y vs Time filtered



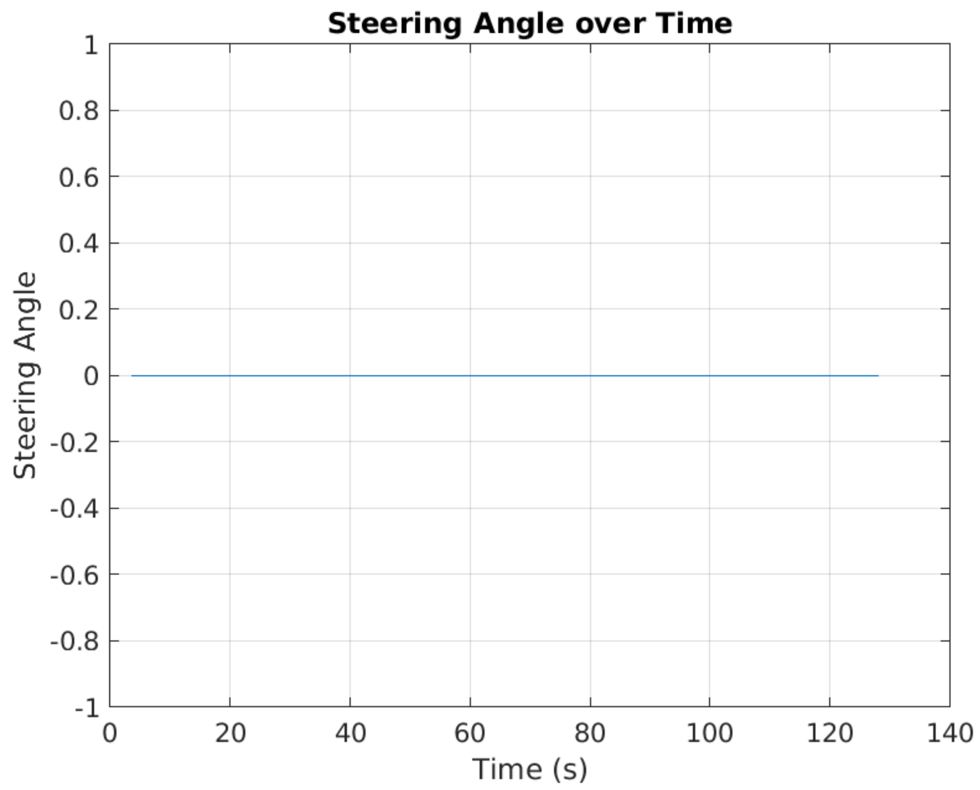
Yaw vs Time filtered



velocity_cmd vs time

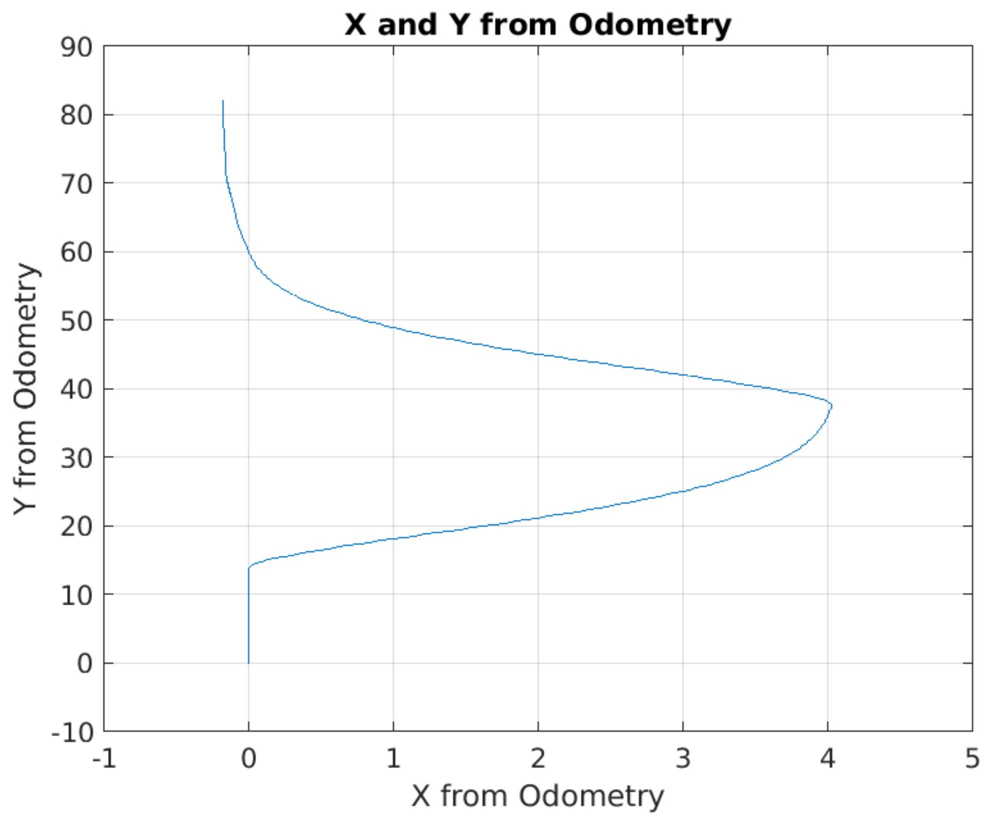


steering_angle vs Time

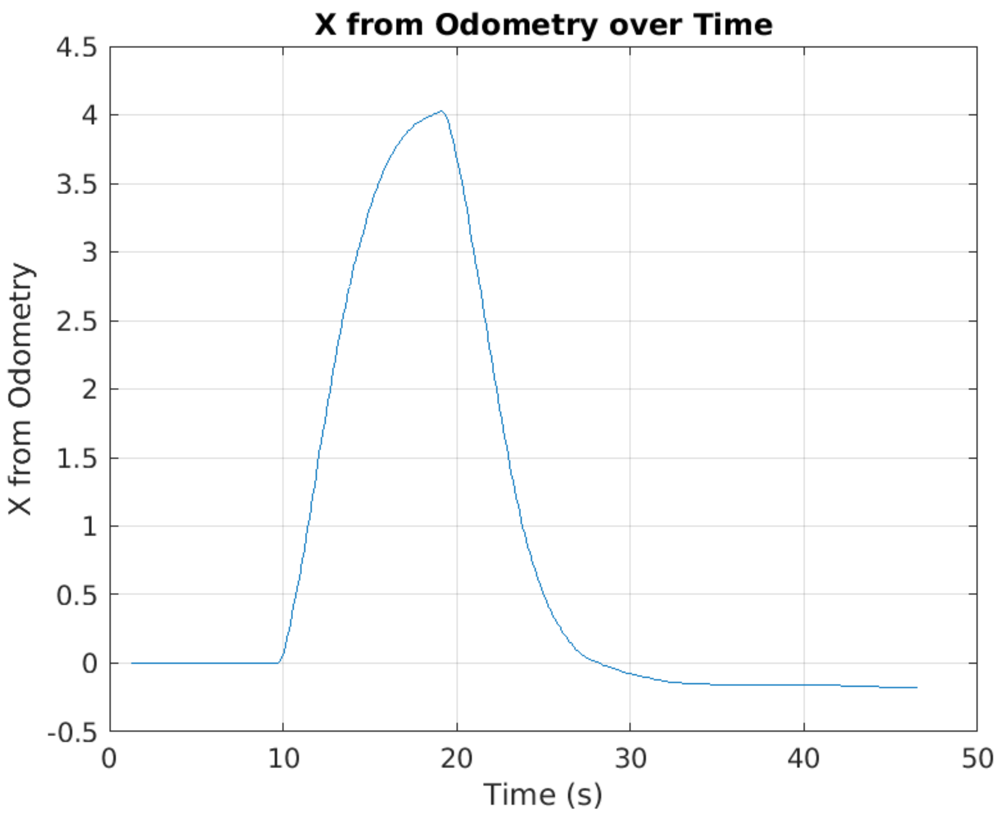


Path2 (Stright line with a change lane):

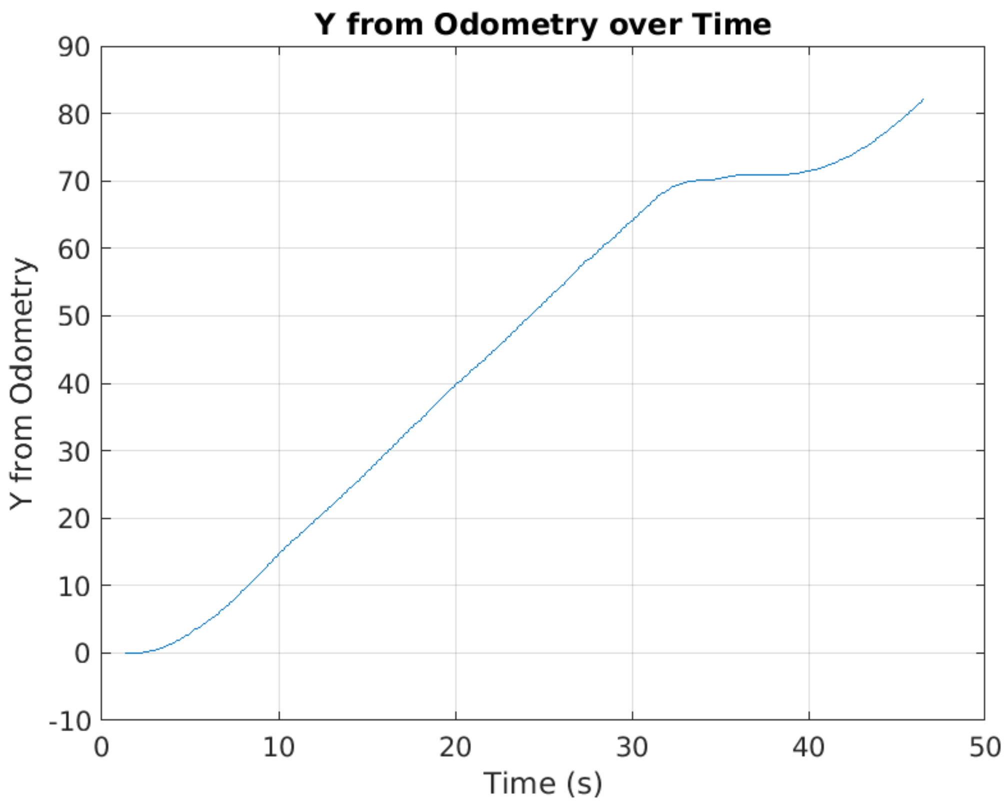
X vs Y


X and Y from Odometry
Y from Odometry
X from Odometry
90
80
70
60
50
40
30
20
10
0
-10
-1
0
1
2
3
4
5

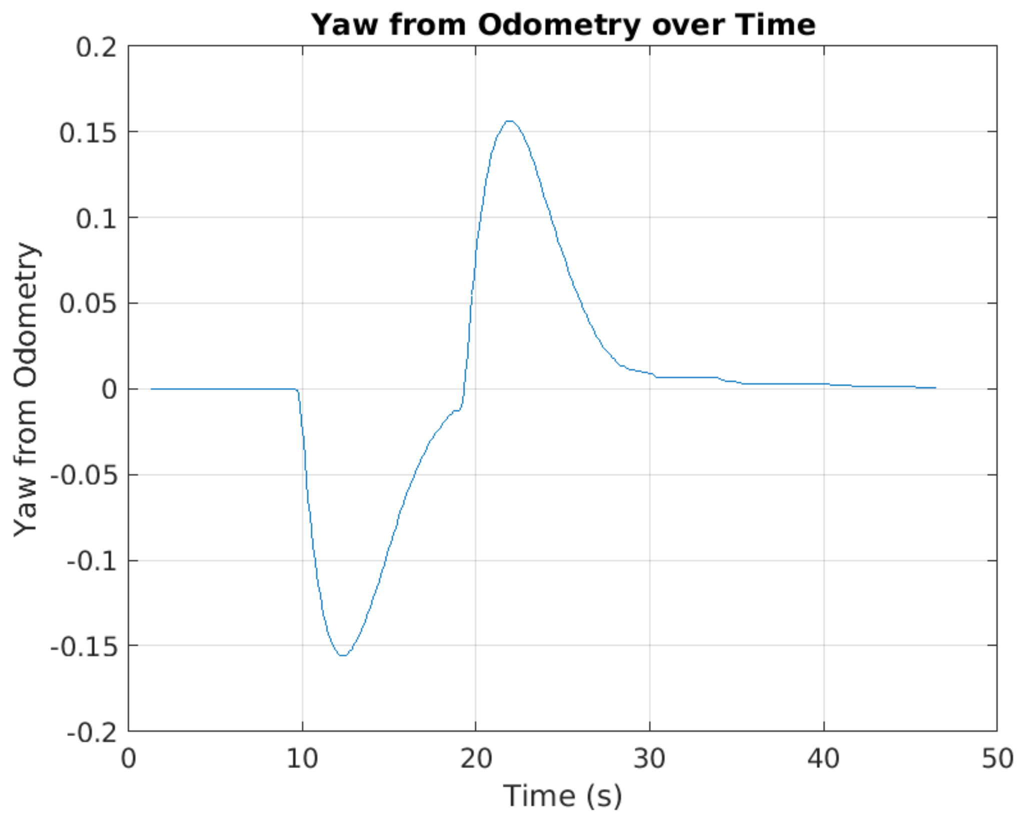
X vs Time


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

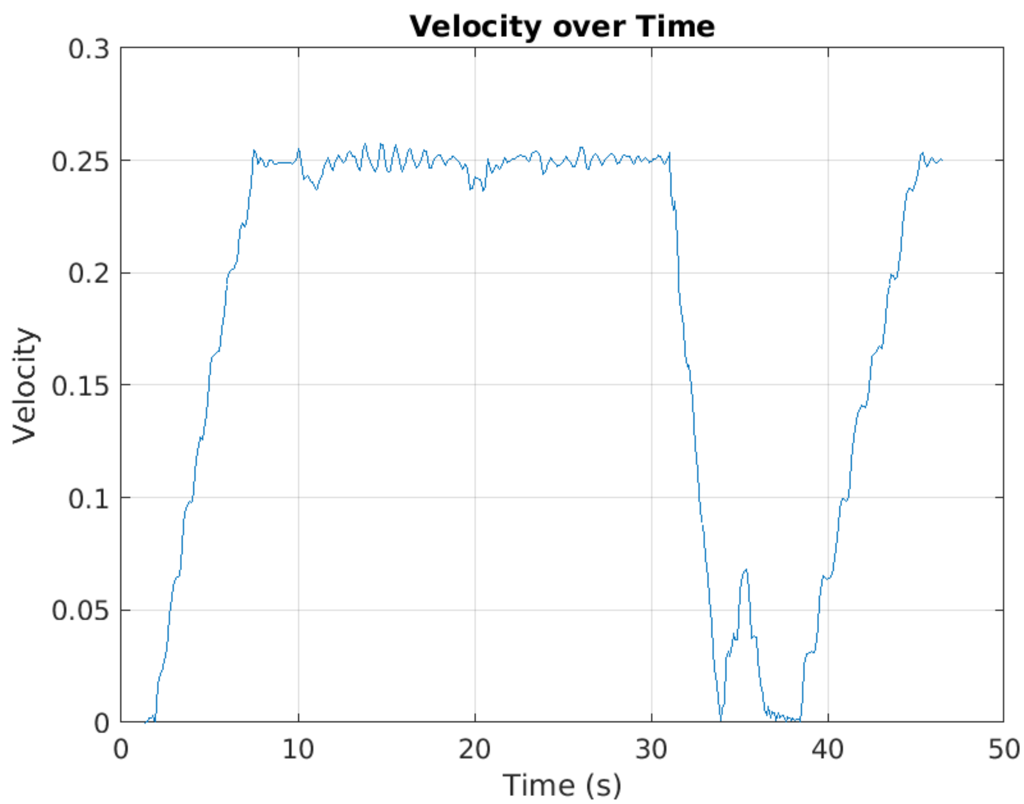
Y vs Time


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

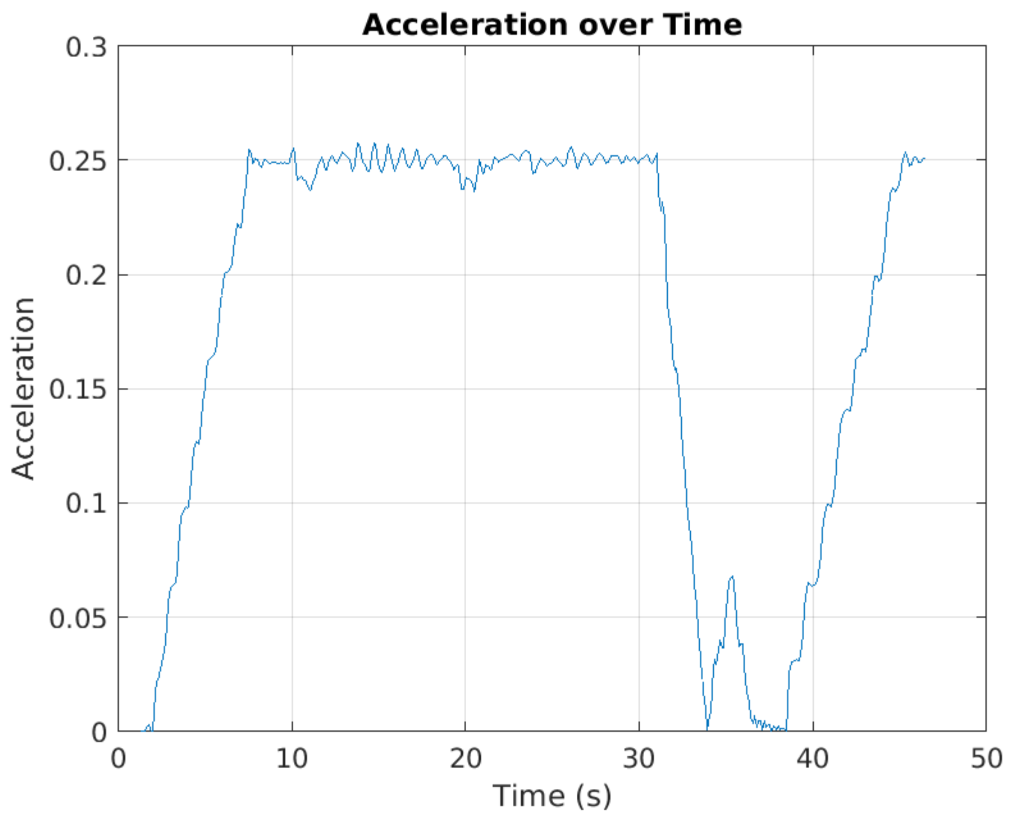
Yaw vs Time



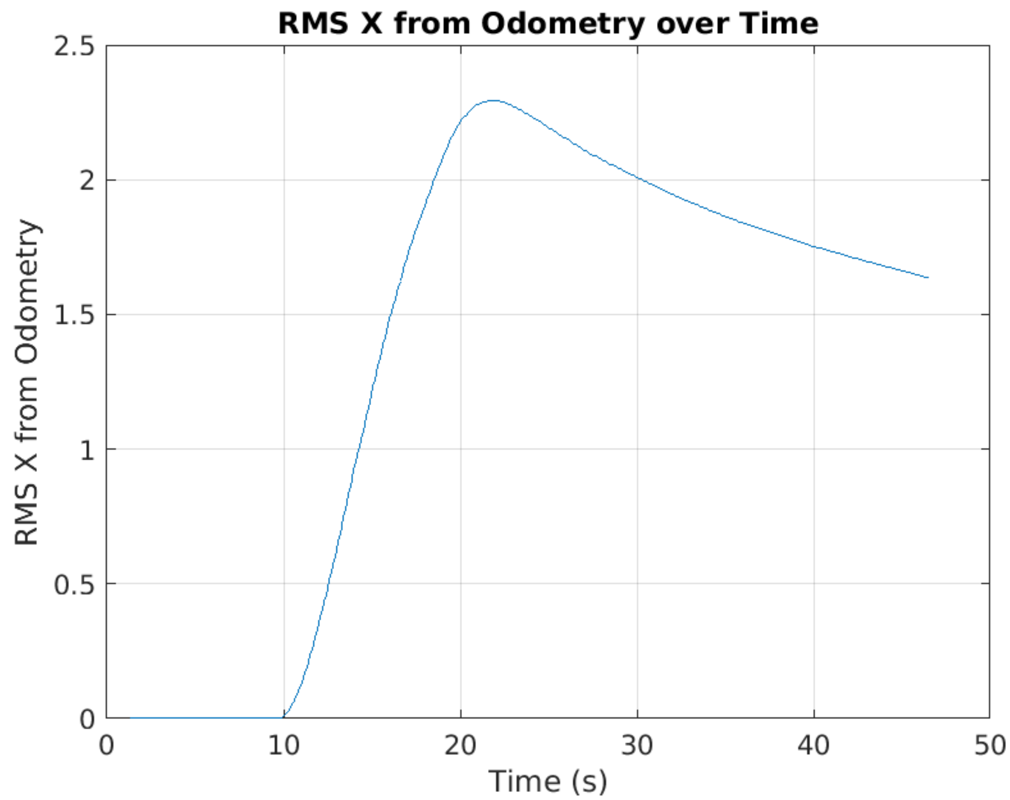
Velocity vs Time



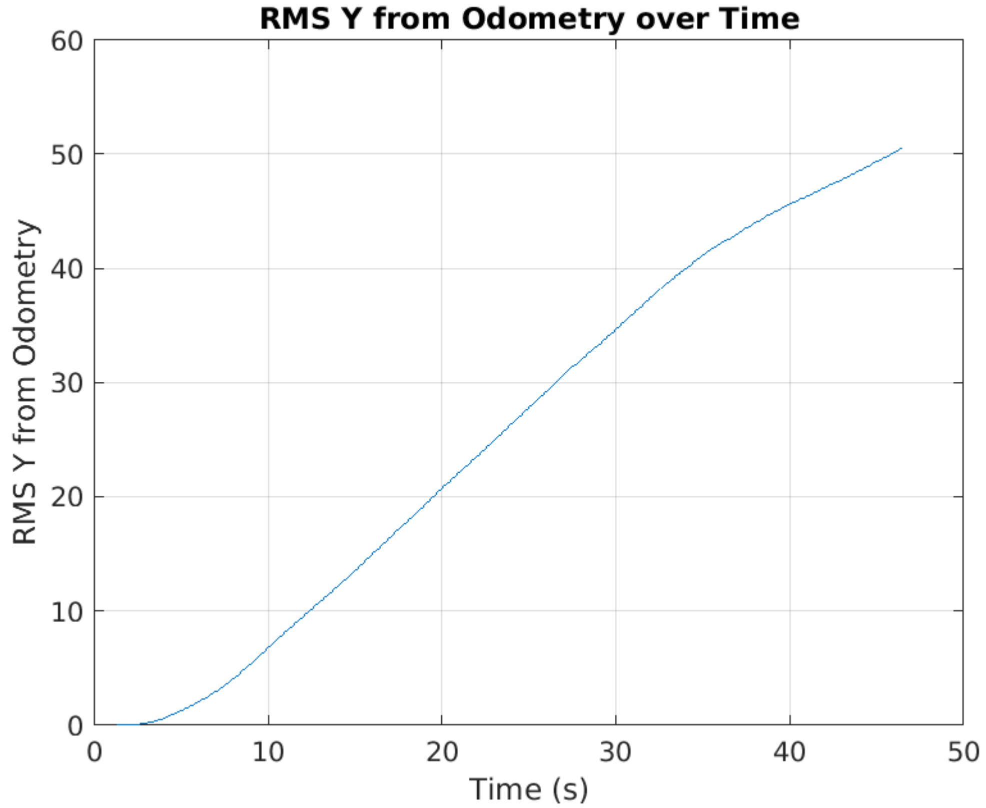
Acceleration vs Time



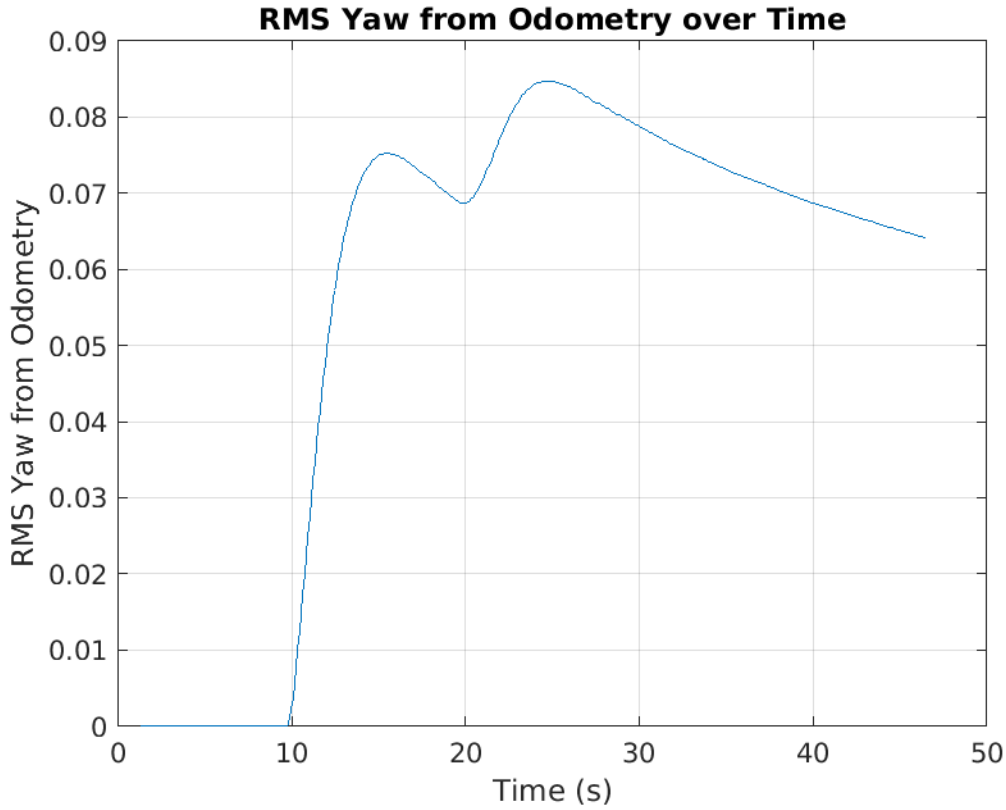
RMS_X vs Time


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

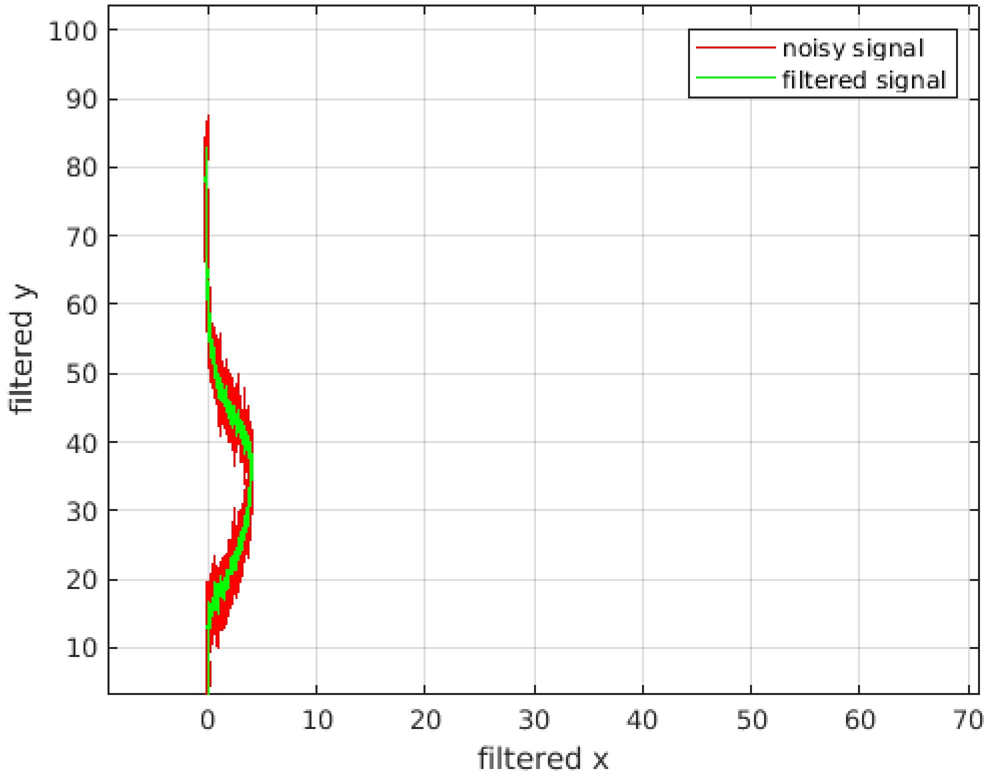
RMS_Y vs Time


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

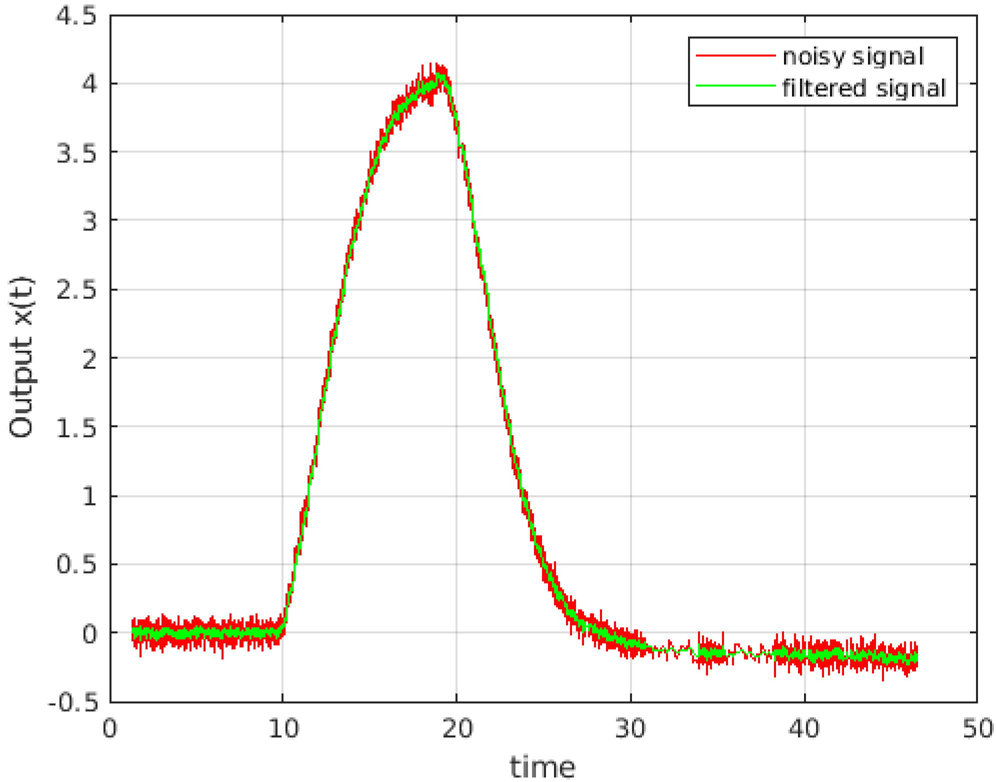
RMS_YAW vs Time



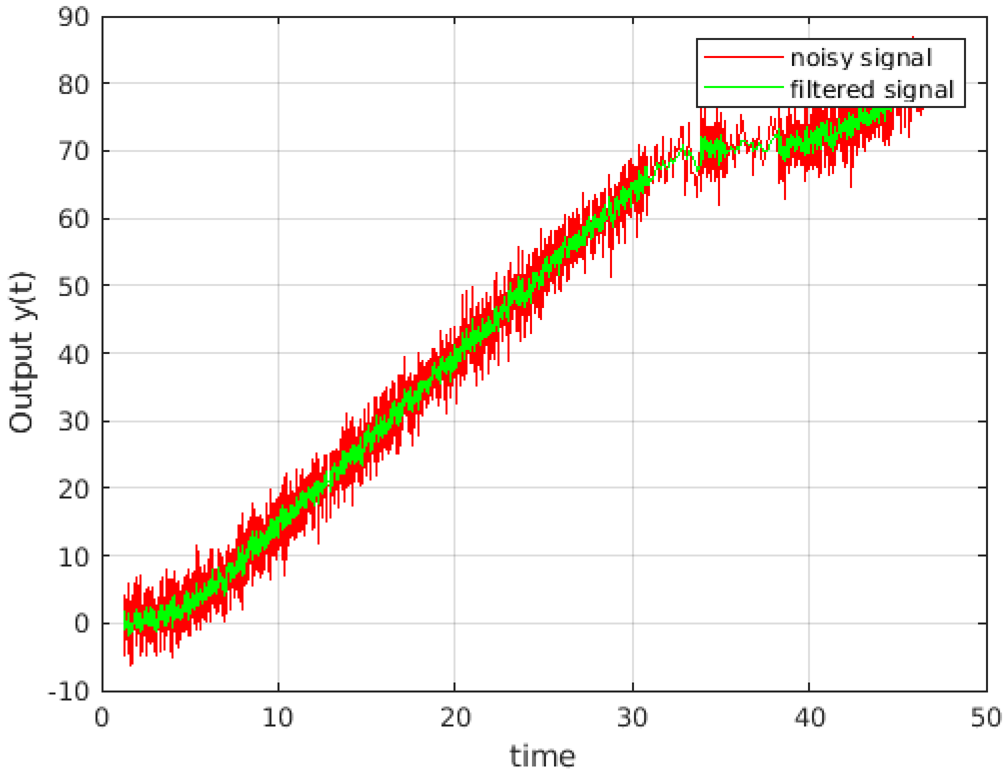
X vs Y filtered



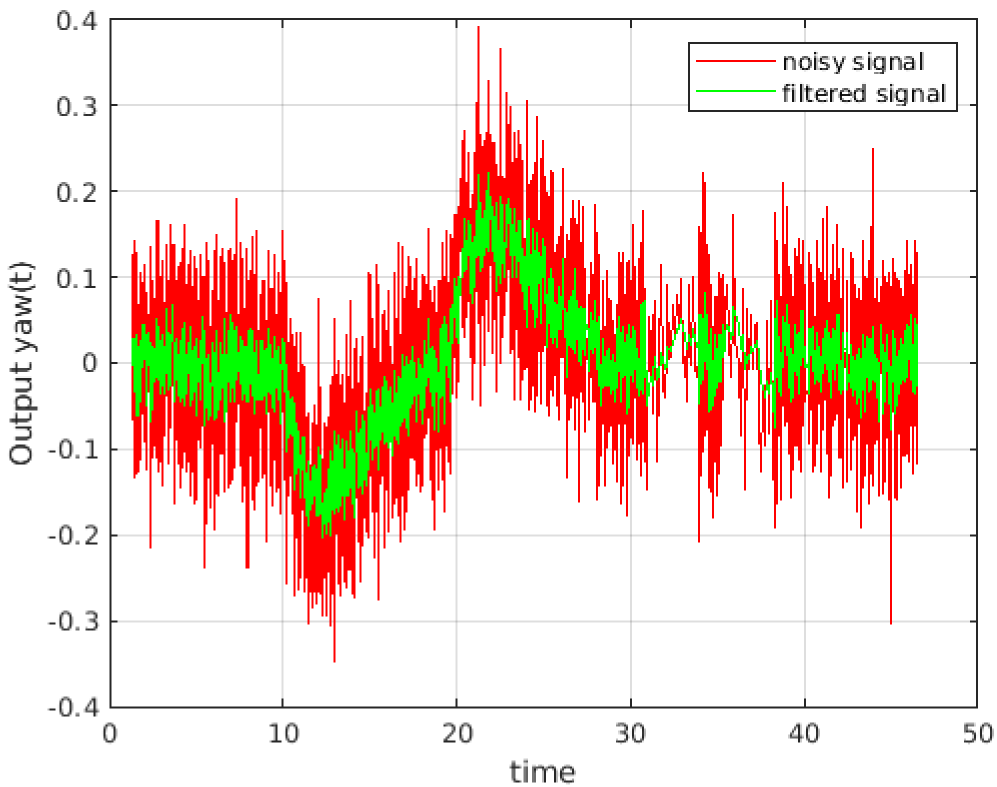
X vs Time filtered



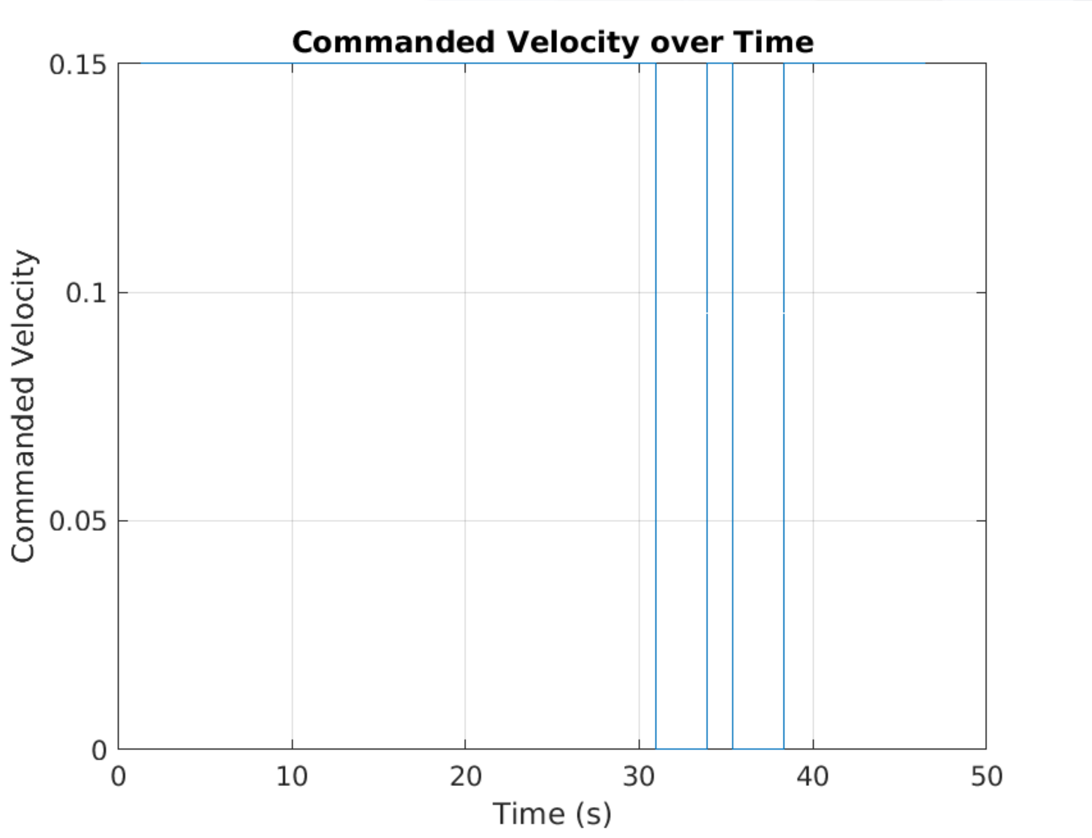
Y vs Time filtered



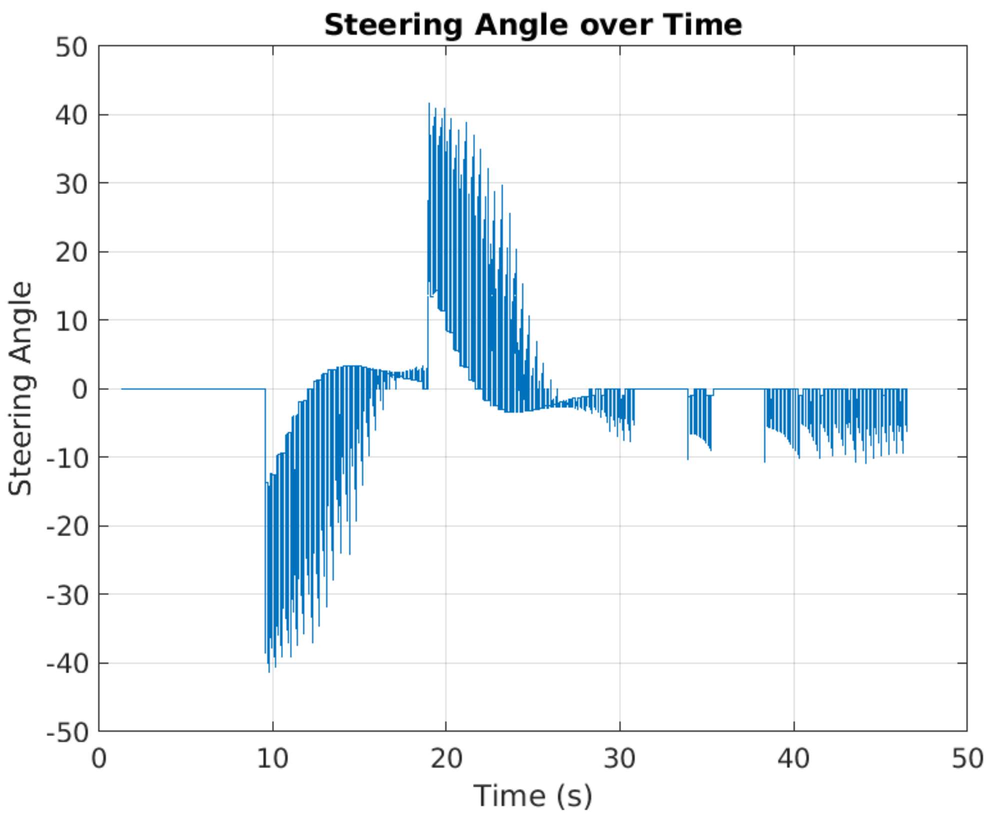
Yaw vs Time filtered



velocity_cmd vs time

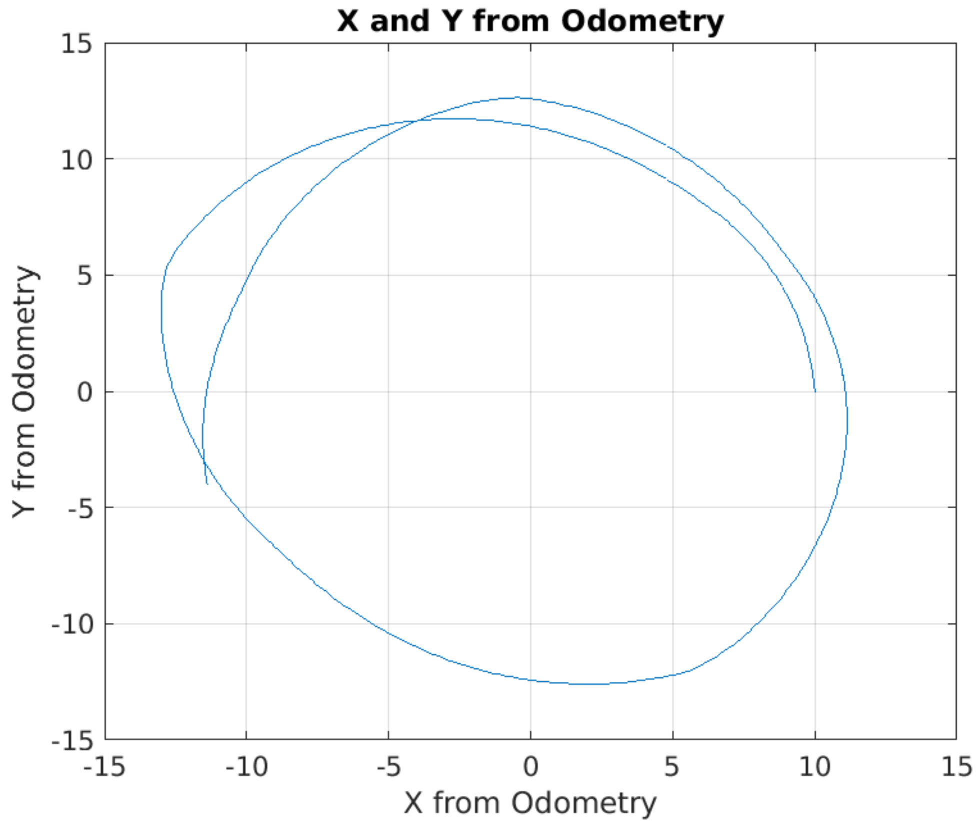


steering_angle vs Time

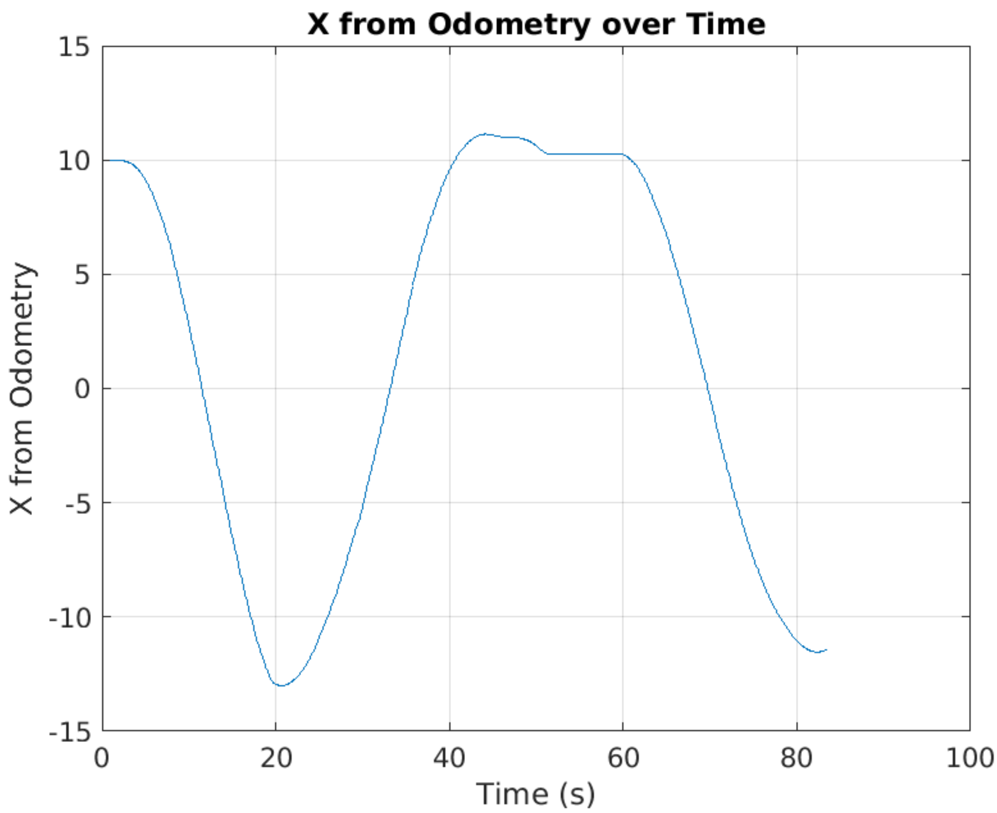


Path3 (Circular path):

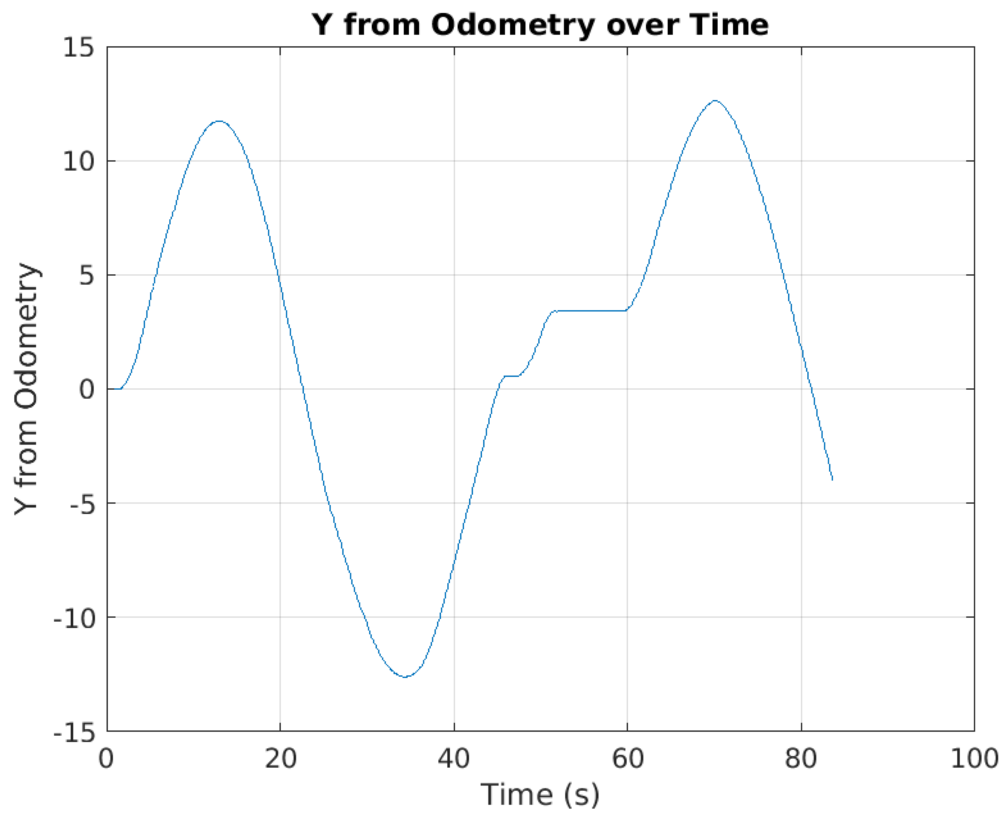
X vs Y



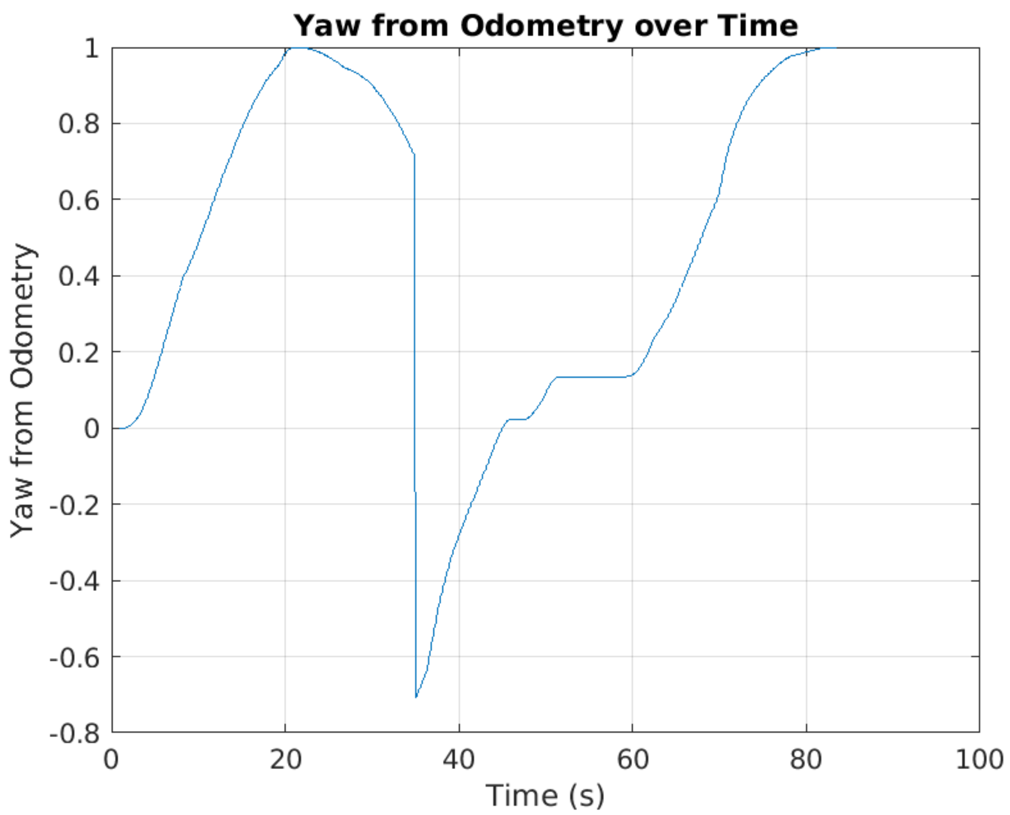
X vs Time



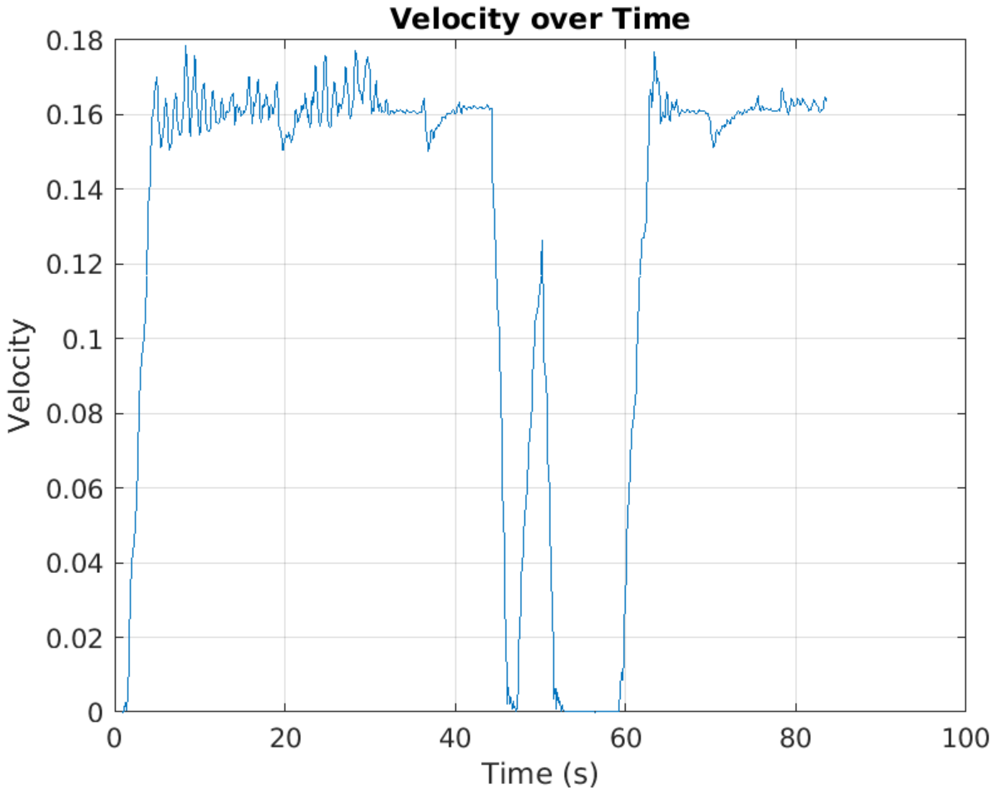
Y vs Time


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

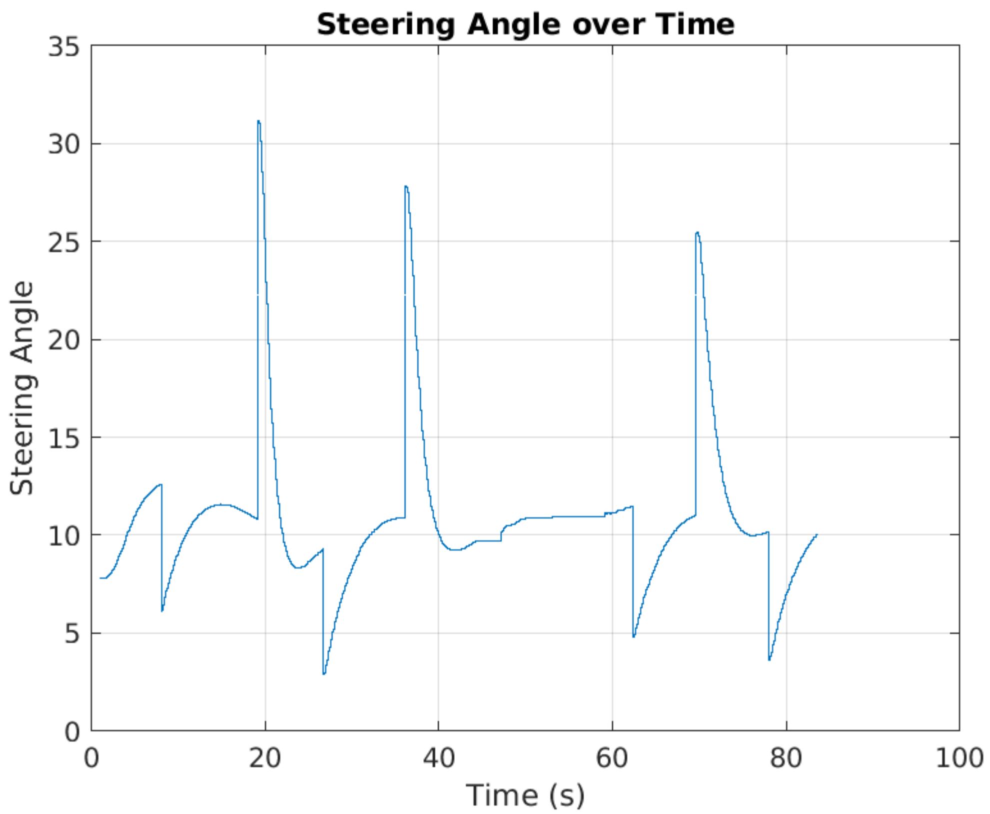
Yaw vs Time


Yaw from Odometry over Time
Yaw from Odometry
Time (s)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
20
40
60
80
100

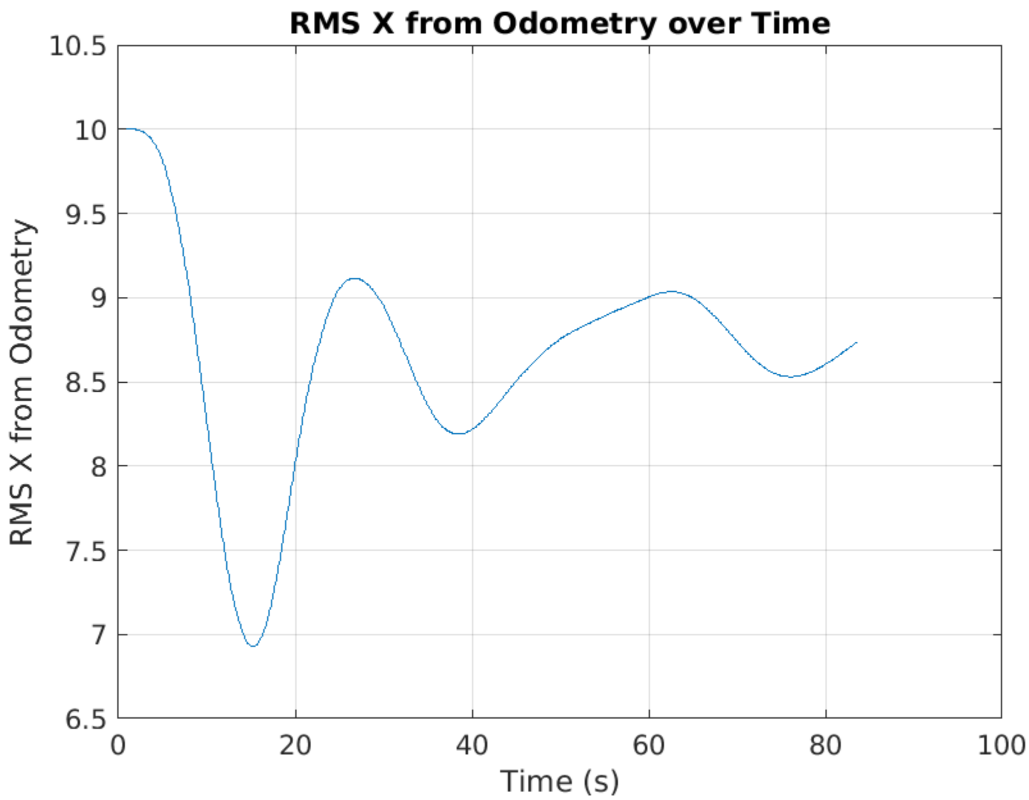
Velocity vs Time



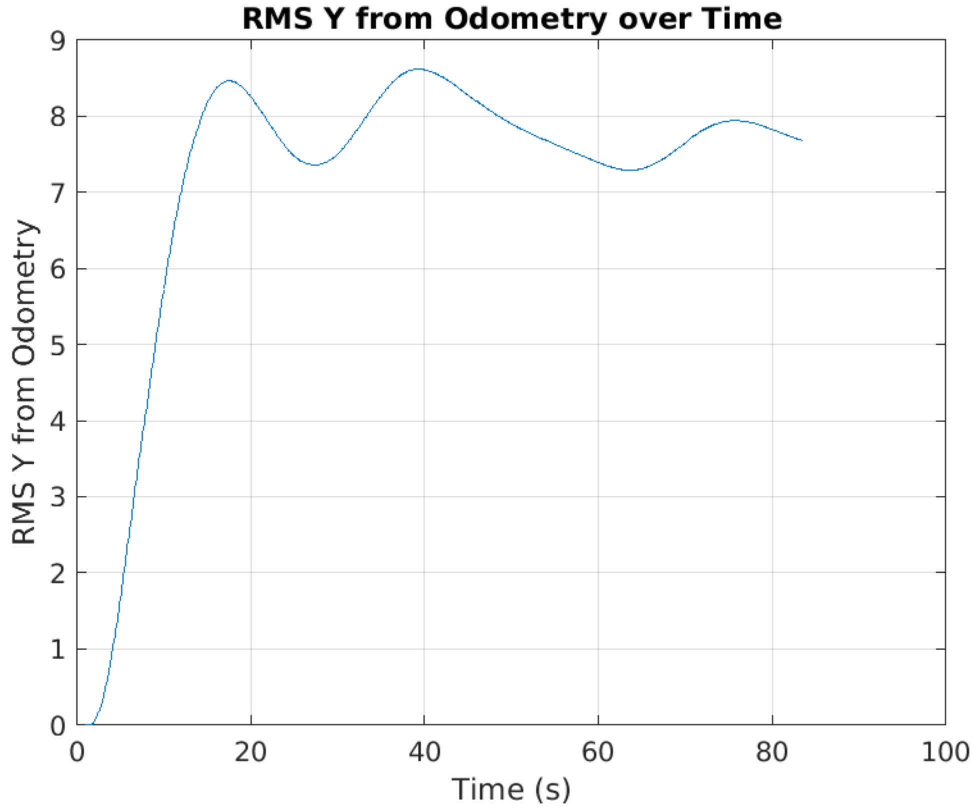
Acceleration vs Time



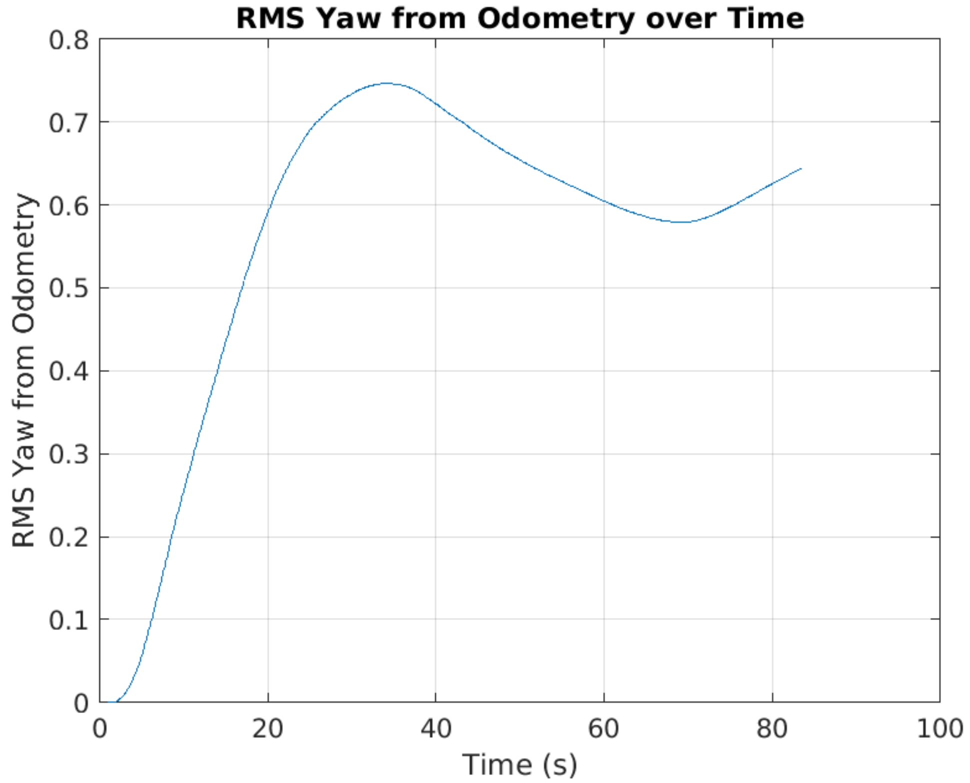
RMS_X vs Time



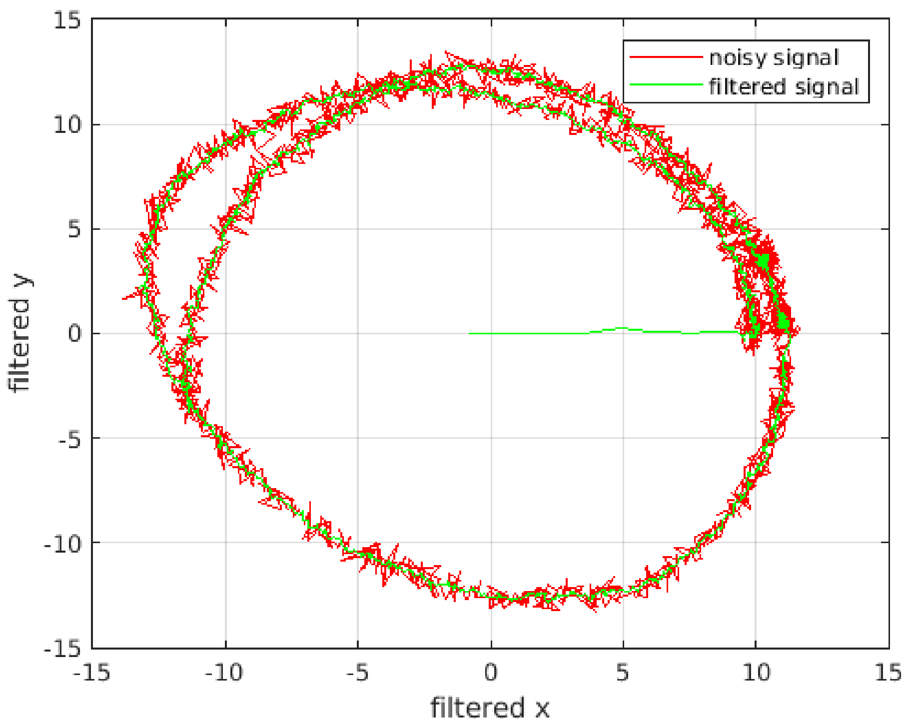
RMS_Y vs Time



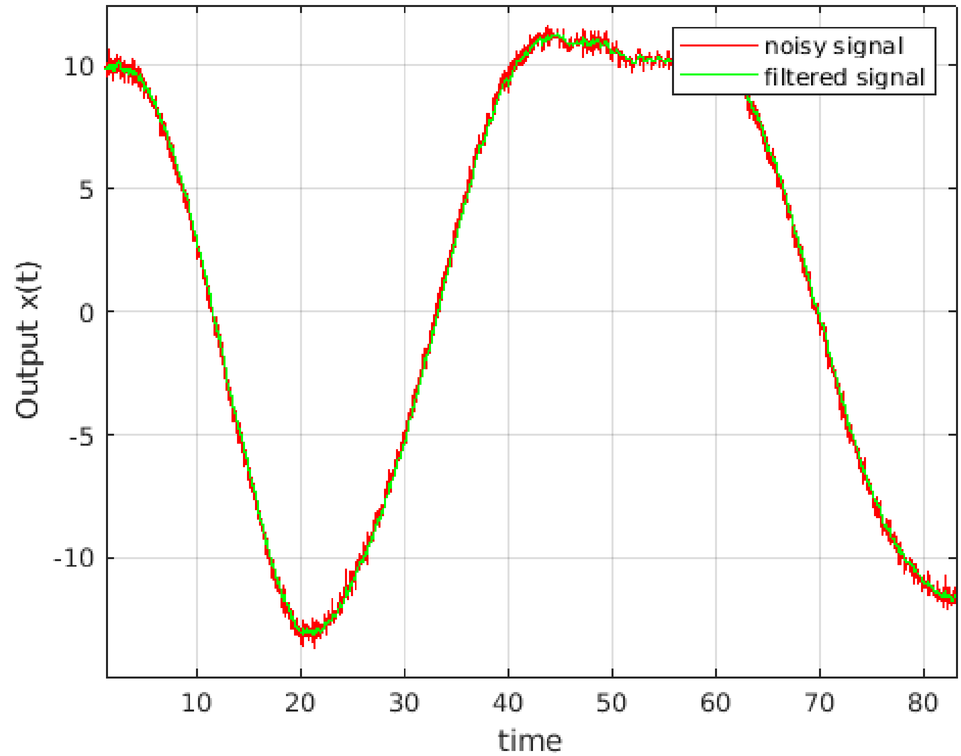
RMS_YAW vs Time



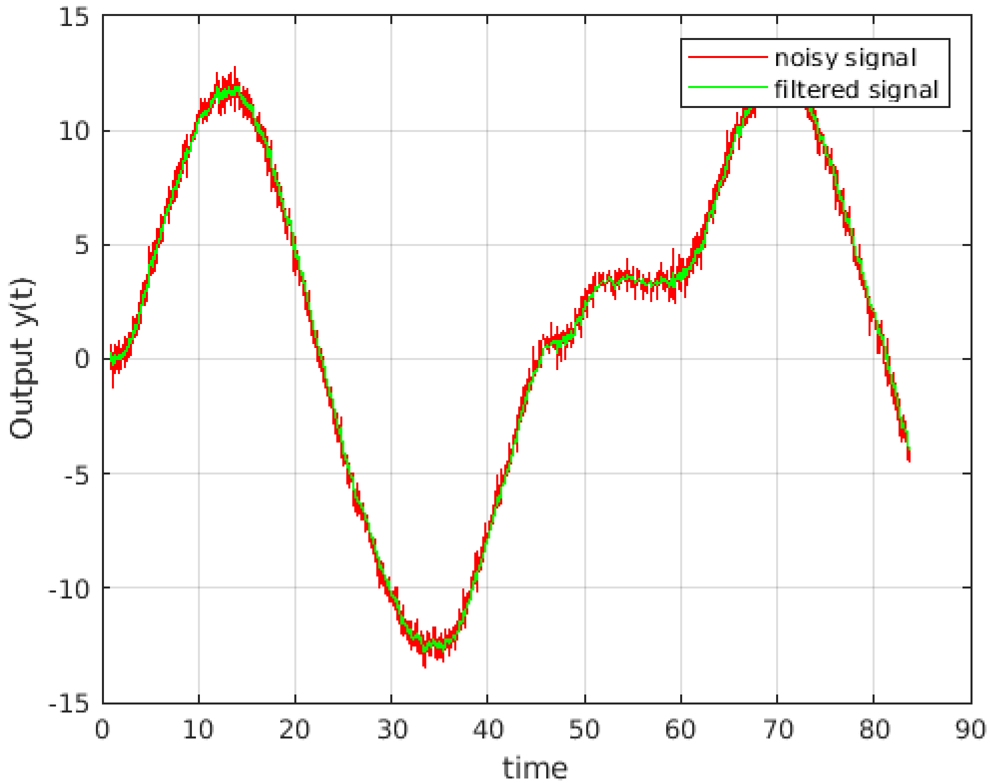
X vs Y filtered



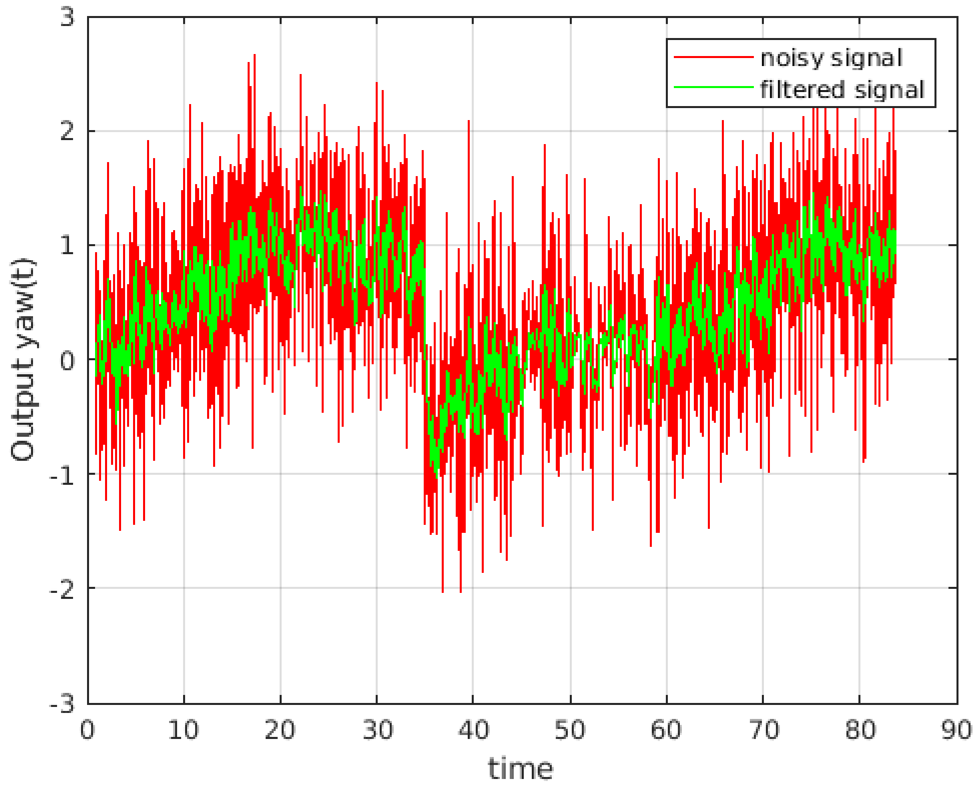
X vs Time filtered



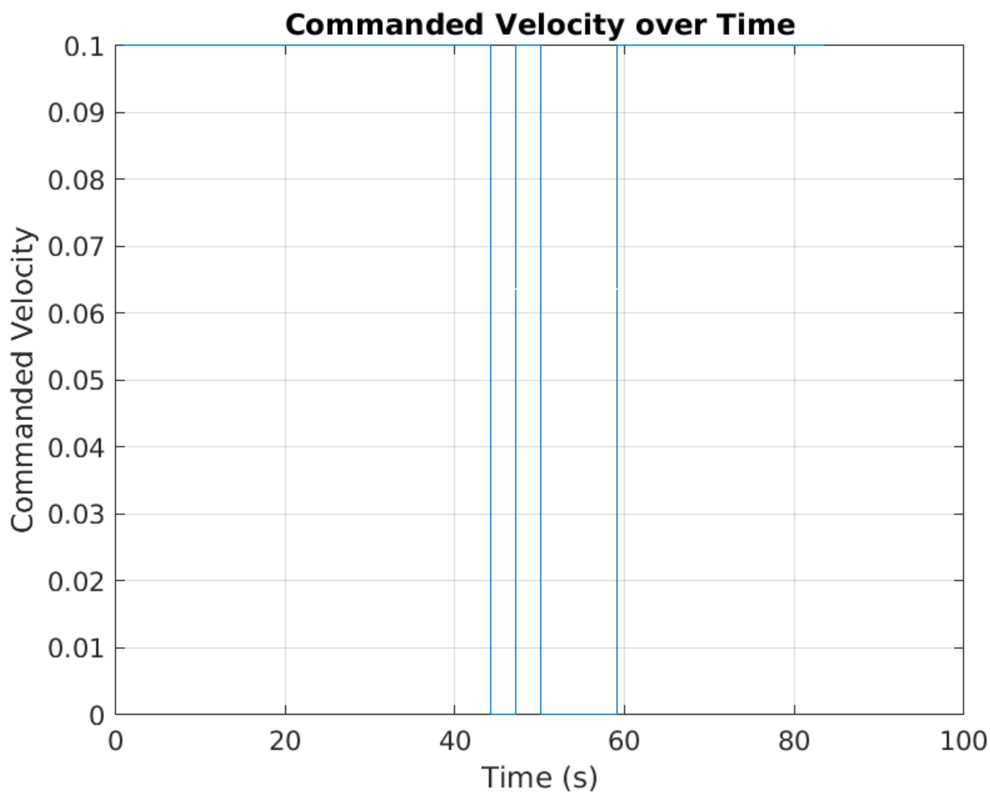
Y vs Time filtered



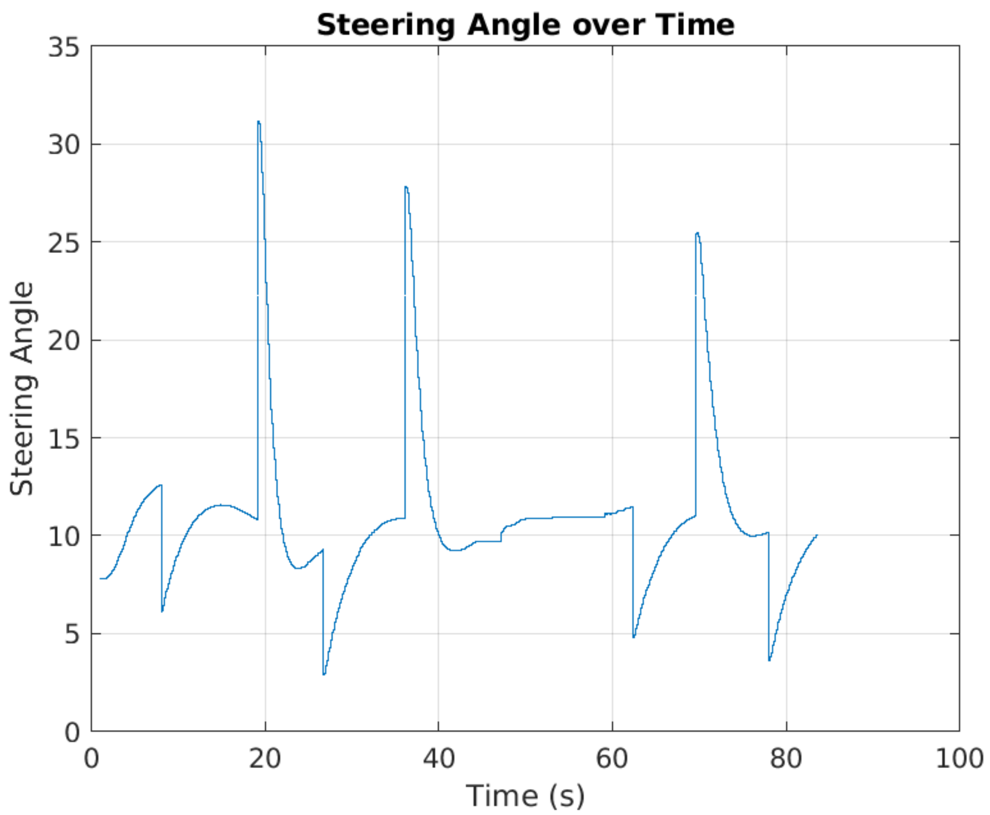
Yaw vs Time filtered



velocity_cmd vs time



steering_angle vs Time



Custom Track:

X vs Y

X vs Time

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
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/1a43vUpi7CoiALsGSzRfYnTOBFUI-1UiY?usp=drive_link

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