UNIVERSITÄT SIEGEN

Regelungs- und Steuerungstechnik RST

Studienarbeit

Trajectory Generation and Lane Control Simulation for Advanced Driver Assistance Systems

Name:	Chirag Angadi	Matr.-Nr.: 1725283
Supervisor:	Prof. Dr.-Ing. habil. Michael Gerke	
Supervising Assistant:	Dr.-Ing. Peter Will	

Date of Submission: 24. June 2025

Abstract

In the pursuit of safer and more intelligent transportation systems, Advanced Driver Assistance Systems (ADAS) have become a cornerstone in reducing human error and enhancing vehicle control. This project presents the development of a modular and simulation-driven framework for trajectory generation and lane control, two essential components of ADAS. The work emphasizes the use of simulation as a cost-effective, flexible, and risk-free means to validate control strategies prior to real-world implementation.

The framework is built using Python for algorithm development and WinFACT for vehicle dynamics simulation. The first phase of the project focuses on the generation of vehicle trajectories and corresponding velocity profiles. A set of customizable Python scripts is developed to define a variety of path geometries, including straight lines, clothoids, and circular arcs, by parametrizing road segments. These configurations allow for smooth, continuous path planning adaptable to various driving scenarios. Alongside the spatial path, velocity profiles are computed to reflect realistic vehicle motion while adhering to road constraints. This approach enables full control over trajectory generation from a centralized script, improving maintainability and scalability.

In the second phase, the generated trajectories are used to simulate lane control behaviour. A simplified single-track vehicle model is employed within the WinFACT simulation environment to represent vehicle dynamics. Python functions compute lateral deviation and velocity in real time at each simulation step, dynamically feeding these control parameters into the dynamic controller. This bidirectional integration between Python and WinFACT enables an interactive control loop, where algorithmic decisions and vehicle responses are evaluated synchronously.

The integration of Python and WinFACT proves to be a significant advantage, combining the computational power and versatility of Python with the structured simulation capabilities of WinFACT. This hybrid approach facilitates real-time analysis, rapid prototyping, and seamless modification of control logic, all within a unified environment.

By leveraging simulation as the foundation for development, this project successfully demonstrates a robust, modular approach to ADAS prototyping. The proposed framework not only supports detailed trajectory and lane control simulations but also lays the groundwork for future enhancements such as multi-lane navigation, dynamic lane changes, and real-time sensor integration. Ultimately, the project highlights the potential of integrated simulation environments to accelerate the design and validation of intelligent driver assistance technologies, contributing to the broader goal of safer and more autonomous road transport.

Contents

List of Figures

1 Introduction

Road safety remains a critical global concern, with a significant number of fatalities and injuries occurring due to traffic accidents each year. According to estimates by the World Health Organization (WHO), approximately 1.34 million lives are lost annually in road accidents, and nearly 50 million people suffer injuries [1]. Alarmingly, road-related fatalities are currently the ninth leading cause of death worldwide, with projections indicating a potential rise to sixth place if trends continue [2]. One of the primary challenges lies in managing increasing traffic volumes within often inadequate urban infrastructure, making modern transportation systems more complex and demanding.

Amidst these challenges, the automotive industry is undergoing a transformative shift, driven by innovations in automation and intelligent driving technologies. Advanced Driver Assistance Systems (ADAS) are at the forefront of this evolution, serving as an intermediate step between conventional driving and full vehicle autonomy. ADAS technologies are designed to assist drivers in real time, enhancing safety, improving comfort, and increasing overall driving efficiency. These systems encompass a wide range of functionalities, including adaptive cruise control, lane keeping, automatic emergency braking, traffic jam assistance, and blind-spot monitoring [3].

ADAS functionalities are generally categorized into five levels of autonomy:

- **Level 1:** Basic driver assistance such as adaptive cruise control and parking assist.
- **Level 2:** Partial automation including auto-steering, lane keeping, lane departure warning, and traffic jam assistance.
- **Level 3:** Conditional automation such as highway pilot and intersection navigation.
- **Level 4:** High automation under specific conditions, with minimal driver intervention.
- **Level 5:** Full automation where the vehicle can operate independently in all scenarios.

Within this hierarchy, trajectory generation and lane control play a foundational role. Trajectory generation involves computing the vehicle's intended path and associated velocity profile while accounting for road geometry, traffic rules, and vehicle dynamics. Lane control ensures that the vehicle maintains proper alignment within its designated lane using real-time data and control feedback mechanisms. Together, these components form the basis of effective path planning and execution in modern driver assistance systems.

However, developing and validating such systems in real-world scenarios pose significant safety risks, logistical challenges, and high costs, especially during the early stages of algorithm development when system behaviour may be unpredictable. Simulation environments provide a practical alternative, offering a controlled, safe, and repeatable platform to prototype and test ADAS functionalities under various conditions without endangering human lives or equipment.

This work proposes a simulation-based approach that integrates the flexibility of Python with the control-oriented capabilities of the WinFACT environment. Python is used to design and execute trajectory generation algorithms and perform real-time calculations of vehicle behaviour, including lateral deviations and speed adjustments. WinFACT, a dynamic simulation platform, is used to simulate vehicle motion using a single-track model that accurately reflects vehicle dynamics while maintaining simplicity. The combination of Python and WinFACT allows for a highly modular and interactive development environment, where control logic and simulation are tightly coupled to facilitate iterative improvements and rapid prototyping.

The core motivation behind this project is to demonstrate the effectiveness of such an integrated simulation framework in accelerating the development and validation of ADAS features. By modelling vehicle dynamics and control logic in a structured yet flexible environment, this work contributes to safer and more efficient methodologies for intelligent vehicle systems.

1.1 Problem Statement

As ADAS systems become more complex, there is an increasing need for robust tools to design and validate key functionalities such as trajectory tracking and lane control. These systems must ensure high reliability and responsiveness across diverse driving conditions while adhering to safety and performance standards. However, testing these control strategies in real vehicles introduces inherent risks and limitations, especially during the initial development phase.

Trajectory generation requires the vehicle to follow a pre-defined path with a feasible and dynamically achievable velocity profile. Simultaneously, lane control necessitates maintaining lateral stability and alignment within the lane boundaries, often relying on sensor feedback and vehicle dynamics models. Ensuring seamless interaction between these two subsystems is critical for the effectiveness of ADAS features.

Traditional simulation tools, while useful, often lack the flexibility to implement custom control logic or require significant effort to adapt. In contrast, Python offers powerful libraries for numerical computation, algorithm development, and data visualization, but does not natively support vehicle dynamics simulation. This gap creates a need for a hybrid solution that leverages the strengths of both environments.

1.2 Objectives

The primary objective of this project is to design and implement a modular, simulation-based framework for trajectory generation and lane control, using Python for algorithm development and WinFACT for simulating vehicle dynamics. The project aims to explore the potential of integrating both platforms to enhance flexibility, accuracy, and development speed for ADAS prototyping. Specifically, the goals are:

- To develop Python scripts capable of generating realistic trajectory paths and corresponding velocity profiles for path planning.
- To simulate vehicle lane control using a simplified single-track model that captures the essential lateral dynamics.
- To compute lateral deviations and velocity adjustments in real time during each simulation cycle.
- To integrate Python functions within WinFACT, ensuring seamless communication and synchronized execution between control logic and the simulation model.

This integrated simulation environment is intended to serve as a foundation for rapid development and evaluation of ADAS functionalities, providing a safer and more adaptable alternative to early-stage physical testing.

1.3 Report Structure

Chapter 1 introduces the motivation, context, and objectives of the project, highlighting the challenges in modern ADAS development and the rationale for using a simulation-based approach. Chapter 2 presents the methodologies employed, including the mathematical formulations and modelling techniques used for trajectory generation and lane control. This chapter also details the simulation architecture, describing how the control logic interacts with the vehicle model. Chapter 3 offers a comprehensive explanation of the Python codebase, including the structure of the main script, class interactions, and implementation details supported by a UML class diagram. Chapter 4 illustrates the outcomes of the trajectory generation module through multiple examples and evaluates the performance of the lane control simulation. Finally, Chapter 5 summarizes the findings, reflects on the contributions of the project, and outlines potential directions for future research and system expansion.

2 Methodologies

This chapter outlines the fundamental mathematical formulations applied in this work, primarily for generating trajectories composed of straight lines, clothoid transitions, and circular arcs. Additionally, it presents the simulation block architecture used for vehicle lane control, emphasizing the computation of lateral deviation and instantaneous velocity. The mathematical expressions for these computations are derived using geometric principles. The geometric modelling techniques applied in this work are referenced from [4, 5].

2.1 Trajectory Generation

The vehicle trajectory is generated by combining different geometric segments, including straight lines, circular arcs, and clothoid curves. Each segment type corresponds to a specific function designed to calculate the precise coordinates and curvature for that geometry. By sequentially connecting these segments, a smooth and continuous path is formed, accurately representing the intended route for the vehicle to follow.

Straight Line

A straight road segment is the simplest type of path where curvature remains zero throughout. The road is defined based on:

- A starting point (X_{start}, Y_{start})
- A direction or heading angle ϕ_s
- A fixed segment length l

For each unit length i from 0 to l, the center point of the lane is computed as:

$$X_{\text{center}} = X_{\text{start}} + i \cdot \cos(\phi_s), \quad Y_{\text{center}} = Y_{\text{start}} + i \cdot \sin(\phi_s) \tag{1}$$

The left and right lane edges are calculated by shifting the centerline laterally using half the lane width w:

$$X_{\text{left}} = X_{\text{center}} + \frac{w}{2} \cdot \cos\left(\phi_s + \frac{\pi}{2}\right), \quad Y_{\text{left}} = Y_{\text{center}} + \frac{w}{2} \cdot \sin\left(\phi_s + \frac{\pi}{2}\right) \tag{2}$$

$$X_{\text{right}} = X_{\text{center}} + \frac{w}{2} \cdot \cos\left(\phi_s - \frac{\pi}{2}\right), \quad Y_{\text{right}} = Y_{\text{center}} + \frac{w}{2} \cdot \sin\left(\phi_s - \frac{\pi}{2}\right) \tag{3}$$

For multiple lanes, the center and boundaries are offset laterally by integer multiples of the full lane width w. Since the path is straight, the curvature remains constant and equal to zero throughout:

$$\kappa = 0 \tag{4}$$

Circular Arc

A circular arc is a geometric segment of constant curvature. It represents a portion of a circle and is defined by a radius R, arc length l, and a starting heading angle ϕ_s.

The curvature of the arc is given by:

$$\kappa = \frac{1}{R} \tag{5}$$

Its angular sweep is:

$$\Delta\theta = \frac{l}{R} \tag{6}$$

To compute the arc center (M_x, M_y) relative to the start position (X_0, Y_0), we use:

$$M_x = X_0 + R\cos\left(\phi_s \pm \frac{\pi}{2}\right) \tag{7}$$

$$M_y = Y_0 + R\sin\left(\phi_s \pm \frac{\pi}{2}\right) \tag{8}$$

The $+\frac{\pi}{2}$ is used for anticlockwise arcs, and $-\frac{\pi}{2}$ for clockwise arcs.

For any point along the arc, the coordinates are determined by incrementing the angular value θ from θ_{start} to θ_{end} in small steps:

$$X = M_x + R\cos(\theta) \tag{9}$$

$$Y = M_y + R\sin(\theta) \tag{10}$$

Lanes are constructed by offsetting from the arc center using the same angle θ, with the radius adjusted accordingly for the centerline, left, and right lane boundaries.

Clothoid

A clothoid curve is a transition curve where the curvature changes linearly with the arc length. It is particularly useful for generating smooth transitions between straight and curved road segments.

The curvature as a function of the arc length s is defined as:

$$\kappa(s) = \frac{s}{A^2} \tag{11}$$

where $A = \sqrt{l \cdot R}$ is the clothoid parameter, derived from the total clothoid length l and the target radius of curvature R.

The angular deviation or deflection angle along the curve is given by:

$$\phi(s) = \frac{s^2}{2A^2} \tag{12}$$

To compute the global coordinates along the clothoid, the local x and y positions are approximated using truncated power series expansions. Introducing a scaled variable $\alpha = \frac{s^2}{2A^2}$, the coordinates are expressed as:

$$x(s) = a\sqrt{2\alpha} \sum_{n=0}^{N} \frac{(-1)^n \alpha^{2n}}{(4n+1)(2n)!} \tag{13}$$

$$y(s) = a\sqrt{2\alpha} \sum_{n=0}^{N} \frac{(-1)^n \alpha^{2n+1}}{(4n+3)(2n+1)!} \tag{14}$$

where $a = A$, and N denotes the number of terms used in the series (commonly, $N = 40$ provides sufficient accuracy). These equations approximate the Fresnel integrals [5] and yield local positions along the clothoid.

To obtain the global coordinates, the local positions are rotated and translated using the initial orientation ϕ_0 and starting position (X_0, Y_0):

$$X = X_0 + x\cos(\phi_0) - y\sin(\phi_0) \tag{15}$$

$$Y = Y_0 + x\sin(\phi_0) + y\cos(\phi_0) \tag{16}$$

Lanes are constructed by applying lateral offsets from the computed centreline. These are calculated by shifting in the perpendicular direction to the current heading θ, resulting in:

$$X_{\text{offset}} = X + w\cos\left(\theta \pm \frac{\pi}{2}\right) \tag{17}$$

$$Y_{\text{offset}} = Y + w\sin\left(\theta \pm \frac{\pi}{2}\right) \tag{18}$$

where w is either the half lane width or the full lane width, depending on whether the offset is from the centreline to the lane boundary or between adjacent lanes.

2.2 Lateral and longitudinal control of the vehicle

This section presents the block diagram of the simulation environment developed for the lateral and longitudinal control of the vehicle. Each component in the diagram is briefly described, including its respective inputs, outputs, and computational approach. The primary focus of this work is on the calculation of lateral distance and instantaneous speed, which are discussed in detail at the end of the section. Both lateral and longitudinal control are implemented using PID controllers: lateral control maintains a constant desired lateral distance, while longitudinal control ensures the vehicle follows a predefined speed profile based on the trajectory.

Structure of simulation

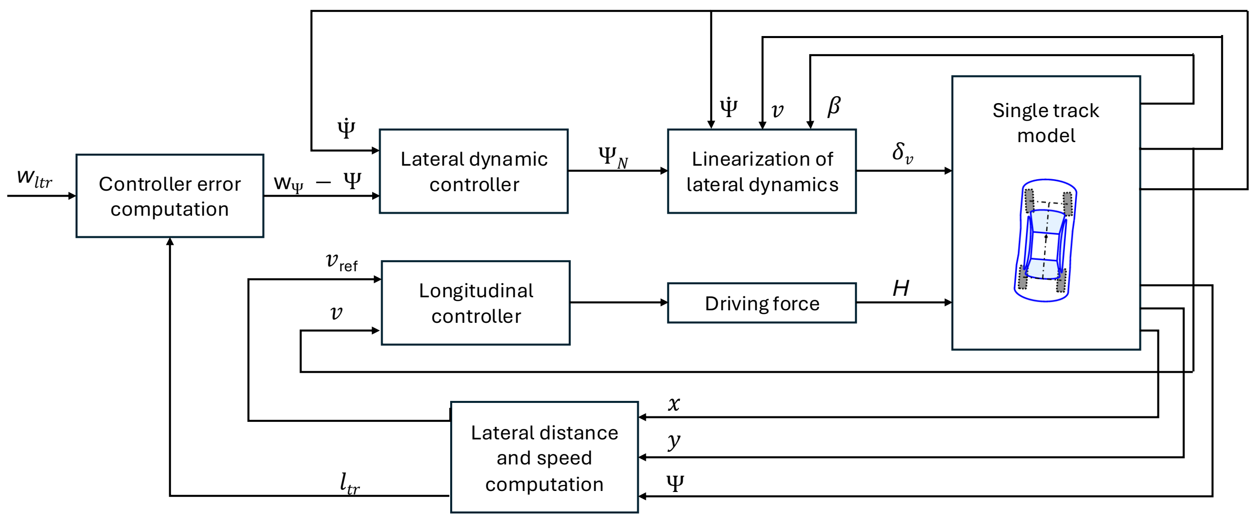

Figure 1: Structure of simulation

where,

Ψ	Yaw angle of the vehicle in, rad
w_Ψ	Desired yaw angle of the vehicle in, rad
Ψ_N	Nominal value of yaw angle of the vehicle in, rad
$\dot{\Psi}$	Yaw angular speed of the vehicle in, rad/s
v	Speed of the vehicle in, m/s
v_{ref}	Reference speed of the vehicle in, m/s
β	Slip angle of the vehicle in, rad
δv	Steering angle, in rad
x	COG X coordinate of vehicle
y	COG Y coordinate of vehicle
l_{tr}	Actual lateral distance
w_{ltr}	Desired lateral distance

Figure 1 illustrates the simulation block diagram implemented in the WinFACT software for simulating the lateral control of a vehicle. The control mechanism operates by regulating the vehicle's yaw angle through a dynamic controller that considers the desired yaw angle (w_Ψ), the actual yaw angle (Ψ), and the instantaneous yaw rate ($\dot{\Psi}$). The controller generates the nominal yaw angle (Ψ_N) as the control output.

This nominal yaw angle is subsequently converted into the steering angle (δ_v) using the "Linearisation of Lateral Dynamics" block. This step is essential, as the final control action is realized through the actuation of the vehicle's steering system. The resulting steering angle, along with the driving force, is then used in the "Single Track Model" block to compute the vehicle dynamics. For simplification, the complex vehicle dynamics are reduced to a single-track (or bicycle) model [6], which facilitates easier computation.

The vehicle model block computes various instantaneous parameters, including the position coordinates (x, y), yaw angle, yaw rate, slip angle, and vehicle speed. However, other parameters are beyond the scope of this work.

Using the computed coordinates of the vehicle's centre of gravity (x, y) and its yaw angle (Ψ), the lateral distance (l_{tr}) is calculated, which is explained in detail in a subsequent section. This block also determines the reference speed (v_{ref}) at each simulation step using linear interpolation between the speeds from the previous and next steps, as obtained from the trajectory data files.

Finally, the computed lateral distance (l_{tr}) and the desired lateral distance (w_{ltr}) are used in the controller's error computation block to evaluate the yaw angle error, defined as the difference $(w_\Psi - \Psi)$.

Why computation of lateral distance is required?
The dynamics of controller is a function of the yaw angle, however neither the desired yaw angle or instantaneous yaw angle is available as an absolute value. To determine these values, a coordinate transformations needs to be done for calculating the difference of desired and actual yaw angle as function of desired lateral distance and actual lateral distance.

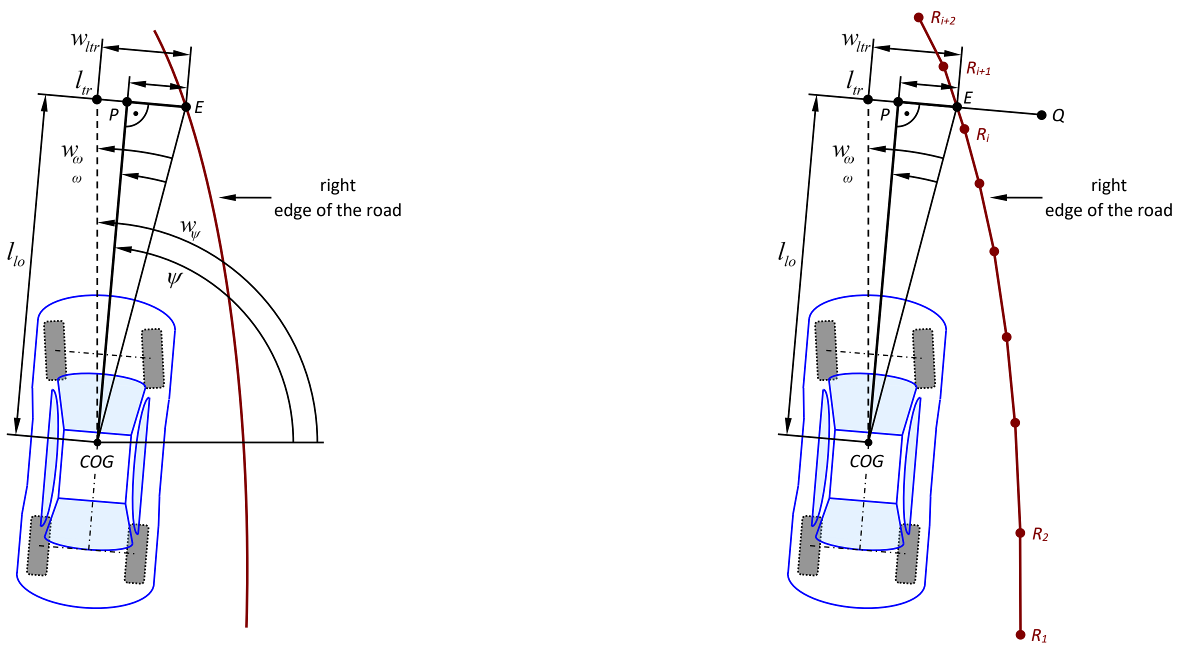

Figure 2: Geometric figure for lateral distance computation

where,

Ψ	Yaw angle of the vehicle in stationary coordinate system, *rad*
w_Ψ	Yaw angle of the vehicle in stationary coordinate system, *rad*
ω	Actual yaw angle of the vehicle in vehicle coordinate system, *rad*
w_ω	Desired yaw angle of the vehicle in vehicle coordinate system, *rad*
l_{tr}	Actual lateral distance between point P and E
w_{ltr}	Desired lateral distance
l_{lo}	Longitudinal distance between COG and point of lateral distance calculation

Figure 2 refers to geometric parameters used to compute difference in desired and actual yaw angles in general stationary coordinate system.

The following mathematical relationships can be deduced from the figure:

$$w_\Psi - \Psi = w_\omega - \omega \tag{19}$$

From trigonometric relationships, the angles w_ω and ω are computed

$$\omega = \arctan\left(\frac{l_{tr}}{l_{lo}}\right) \tag{20}$$

$$w_\omega = \arctan\left(\frac{w_{ltr}}{l_{lo}}\right) \tag{21}$$

From the above equations, the control error results in:

$$w_\Psi - \Psi = \arctan\left(\frac{w_{ltr}}{l_{lo}}\right) - \arctan\left(\frac{l_{tr}}{l_{lo}}\right) \tag{22}$$

Calculation of lateral Distance

Figure 2 illustrates the geometric setup used to compute the vehicle's lateral distance from the right road edge. The vehicle is represented with its current orientation and centre of gravity (COG). A look-ahead point P is projected in the direction of the vehicle's heading angle Ψ, and a perpendicular line is drawn through P toward the right edge of the road. The intersection point E between this line and the road edge segment (R_i, R_{i+1}) is used to calculate the shortest (signed) lateral distance. This setup also helps in determining the vehicle's longitudinal position along the path and the associated reference and maximum velocities.

To compute the lateral distance of the vehicle from the road edge:

- Let the vehicle's current position and yaw angle be (x, y, Ψ).
- A point P projected l_{lo} meters ahead in the heading direction is given by:

$$P = (p_x, p_y) = (x + l_{lo} \cdot \cos(\Psi),\ y + l_{lo} \cdot \sin(\Psi)) \tag{23}$$

- A perpendicular helper point Q is:

$$Q = (q_x, q_y) = (p_x + l_{lo} \cdot \cos(\Psi - \frac{\pi}{2}),\ p_y + l_{lo} \cdot \sin(\Psi - \frac{\pi}{2})) \tag{24}$$

- The right edge of the road is defined by discrete points $R_i = (x_i, y_i)$.
- The intersection point $E = (e_x, e_y)$ between line PQ and the segment (R_i, R_{i+1}) is computed using determinant-based Cramer's rule [7]:

$$\begin{aligned} e_x &= \frac{\rho_1 \cdot n_{y2} - \rho_2 \cdot n_{y1}}{n_{x1} \cdot n_{y2} - n_{y1} \cdot n_{x2}} \\ e_y &= \frac{n_{x1} \cdot \rho_2 - n_{x2} \cdot \rho_1}{n_{x1} \cdot n_{y2} - n_{x2} \cdot n_{y1}} \end{aligned} \tag{25}$$

  where (n_{x1}, n_{y1}) and (n_{x2}, n_{y2}) are the normal vectors to the road segment and PQ line, and ρ_1, ρ_2 are the respective Hessian distance parameters.
- The lateral distance is:

$$l_{tr} = \sqrt{(e_x - p_x)^2 + (e_y - p_y)^2} \tag{26}$$

- The sign of l_{tr} is determined by whether the vehicle is left or right of the road edge.

Calculation of instantaneous speed

The longitudinal progress along the route (S-position) is calculated as:

$$S_{\text{pos}} = S_i + \sqrt{(e_x - x_i)^2 + (e_y - y_i)^2} \tag{27}$$

Given S_{pos}, the reference and maximum speed are linearly interpolated:

$$V_{\text{ref}} = V_{\text{ref}_i} + \frac{V_{\text{ref}_{i+1}} - V_{\text{ref}_i}}{S_{i+1} - S_i} \cdot (S_{\text{pos}} - S_i) \tag{28}$$

$$V_{\max} = V_{\max_i} + \frac{V_{\max_{i+1}} - V_{\max_i}}{S_{i+1} - S_i} \cdot (S_{\text{pos}} - S_i) \tag{29}$$

3 Code Implementation

3.1 Trajectory Generation

This section provides a detailed explanation of the user-defined functions implemented for each type of road segment geometry. These functions are described individually, as they are crucial to the trajectory generation process. Other aspects of the code, such as file handling and speed profile generation, are not detailed here and can be referred to in their respective code files.

3.1.1 UML Diagram Description

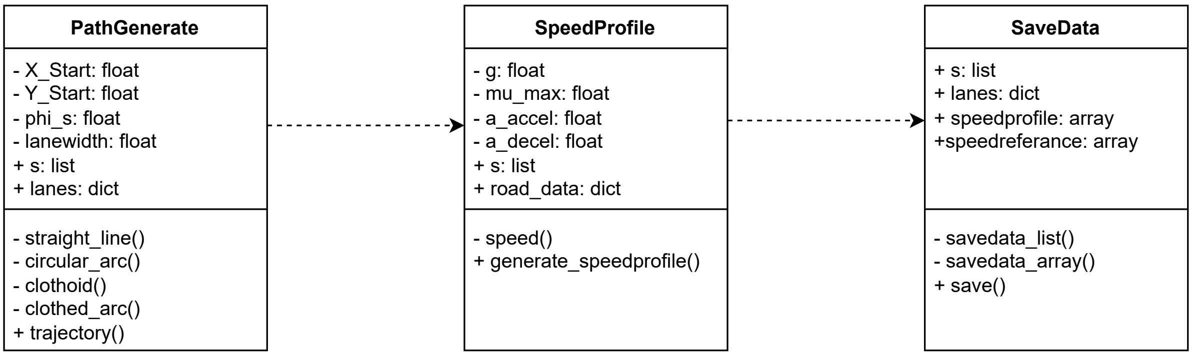

Figure 3: UML class diagram

Figure 3 illustrates UML class diagram represents three independent classes involved in the process of generating and saving path and speed profile data.

- **PathGenerate**
  This class is responsible for constructing and generating the overall path. It includes methods such as `straight_line()`, `circular_arc()`, and `clothoid_arc()` for generating individual segments based on specific geometric types. Additionally, the `trajectory()` method compiles and outputs the complete trajectory by combining all defined geometric segments, which can then be utilized by subsequent processing methods.

- **SpeedProfile**
  The `SpeedProfile` class takes the generated path as an input argument during its initialization. It uses this path data to generate speed profiles through its method `generate_speedprofile()`. This design illustrates a dependency on the output of the `PathGenerate` class.

- **SaveData**
  The `SaveData` class depends on both the path generated by `PathGenerate` and the speed profile data produced by `SpeedProfile`. It accepts these as inputs during initialization and provides a method `save_to_txt()` to save this combined data locally as text files.

This separation of concerns ensures modularity, allowing each class to focus on a specific task while communicating via well-defined data exchanges. The dependencies are represented using directed arrows, indicating how data flows from one class to another.

3.1.2 Straight Line

The implementation of a straight road segment involves computing and storing the geometric coordinates iteratively, starting from a given point and extending in the direction of the heading angle. This is achieved using the function `straight_line(self, l, X_start, Y_start, phi_s)`, where the parameters are: the segment length l, starting coordinates $(X_{\text{start}}, Y_{\text{start}})$, and initial heading angle ϕ_s. The function generates a multi-lane road trajectory and returns the end coordinates $(X_{\text{end}}, Y_{\text{end}})$ and final heading ϕ_e.

The following steps summarizes the procedure:

1. Initialization (Code lines 2–5)
The road segment is registered in the data structure with its type identified as a `Straight line`. The curvature at both the beginning and end is set to zero. The segment index is also initialized.

2. Outer Loop for Lane Geometry Generation (Code lines 7–23)
A loop iterates from 0 to the total segment length. In each iteration:

- The longitudinal distance is incremented.
- The centreline point is calculated based on the heading angle from the start position.
- Left and right boundaries of the lane are computed by laterally offsetting from the centreline using half the lane width in the perpendicular direction.

3. Inner Loop for Multi-Lane Construction (Code lines 16–22)
If multiple lanes exist, the inner loop constructs additional lanes by:

- Using the left edge of the previous lane as the right edge of the current lane.
- Computing the centreline and left edge by further offsetting from the base centreline.

4. Finalization (Code lines 25–31)
At each step, the calculated centreline and boundary points are stored along with the corresponding arc length and curvature (zero for a straight segment). Upon completion, the final position and unchanged heading are recorded, and the road data structure is updated with the segment's end index and counter increment.

Listing 1: Function to compute and generate straight line trajectory

```
1  def straight_line(self, l, X_start, Y_start, phi_s):
2      self.road_data[self.road_data_indexcount] = {}
3      self.road_data[self.road_data_indexcount]['typ'] = 'Straight line'
4      self.road_data[self.road_data_indexcount]['curvaturebegin'] = 0
5      self.road_data[self.road_data_indexcount]['sbegin'] = 0
6
7      for i in range(l + 1):                  # loop to calculate X,Y iteratively till
           length(l)
8          self.S.append(self.indexcount)
9          self.curvature.append(0)
10         self.lanes[0]['X_center'].append(X_start + i * cos(phi_s))
11         self.lanes[0]['Y_center'].append(Y_start + i * sin(phi_s))
12         self.lanes[0]['X_left'].append(self.lanes[0]['X_center'][self.indexcount] +
               self.lanewidth_half * cos(phi_s + pi / 2))
```

```
        self.lanes[0]['Y_left'].append(self.lanes[0]['Y_center'][self.indexcount] +
            self.lanewidth_half * sin(phi_s + pi / 2))
        self.lanes[0]['X_right'].append(self.lanes[0]['X_center'][self.indexcount] +
             self.lanewidth_half * cos(phi_s - pi / 2))
        self.lanes[0]['Y_right'].append(self.lanes[0]['Y_center'][self.indexcount] +
             self.lanewidth_half * sin(phi_s - pi / 2))
        for j in range(1, self.Number_of_lanes):
            self.lanes[j]['X_right'] = self.lanes[j-1]['X_left']
            self.lanes[j]['Y_right'] = self.lanes[j-1]['Y_left']
            self.lanes[j]['X_center'].append(self.lanes[0]['X_center'][self.
                indexcount] + (j * self.lanewidth) * cos(phi_s + pi / 2))
            self.lanes[j]['Y_center'].append(self.lanes[0]['Y_center'][self.
                indexcount] + (j * self.lanewidth) * sin(phi_s + pi / 2))
            self.lanes[j]['X_left'].append(self.lanes[0]['X_center'][self.indexcount]
                 + (j * self.lanewidth + self.lanewidth_half) * cos(phi_s + pi / 2))
            self.lanes[j]['Y_left'].append(self.lanes[0]['Y_center'][self.indexcount]
                 + (j * self.lanewidth + self.lanewidth_half) * sin(phi_s + pi / 2))
        self.indexcount += 1

    self.road_data[self.road_data_indexcount]['send'] = self.indexcount - 1
    self.road_data[self.road_data_indexcount]['curvatureend'] = 0
    self.road_data_indexcount += 1

    phi_e = phi_s
    X_end = self.lanes[0]['X_center'][self.indexcount - 1]
    Y_end = self.lanes[0]['Y_center'][self.indexcount - 1]

    return X_end, Y_end, phi_e
```

3.1.3 Circular Arc

The `circular_arc(self, l, R, X_start, Y_start, phi_s, direction)` function generates a curved road segment by discretizing a circular arc into small angular steps. At each step, it calculates the centreline and lane boundary points for each lane based on the turning direction, arc radius, and initial position and heading. The function supports multiple lanes and correctly offsets each lane's geometry from the arc centre, enabling the modelling of both clockwise and anticlockwise curved road sections. It returns the end coordinates and orientation of the arc segment for seamless continuation of subsequent road elements.

1. Initialization (Code lines 1–9)
The function begins by adding a new entry to the variable `road_data`. It sets the segment type to "CircularArc" and stores the starting index of the segment. The curvature is computed based on the turn direction: negative for clockwise and positive for anticlockwise turns. This curvature value is stored as the beginning curvature for the road segment.

2. Determining Arc Center (Code lines 11–13)
The centre of the circular arc is calculated by shifting the start point in a direction perpendicular to the road's initial orientation. This is done to establish the centre of the turning circle from which all subsequent points will be generated.

3. Computing Angular Range (Code lines 15–17)
The angular range required to draw the arc is computed based on the arc length and radius. The start and end angles for the arc are determined accordingly, considering the direction of the turn.

4. Loop for Computing Arc Points (Code lines 19–43)
The function iterates over the angular range using a custom float-based loop. For each step:

- The index and curvature values are updated and stored.
- The centreline coordinates of lane 0 are computed using the arc centre and current angle.
- Left and right boundary coordinates for lane 0 are calculated by adjusting the radius based on half the lane width.

5. Loop for Additional Lane Geometry (Code lines 31–41)
For lanes beyond lane 0, the right boundary of each lane is inherited from the left boundary of the preceding lane. Then, the centre and left boundary points of the current lane are computed by adjusting the radius appropriately using the lane width. This ensures spatial continuity and alignment across all lanes.

6. Finalization (Code lines 43–54)
The segment end index and final curvature value are recorded. The segment radius is stored with a sign indicating the turn direction. Finally, the end position and updated orientation angle are calculated and returned, marking the terminal point of the circular arc.

Listing 2: Function to compute and generate circular arc trajectory

```
1  def circular_arc(self, l, R, X_start, Y_start, phi_s, direction):
2      self.road_data[self.road_data_indexcount] = {}
3      self.road_data[self.road_data_indexcount]['typ'] = 'CircularArc'
4      self.road_data[self.road_data_indexcount]['sbegin'] = self.indexcount
5
6      # Direction: anticlockwise = +1 curvature, clockwise = -1 curvature
7      sign = -1 if direction == 'clockwise' else 1
8      curvature = sign / R
9      self.road_data[self.road_data_indexcount]['curvaturebegin'] = curvature
10
11     # Arc center
12     Mx = X_start + R * cos(phi_s + sign * pi / 2)
13     My = Y_start + R * sin(phi_s + sign * pi / 2)
14
15     # Start and end angle for the arc
16     theta_start = phi_s - sign * pi / 2 + sign * (1 / R)
17     theta_end = phi_s - sign * pi / 2 + sign * (l / R)
18
19     for i in self.range_float(theta_start, theta_end, sign * (1 / R)):
20         self.S.append(self.indexcount)
21         self.curvature.append(curvature)
22
23         # Lane 0
24         self.lanes[0]['X_center'].append(Mx + R * cos(i))
25         self.lanes[0]['Y_center'].append(My + R * sin(i))
```

```
            self.lanes[0]['X_left'].append(Mx + (R - sign * self.lanewidth_half) *
                cos(i))
            self.lanes[0]['Y_left'].append(My + (R - sign * self.lanewidth_half) *
                sin(i))
            self.lanes[0]['X_right'].append(Mx + (R + sign * self.lanewidth_half) *
                cos(i))
            self.lanes[0]['Y_right'].append(My + (R + sign * self.lanewidth_half) *
                sin(i))

            for j in range(1, self.Number_of_lanes):
                self.lanes[j]['X_right'] = self.lanes[j - 1]['X_left']
                self.lanes[j]['Y_right'] = self.lanes[j - 1]['Y_left']

                offset_center = R - sign * j * self.lanewidth
                offset_left = R - sign * (j * self.lanewidth + self.lanewidth_half)

                self.lanes[j]['X_center'].append(Mx + offset_center * cos(i))
                self.lanes[j]['Y_center'].append(My + offset_center * sin(i))
                self.lanes[j]['X_left'].append(Mx + offset_left * cos(i))
                self.lanes[j]['Y_left'].append(My + offset_left * sin(i))

            self.indexcount += 1

        self.road_data[self.road_data_indexcount]['send'] = self.indexcount - 1
        self.road_data[self.road_data_indexcount]['curvatureend'] = curvature
        self.road_data[self.road_data_indexcount]['R'] = sign * R
        self.road_data_indexcount += 1

        phi_e = phi_s + sign * l / R
        X_end = self.lanes[0]['X_center'][self.indexcount - 1]
        Y_end = self.lanes[0]['Y_center'][self.indexcount - 1]

        return X_end, Y_end, phi_e
```

3.1.4 Clothoid Arc

The clothoid transition is implemented in two modes: from a straight line to a circular arc, and from a circular arc back to a straight line. Both use the same underlying logic with different directions of curvature progression.

def clothoid(self, alpha, a)
The function calculates a point on a clothoid (Euler spiral) using a truncated Taylor series approximation of the Fresnel integrals. It iteratively computes two summations for the x and y components over 41 terms, representing the cosine and sine integrals, respectively. These are then scaled by a factor involving the input parameters `alpha` and `a` to obtain the coordinates of the point on the clothoid. The function returns the computed x and y values as a list.

def clothoid_arc(self, l, R, X_start, Y_start, phi_s, direction, transition)
This function generates a multi-lane road segment following a clothoid (spiral) curve. It takes as

input the segment length l, target curvature radius R, starting coordinates (X_start, Y_start), initial heading angle ϕ_s, the turning direction (anticlockwise or clockwise), and clothoid used in transition from (line to circle or circle to line). The function calculates the centreline and lane boundaries by incrementally computing clothoid points and laterally offsetting them for multiple lanes.

1. Initialization and Parameters

The transition begins by storing metadata for the road segment. The clothoid parameter A is computed from the input length l and radius R at the end of the clothoid:

$$A = \sqrt{l \cdot R}, \quad \phi = \frac{l^2}{2A^2}$$

2. Line to Circle Transition

If the transition is from a straight line to a circular arc:

- Curvature starts from 0 and increases linearly: $\kappa = \frac{s}{A^2}$
- A loop iterates over each unit step from 1 to l
- At each step:
  - Compute the curvature and angle: $\theta = \frac{s^2}{2A^2}$
  - Call the helper function `clothoid()` to get coordinates in local frame
  - Rotate and translate these coordinates to global frame using heading ϕ_s
  - Compute left and right boundary points using offset angles $\phi_s \pm \theta \pm \frac{\pi}{2}$
- For multiple lanes, lateral offsets are added perpendicular to the current heading
- Update index and store curvature at each point

3. Circle to Line Transition

If the transition is from a circular arc to a straight line:

- Curvature decreases from $1/R$ to 0 in reverse order
- First, compute where the clothoid begins using the total transformation of a full clothoid
- Then loop in reverse from $l - 1$ to 0:
  - Compute local coordinates using `clothoid()`
  - Apply rotation based on the offset angle (accounting for the full arc sweep)
  - Compute boundary points and lane centrelines similarly
- Store curvature and spatial position data at each step

4. Finalization

After the loop, curvature end values and final index values are stored. The final global position and heading angle are computed and returned:

$$\phi_e = \phi_s \pm \phi$$

$$(X_{\text{end}}, Y_{\text{end}}) = \text{Last computed lane centre point}$$

This implementation enables realistic and smooth clothoid transitions suitable for highways, rail tracks, and robotic path planning.

Listing 3: Function to compute and generate clothoid arc trajectory

```python
def clothoid(self, alpha, a):
    sum_x = 0
    sum_y = 0

    for count in range(41):
        sum_x += ((-1) ** count * alpha ** (2 * count)) / ((4 * count + 1) *
            factorial(2 * count))
        sum_y += ((-1) ** count * alpha ** (2 * count + 1)) / ((4 * count + 3) *
            factorial(2 * count + 1))

    x = a * sqrt(2 * alpha) * sum_x
    y = a * sqrt(2 * alpha) * sum_y

    result = [x, y]
    return result

def clothoid_arc(self, l, R, X_start, Y_start, phi_s, direction, transition):
    self.road_data[self.road_data_indexcount] = {}
    self.road_data[self.road_data_indexcount]['typ'] = 'Clothoid'
    self.road_data[self.road_data_indexcount]['sbegin'] = self.indexcount

    A = sqrt(l * R)
    phi = (l ** 2) / (2 * (A ** 2))
    phi_sign = 1 if direction == 'anticlockwise' else -1
    curvature_sign = 1 if direction == 'anticlockwise' else -1

    if transition == 'line_to_circle':
        self.road_data[self.road_data_indexcount]['curvaturebegin'] = 0

        for i in range(1, l + 1):
            s = i
            curv = curvature_sign * i / (A ** 2)
            theta = phi_sign * (i ** 2) / (2 * A ** 2)

            self.S.append(self.indexcount)
            self.curvature.append(curv)
            xy = self.clothoid((i ** 2) / (2 * A ** 2), A)
```

```
            x = xy[0]
            y = xy[1]

            X_center = X_start + x * cos(phi_s) - y * sin(phi_s) if direction == '
                anticlockwise' else X_start + x * cos(phi_s) + y * sin(phi_s)
            Y_center = Y_start + x * sin(phi_s) + y * cos(phi_s) if direction == '
                anticlockwise' else Y_start + x * sin(phi_s) - y * cos(phi_s)

            self.lanes[0]['X_center'].append(X_center)
            self.lanes[0]['Y_center'].append(Y_center)

            angle = phi_s + theta if direction == 'anticlockwise' else phi_s - theta

            self.lanes[0]['X_left'].append(X_center + self.lanewidth_half * cos(angle
                 + pi / 2))
            self.lanes[0]['Y_left'].append(Y_center + self.lanewidth_half * sin(angle
                 + pi / 2))
            self.lanes[0]['X_right'].append(X_center + self.lanewidth_half * cos(
                angle - pi / 2))
            self.lanes[0]['Y_right'].append(Y_center + self.lanewidth_half * sin(
                angle - pi / 2))

            for j in range(1, self.Number_of_lanes):
                self.lanes[j]['X_right'] = self.lanes[j - 1]['X_left']
                self.lanes[j]['Y_right'] = self.lanes[j - 1]['Y_left']
                offset = self.lanewidth_half + j * self.lanewidth
                self.lanes[j]['X_center'].append(X_center + j * self.lanewidth * cos(
                    angle + pi / 2))
                self.lanes[j]['Y_center'].append(Y_center + j * self.lanewidth * sin(
                    angle + pi / 2))
                self.lanes[j]['X_left'].append(X_center + offset * cos(angle + pi /
                    2))
                self.lanes[j]['Y_left'].append(Y_center + offset * sin(angle + pi /
                    2))

            self.indexcount += 1

        self.road_data[self.road_data_indexcount]['curvatureend'] = curvature_sign /
             R

    elif transition == 'circle_to_line':
        self.road_data[self.road_data_indexcount]['curvaturebegin'] = curvature_sign
             / R

        xy = self.clothoid((l ** 2) / (2 * A ** 2), A)
        x = xy[0]
        y = xy[1]

        angle_shift = phi_sign * phi
        X_start_clothoide = X_start + x * cos(phi_s + angle_shift) + y * sin(phi_s +
             angle_shift) if direction == 'anticlockwise' else X_start + x * cos(
```

```
            phi_s + angle_shift) - y * sin(phi_s + angle_shift)
        Y_start_clothoide = Y_start + x * sin(phi_s + angle_shift) - y * cos(phi_s + angle_shift) if direction == 'anticlockwise' else Y_start + x * sin(phi_s + angle_shift) + y * cos(phi_s + angle_shift)

        for i in range(l - 1, -1, -1):
            curv = curvature_sign * i / (A ** 2)
            theta = phi_sign * (i ** 2) / (2 * A ** 2)
            self.S.append(self.indexcount)
            self.curvature.append(curv)
            xy = self.clothoid((i ** 2) / (2 * A ** 2), A)
            x = xy[0]
            y = xy[1]

            phi_offset = phi_s + phi_sign * phi - pi

            X_center = X_start_clothoide + x * cos(phi_offset) + y * sin(phi_offset) if direction == 'anticlockwise' else X_start_clothoide + x * cos(phi_offset) - y * sin(phi_offset)
            Y_center = Y_start_clothoide + x * sin(phi_offset) - y * cos(phi_offset) if direction == 'anticlockwise' else Y_start_clothoide + x * sin(phi_offset) + y * cos(phi_offset)

            self.lanes[0]['X_center'].append(X_center)
            self.lanes[0]['Y_center'].append(Y_center)

            angle = phi_offset - theta if direction == 'anticlockwise' else phi_offset + theta

            self.lanes[0]['X_right'].append(X_center + self.lanewidth_half * cos(angle + pi / 2))
            self.lanes[0]['Y_right'].append(Y_center + self.lanewidth_half * sin(angle + pi / 2))
            self.lanes[0]['X_left'].append(X_center + self.lanewidth_half * cos(angle - pi / 2))
            self.lanes[0]['Y_left'].append(Y_center + self.lanewidth_half * sin(angle - pi / 2))

            for j in range(1, self.Number_of_lanes):
                self.lanes[j]['X_right'] = self.lanes[j - 1]['X_left']
                self.lanes[j]['Y_right'] = self.lanes[j - 1]['Y_left']
                offset = self.lanewidth_half + j * self.lanewidth
                self.lanes[j]['X_center'].append(X_center + j * self.lanewidth * cos(angle - pi / 2))
                self.lanes[j]['Y_center'].append(Y_center + j * self.lanewidth * sin(angle - pi / 2))
                self.lanes[j]['X_left'].append(X_center + offset * cos(angle - pi / 2))
                self.lanes[j]['Y_left'].append(Y_center + offset * sin(angle - pi / 2))

```

```
            self.indexcount += 1

        self.road_data[self.road_data_indexcount]['curvatureend'] = 0

    self.road_data[self.road_data_indexcount]['send'] = self.indexcount - 1
    self.road_data[self.road_data_indexcount]['A'] = A
    self.road_data_indexcount += 1

    phi_e = phi_s + phi_sign * phi
    X_end = self.lanes[0]['X_center'][self.indexcount - 1]
    Y_end = self.lanes[0]['Y_center'][self.indexcount - 1]

    return X_end, Y_end, phi_e
```

3.2 Computation of Lateral Distance and Vehicle Speed

This section introduces the Python code developed to calculate the lateral deviation and instantaneous speed of the vehicle at each simulation step. The code is integrated into the WinFACT simulation environment within the "Lateral Distance and Speed Computation Block" (see Figure 1). It utilizes the precomputed trajectory and velocity profile data and processes them iteratively during the simulation as outlined below:

1. File Operations Function (Code lines: 14–22)

The `file_operations()` function is designed to read numerical data from text files. It replaces commas with decimal points to ensure proper float conversion, then reads each line and appends the float values to the given array.

2. Main Function Definition (Code line: 25)

The function `main()` is defined to compute the lateral distance from the reference path, the longitudinal position along the trajectory, and both the reference and maximum velocities of the vehicle. Several global variables are initialized to retain values between function calls.

3. Variable Initialization (Code lines: 26–54)

File paths for the X and Y coordinates of the route, arc-length, maximum velocity, and reference velocity are defined. Variables used in geometric calculations such as coordinates, indices, vectors, distances, and outputs are initialized. A constant value of π is also defined.

4. File Reading and Validation (Code lines: 57–105)

When the simulation begins (`Init == 1`), the code attempts to read each input file using `file_operations()`. If all files are successfully read and the arrays are of equal length, a flag `data_read` is set to 1. If any file fails or lengths mismatch, error messages are printed and the simulation is set to terminate.

5. Output Initialization (Code lines: 108–111)

Default values are assigned for lateral distance and velocity parameters. These are used as fallbacks in case the input files could not be read.

6. Simulation Termination Handling (Code lines: 113–115)

If `Terminate == 1`, the function prints a termination message and calls the `Terminate()` function to halt simulation execution.

7. Main Calculation Logic (Code lines: 117–215)

If the input data has been successfully loaded (`data_read == 1`), the following calculations are performed for each simulation step:

- **Point Projection (Code lines: 121–126)**: A point P is projected 10 meters ahead in the heading direction. A helper point Q is constructed perpendicular to the heading.
- **Closest Segment Identification (Code lines: 129–140)**: A loop iterates through route segments to find the closest road segment index where the vehicle lies to the left of the segment.
- **Intersection Calculation (Code lines: 143–174)**: Vector components and lengths are computed for both the route segment and the PQ line. The intersection point (foot of the perpendicular) is found using Cramer's Rule.
- **Lateral Distance Computation (Code lines: 177–182)**: The lateral distance is calculated as the Euclidean distance from point P to the intersection point E. The sign is adjusted based on whether the point is to the left or right of the path.
- **Arc Length Positioning (Code lines: 185–190)**: The longitudinal position along the route is determined using the segment index and distance from the segment base, adjusted by a fixed correction.
- **Clamping Boundary Values (Code lines: 192–198)**: The computed position is checked to ensure it stays within the valid arc-length range of the trajectory.

8. Speed Computation Using Interpolation (Code lines: 200–215)

The current position along the trajectory is used to locate surrounding indices in the reference and maximum velocity arrays. Linear interpolation is then applied to compute smooth values for both the reference and maximum velocity at the current position.

9. Fallback Output Assignment (Code lines: 217–221)

If the input data was not successfully read, fallback values for lateral distance and speed are reassigned.

10. Output Assignment (Code lines: 224–227)

The final computed or fallback values for lateral distance, longitudinal position, reference speed, and maximum speed are written to output variables. An additional flag is set to indicate whether data reading was successful (`Out_5.value`).

Listing 4: Lateral distance and speed etimation block in simulation environment

```
import math

# Global variables
X_route = []
Y_route = []
S_route = []
Vmax_route = []
Vref_route = []
data_read = 0
i_small_old = 0
sp_smaller_old = 0

# file_operations is user defined function to read file, replace ',' to '.' if it
    contains and to store in a list.
def file_operations(datapath, array):
    with open(datapath, 'r') as f:
        content = f.read()
        modified_content = content.replace(',' , '.')
    with open(datapath, 'w') as f:
        f.write(modified_content)
    with open(datapath, 'r') as f:
        for org_line in f:
            array.append(float(org_line))

# Below is main function which calculates lateral distance, position, referance
    velocity and maximun velocity
def main(Init, Terminate, psi, pos_x, pos_y):
    global X_route, Y_route, S_route, Vmax_route, Vref_route, data_read, i_small_old
        , sp_smaller_old

    # Variables for file operations
    dataPath_X = "Strecke_rechts_X.txt"
    dataPath_Y = "Strecke_rechts_Y.txt"
    dataPath_S = "V-Profil_pos.txt"
    dataPath_Vmax = "V-Profil_max.txt"
    dataPath_Vref = "V-Profil_ref.txt"
    x_read, y_read, s_read, vmax_read, vref_read = 0, 0, 0, 0, 0

    # Variables for calculations
    p_x, p_y = 0, 0
    q_x, q_y = 0, 0
    i, i_smaller = 0, 0
    condition = 0
    dx1, dy1, dx2, dy2, l1, l2 = 0, 0, 0, 0, 0, 0
    nx1, ny1, nx2, ny2 = 0, 0, 0, 0
    rho1, rho2 = 0, 0
    det_1, det_2, det_3, det_4 = 0, 0, 0, 0
    e_x, e_y = 0, 0
    lateral_distance = 0

```

```python
    # Position, speed reference values, and maximum speed variables
    sp, sp_smaller = 0, 0
    pos_S, pos_V_ref, pos_V_max = 0, 0, 0
    m_V_ref_N, m_V_max_N, m_V_max_ref_Z, m_V_ref, m_V_max = 0, 0, 0, 0, 0

    # Constants
    pi = 3.1415926535

    # During the initialization of the simulation Init = 1
    if Init == 1:

        # Reading the file containing the complete trajectory in X coordinates and storing in new array
        try:
            file_operations(dataPath_X, X_route)
            x_read = 1
        except FileNotFoundError:
            pass

        # Reading the file containing the complete trajectory in Y coordinates and storing in new array
        try:
            file_operations(dataPath_Y, Y_route)
            y_read = 1
        except FileNotFoundError:
            pass

        # Reading the file containing the indices and storing in new array
        try:
            file_operations(dataPath_S, S_route)
            s_read = 1
        except FileNotFoundError:
            pass

        # Reading the file containing the maximum speed and storing in new array
        try:
            file_operations(dataPath_Vmax, Vmax_route)
            vmax_read = 1
        except FileNotFoundError:
            pass

        # Reading the file containing the reference speed and storing in new array
        try:
            file_operations(dataPath_Vref, Vref_route)
            vref_read = 1
        except FileNotFoundError:
            pass

        # Check if all files were successfully read
        if x_read == 1 and y_read == 1 and s_read == 1 and vmax_read == 1 and vref_read == 1:
```

```python
            if len(X_route) == len(Y_route) == len(S_route) == len(Vmax_route) == len(Vref_route):
                data_read = 1
            else:
                print("One of the data files is faulty!")
                print("Simulation will be terminated!")
                data_read = 0
        else:
            print("One of the data files could not be read!")
            print("Simulation will be terminated!")
            data_read = 0

        # Set initial values for outputs
        lateral_distance = 1.5
        pos_S = 0
        pos_V_ref = 100 / 3.6
        pos_V_max = 100 / 3.6

    elif Terminate == 1:
        print("Termination in progress!")
        Terminate()

    if data_read == 1:
        # The following lines of code are executed in every simulation step after the data files are completely read

        # Calculate point P
        p_x = pos_x + 10 * math.cos(psi)
        p_y = pos_y + 10 * math.sin(psi)

        # Calculate helper point Q
        q_x = p_x + 10 * math.cos(psi - (pi / 2))
        q_y = p_y + 10 * math.sin(psi - (pi / 2))

        # Find the maximum value of the index i for which the condition is less than or equal to zero ...
        i = i_small_old
        condition = (q_x - p_x) * (Y_route[i] - p_y) - (X_route[i] - p_x) * (q_y - p_y)

        while condition <= 0 and i <= len(X_route) - 2:
            if condition <= 0:
                i_smaller = i
            i += 1
            if i >= len(X_route):
                i = len(X_route) - 1
            condition = (q_x - p_x) * (Y_route[i] - p_y) - (X_route[i] - p_x) * (q_y - p_y)

        i_small_old = i_smaller # Save the current "i_smaller" for the next run

```

```
        # Components of the line by connecting the two vertices of the "edge of the road"
        dx1 = X_route[i_smaller + 1] - X_route[i_smaller]
        dy1 = Y_route[i_smaller + 1] - Y_route[i_smaller]

        # Components of the line by connecting points P and Q
        dx2 = q_x - p_x
        dy2 = q_y - p_y

        # Calculating the length of the connecting lines
        l1 = math.sqrt(dx1 * dx1 + dy1 * dy1)
        l2 = math.sqrt(dx2 * dx2 + dy2 * dy2)

        # Components of the normal vector, perpendicular to the polygon chain
        nx1 = dy1 / l1
        ny1 = -1 * (dx1 / l1)

        # Components of the normal vector, perpendicular to the line PQ
        nx2 = dy2 / l2
        ny2 = -1 * (dx2 / l2)

        # Calculate the distance parameter Rho for the Hessian normal form of the two straight lines
        rho1 = nx1 * X_route[i_smaller] + ny1 * Y_route[i_smaller]
        rho2 = nx2 * p_x + ny2 * p_y

        # Calculate the intersection of the straight lines using Cramers rule
        # For  Cramers  rule and lienar equations refer to:
        det_1 = rho1 * ny2 - rho2 * ny1
        det_2 = nx1 * ny2 - ny1 * nx2
        det_3 = nx1 * rho2 - nx2 * rho1
        det_4 = nx1 * ny2 - nx2 * ny1

        e_x = det_1 / det_2
        e_y = det_3 / det_4

        # Calculate the value of the lateral distance
        lateral_distance = math.sqrt((e_x - p_x) ** 2 + (e_y - p_y) ** 2)
        condition = (e_x - pos_x) * (p_y - pos_y) - (p_x - pos_x) * (e_y - pos_y)

        # If point P is to the right of the right edge of the road,the sign of the distance amount must be corrected
        if condition < 0:
            lateral_distance *= -1

        # Determine position S along the trajectory
        if i_smaller <= 1:
            pos_S = S_route[i_smaller]
        else:
            pos_S = S_route[i_smaller] + math.sqrt((e_x - X_route[i_smaller]) ** 2 + (e_y - Y_route[i_smaller]) ** 2)
```

```python

        pos_S -= 10 # Correct the calculated route point by the distance l10 = 10m

        if pos_S <= S_route[0]:
            pos_S = S_route[0]
        else:
            pos_S = pos_S

        if pos_S >= S_route[-1]:
            pos_S = S_route[-1]

        sp = sp_smaller_old # Find out the index value for the current position/speed
        while S_route[sp] < pos_S and sp <= len(S_route) - 1:
            sp_smaller = sp
            sp += 1

        sp_smaller_old = sp_smaller # Save value for sp_smaller for the next run

        # Linear interpolation Slope for V_ref and V_max
        m_V_ref_N = Vref_route[sp_smaller + 1] - Vref_route[sp_smaller]
        m_V_max_N = Vmax_route[sp_smaller + 1] - Vmax_route[sp_smaller]
        m_V_max_ref_Z = S_route[sp_smaller + 1] - S_route[sp_smaller]
        m_V_ref = m_V_ref_N / m_V_max_ref_Z
        m_V_max = m_V_max_N / m_V_max_ref_Z

        pos_V_ref = Vref_route[sp_smaller] + m_V_ref * (pos_S - S_route[sp_smaller])
        pos_V_max = Vmax_route[sp_smaller] + m_V_max * (pos_S - S_route[sp_smaller])

    else:
        lateral_distance = 1.5
        pos_S = 0
        pos_V_ref = 100 / 3.6
        pos_V_max = 100 / 3.6


    Out_1.value = lateral_distance
    Out_2.value = pos_S
    Out_3.value = pos_V_ref
    Out_4.value = pos_V_max

    if data_read == 0:
        Out_5.value = 1
    else:
        Out_5.value = 0
```

4 Results

This chapter presents several examples of different paths generated using the developed code files described in Chapter 3, along with the simulation results demonstrating the vehicle's lateral and longitudinal control performance.

4.1 Trajectory generation

The different classes for path generation, speed profile creation, and file saving described in Section 3.1.1 are utilized in the `main.py` file. The main function is responsible for generating vehicle trajectories and storing the resulting data as `.txt` files in the local drive. All initial settings and variables required for trajectory generation are defined in `main.py`, enabling trajectory creation based on the specified configurations.

This structure provides flexibility, allowing the user to modify path parameters solely in `main.py` without altering code in other modules.

4.1.1 Single lane trajectory

Listing 5: Variable settings in main.py file for generating single lane trajectory

```
# path configurations
path_type = ['Str', 'Clo1', 'Cir1', 'Clo2', 'Str', 'Clo3', 'Cir2', 'Clo4', 'Str', 'Clo1', 'Cir1', 'Clo2', 'Str']
length  = [ 500,    20,    300,    20,    200,   20,    80,    20,    200,    100,    160,    100,    500]
arc     = [  0,    240,    240,   240,      0,    70,    70,    70,      0,    120,    120,    120,      0]

number_lanes = 1      # number of lanes to generate
lanewidth = 30        # width of a single lane

# initial conditions
X_start = 0           # starting X-coordinate
Y_start = 15          # starting Y-coordinate
phi_s = 0             # initial heading angle in radians
```

The above settings generate a single-lane trajectory. The variable `path_type` is a list of strings that define the geometry of each path segment. The types of geometries are as follows:

- `'Str'`: Straight line
- `'Clo1'`: Clothoid arc (anticlockwise), transitioning from line to circle
- `'Clo2'`: Clothoid arc (anticlockwise), transitioning from circle to line
- `'Clo3'`: Clothoid arc (clockwise), transitioning from line to circle
- `'Clo4'`: Clothoid arc (clockwise), transitioning from circle to line

- `'Cir1'`: Circular arc with anticlockwise direction
- `'Cir2'`: Circular arc with clockwise direction

The sequence of elements in `path_type` determines the overall geometry of the lane. In this example, the trajectory starts with a straight segment, followed by a clothoid arc connecting to a circular arc in an anticlockwise direction, and continues with alternating path types as specified.

Furthermore, the variable `length` is a list that specifies the length of each path segment, whether it is a straight line or a curved segment, corresponding to the respective indices in the `path_type` list. Similarly, the variable `arc` is a list that defines the radius of curvature for each segment, again based on the indices of `path_type`. As shown in the example above, the radius of curvature for straight segments is set to zero, which is consistent with the nature of straight-line geometry.

The variables `X_start`, `Y_start`, and `phi_s` define the initial position and heading angle of the path. Additionally, the variable `number_lanes` specifies the number of lanes to be generated, while `lanewidth` sets the width of each individual lane.

Figure 4 illustrates the result of a single-lane trajectory generated using the configuration settings described earlier.

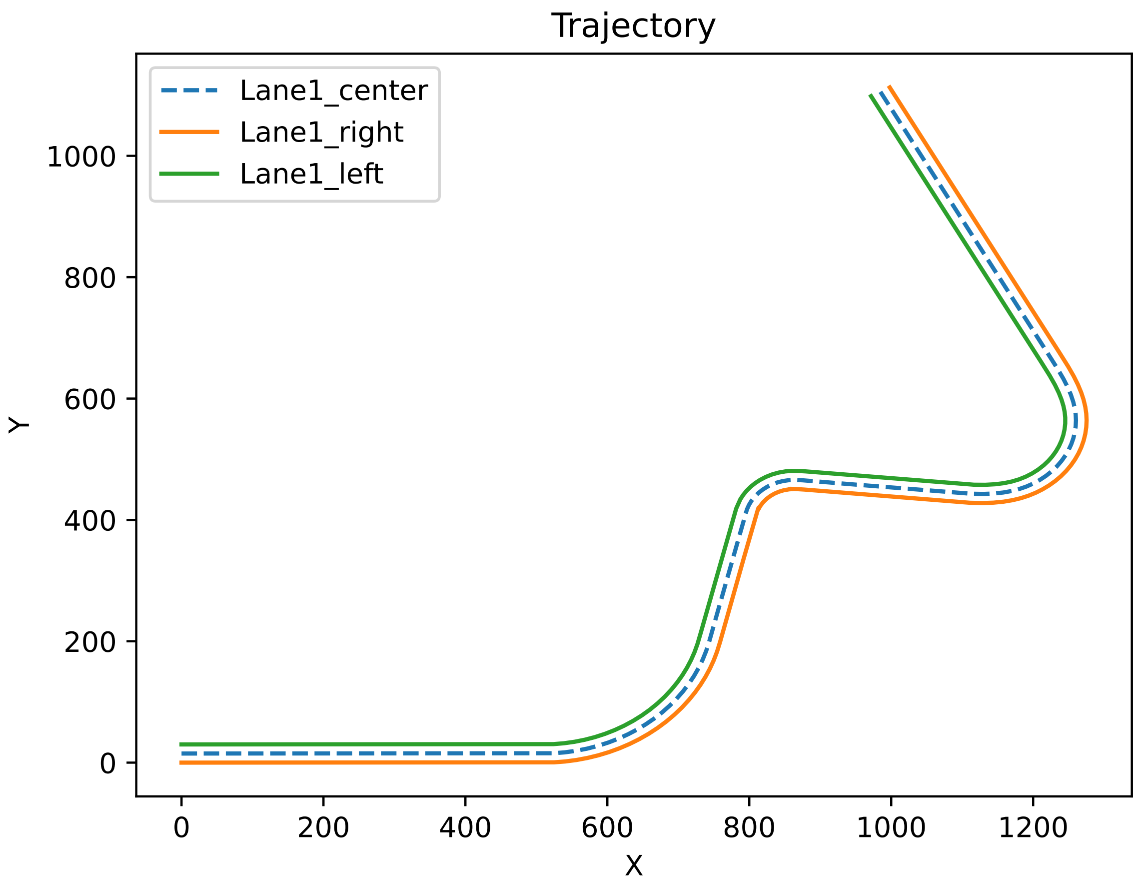

Figure 4: Single lane trajectory

4.1.2 Multi lane trajectory

Figure 5 illustrates a three-lane trajectory generated by setting the `number_lanes` variable to 3, while keeping all other parameter settings unchanged. The resulting path maintains the same geometric shape as shown in Figure 4, with the addition of two parallel lanes.

Listing 6: Variable setting in main.py file for generating multi-lane trajectory

```
number_lanes = 3      # number of lanes to generate
```

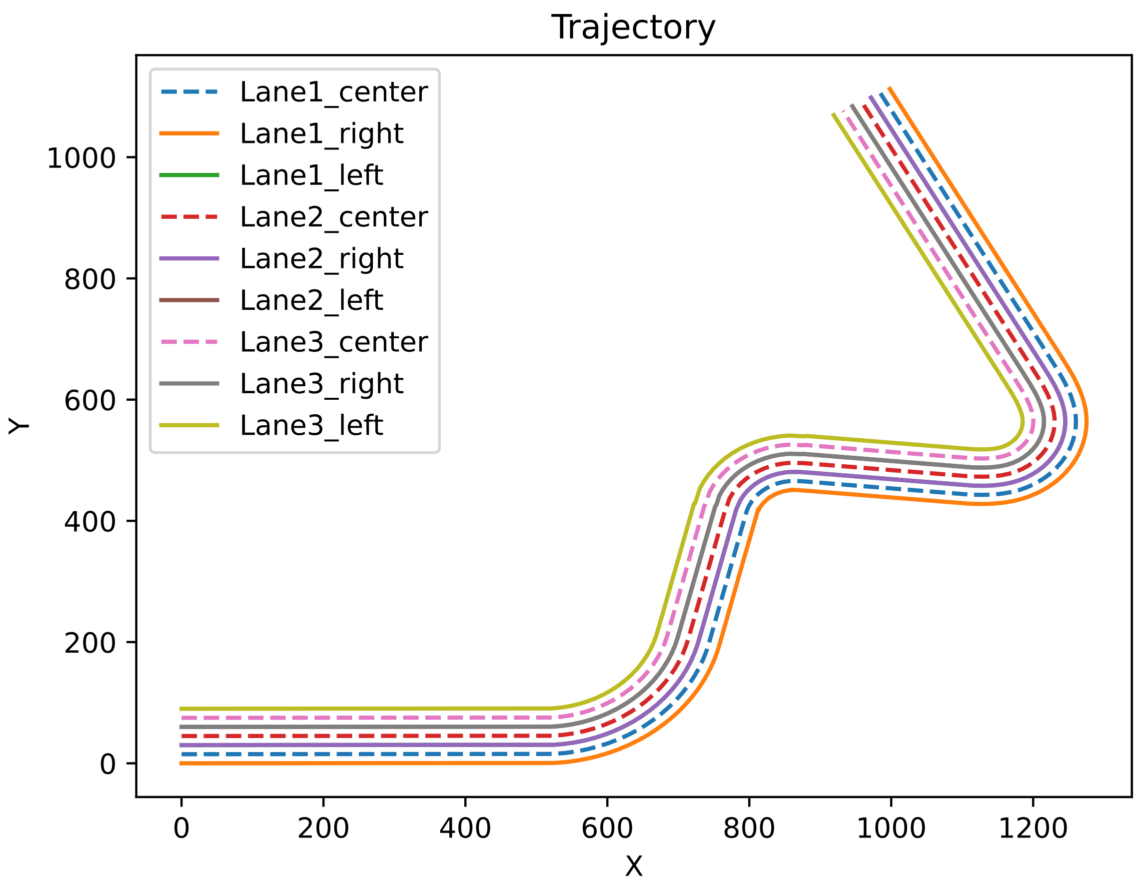

Figure 5: Multi lane trajectory

4.1.3 Multi lane extended trajectory

In this example, the path is extended by appending additional geometric segments to the variables `path_type`, `length`, and `arc`, while keeping the remaining parameters unchanged from the previous example. As illustrated in Figure 6, an extra circular arc followed by a straight segment has been added when compared to the path shown in Example 2 (refer to Figure 5).

This approach highlights the flexibility of the implemented coding structure, which allows for dynamic path generation. Various geometric elements can be easily incorporated or modified, making the framework highly adaptable for diverse path planning scenarios.

Listing 7: Variable settings in main.py file for generating multi-lane extended trajectory

```
# path configurations
path_type = ['Str', 'Clo1', 'Cir1', 'Clo2', 'Str', 'Clo3', 'Cir2', 'Clo4', 'Str'
    , 'Clo1', 'Cir1', 'Clo2', 'Str', 'Clo1', 'Cir1', 'Clo2', 'Str']
length  = [ 500,    20,    300,   20,    200,   20,    80,    20,    200,   100,
        160,   100,   500,  100,   160,   100,   500]
arc     = [  0,    240,    240,  240,      0,   70,    70,    70,      0,   120,
        120,   120,     0,  120,   120,   120,     0]
```

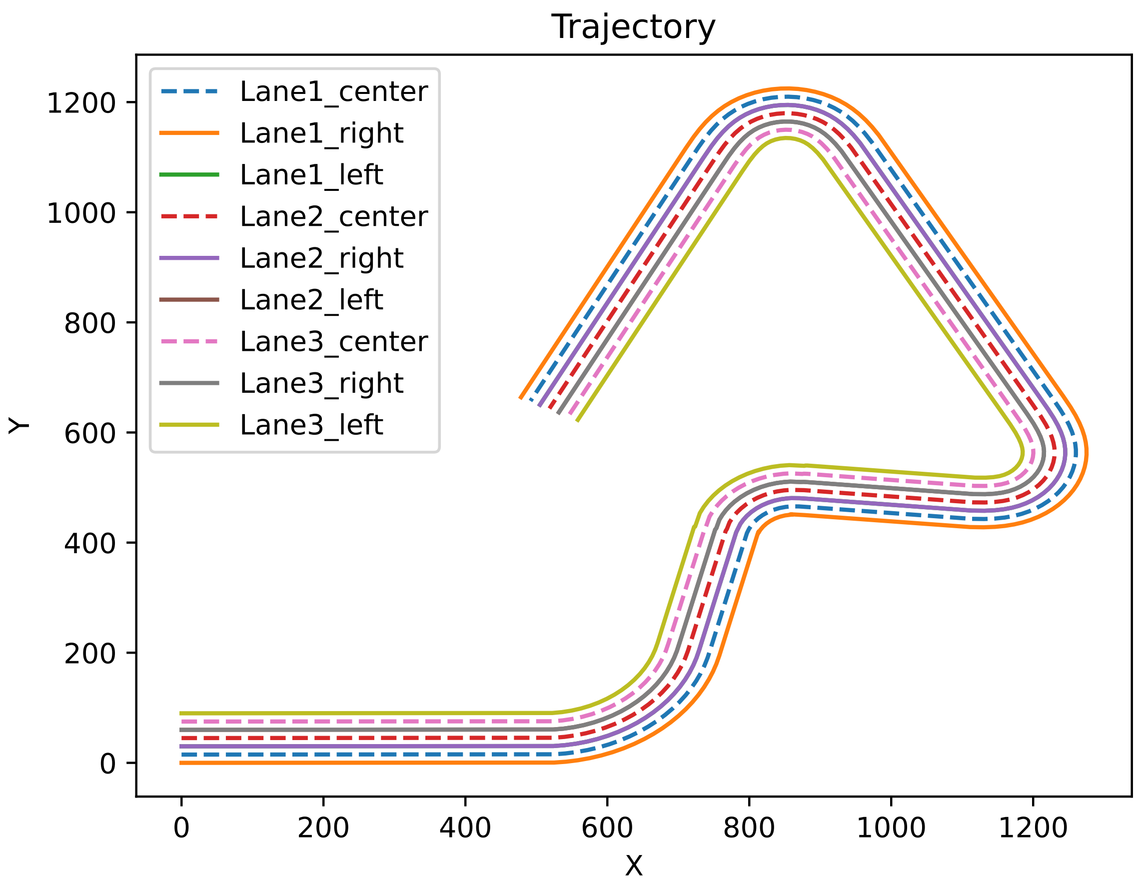

Figure 6: Multi lane extended trajectory

4.2 Lane control simulation

This section presents graphical results of various parameters obtained from the lateral control simulation conducted in the WinFACT software environment, integrated with a Python script for real-time lateral deviation computation. The simulation structure employed is the same as described in the methodology section (see Figure 1).

The path used for the simulation was generated using the Python scripts developed in this work (refer to Chapter 3) and is illustrated in Figure 7. As observed, the vehicle path initially consists of a straight segment, followed by a circular arc with a smooth clothoid transition, and then transitions back to another straight segment.

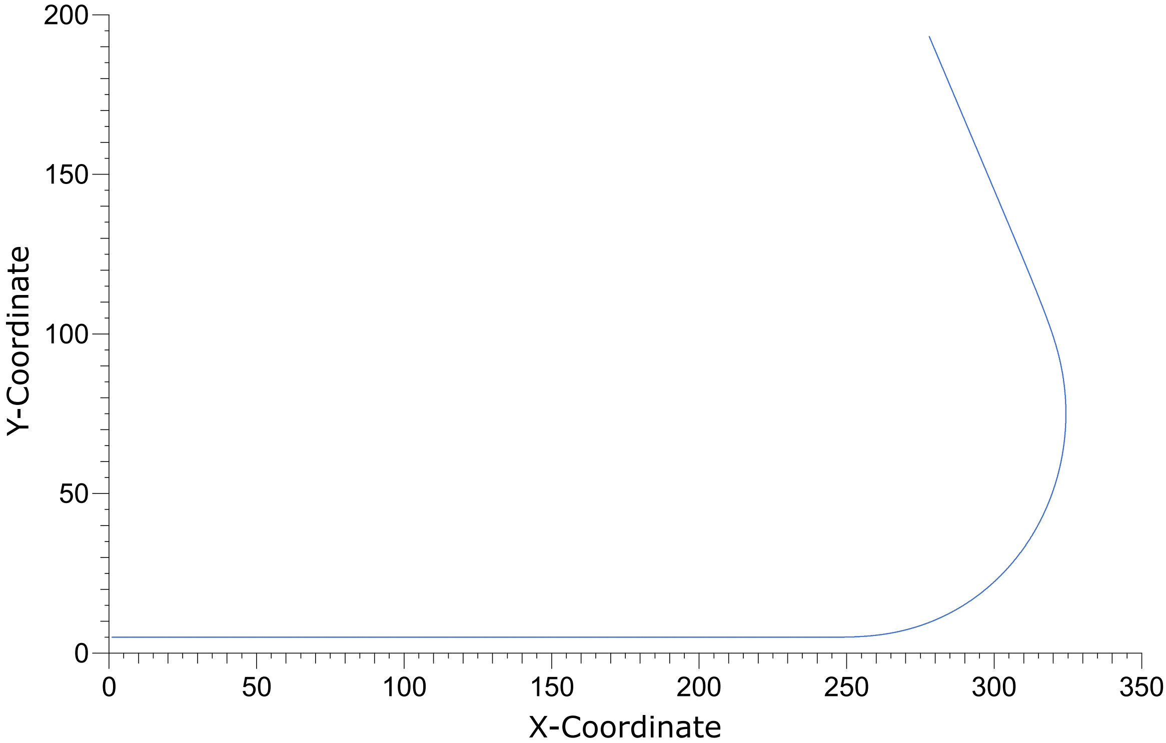

Figure 7: Path followed by vehicle

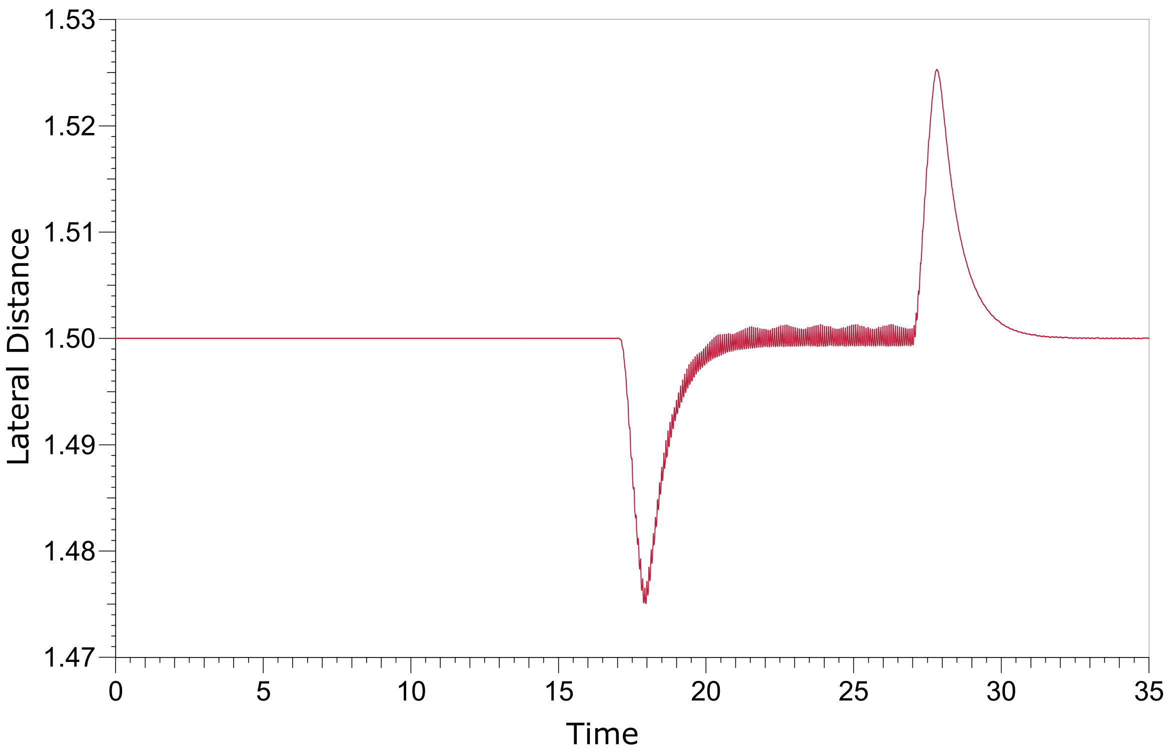

Figure 8: Lateral distance

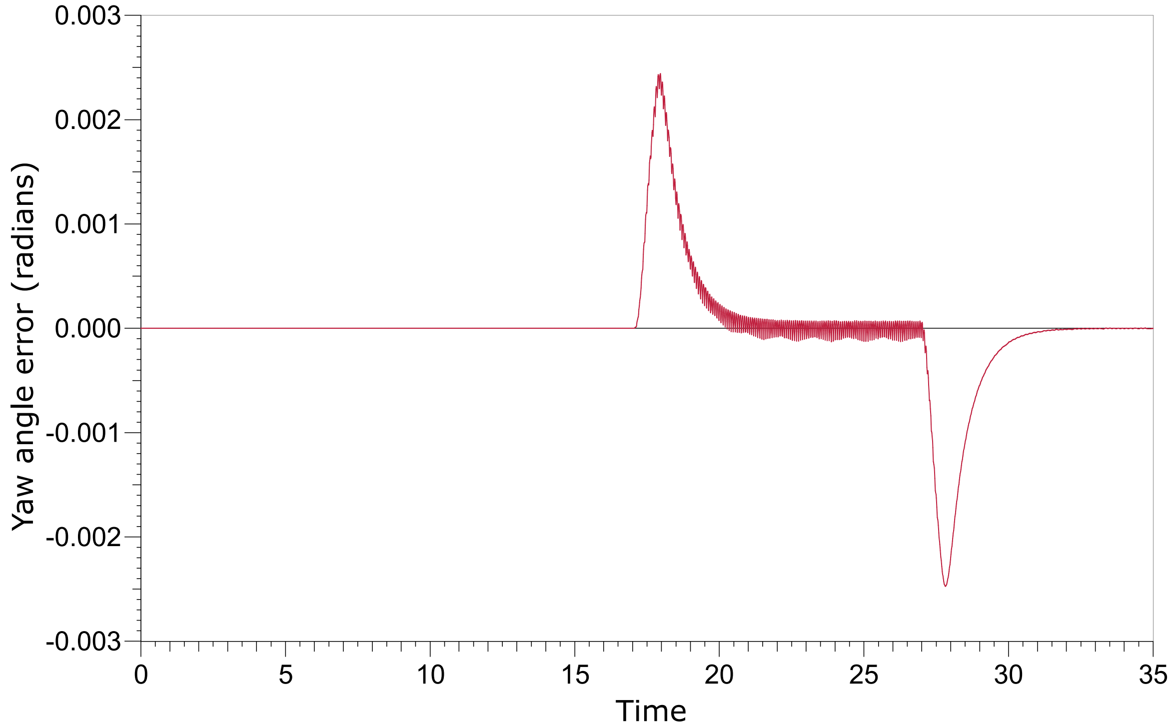

Figure 9: Yaw angle error

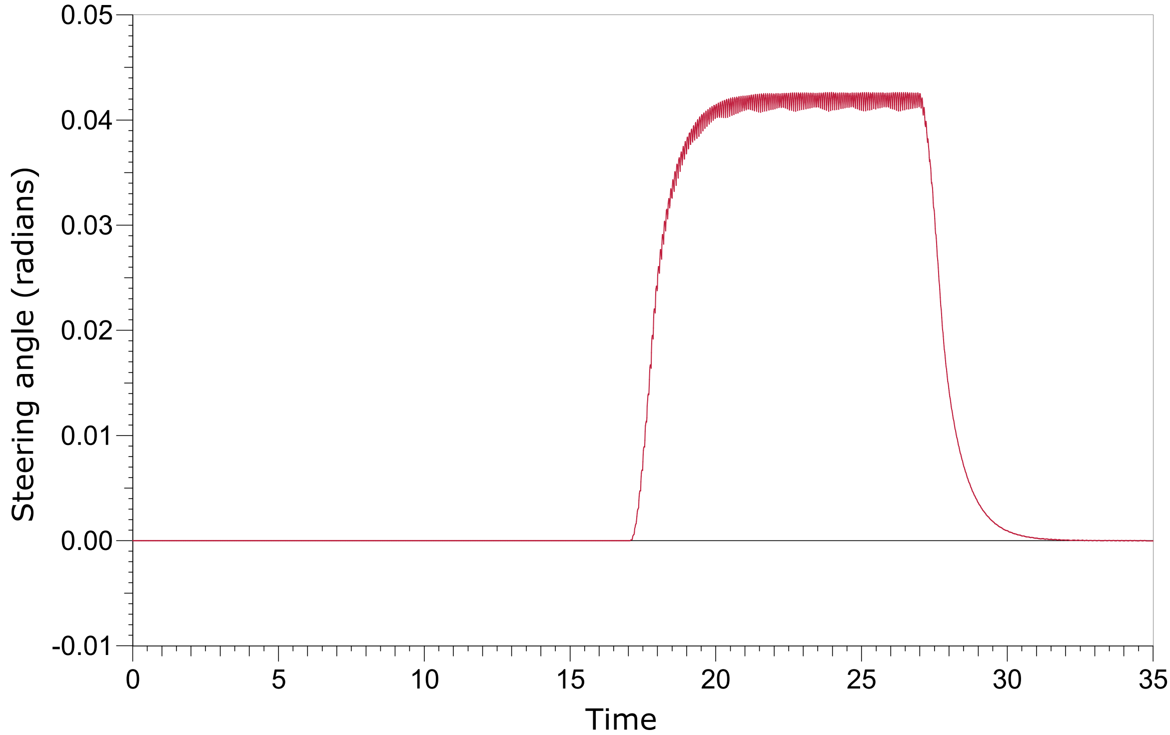

Figure 10: Steering angle

Figure 8 shows the lateral distance computed during the simulation. A single-lane road with a width of 3 meters is used; therefore, the centreline is located 1.5 meters from the right edge of the road. This implies that the desired lateral distance the vehicle must maintain is 1.5 meters relative to the right edge.

As seen in Figure 8, the lateral distance initially remains constant at 1.5 meters while the vehicle follows a straight path. During this phase, both the yaw angle error and the steering angle remain at zero (see Figure 9 and Figure 10).

When the road transitions from a straight to a curved segment, a sudden spike appears in the lateral distance, and a corresponding deviation in the yaw angle error is observed. In response, the controller becomes active and adjusts the steering angle to restore the desired lateral distance. While navigating the curved road, the vehicle exhibits slight deviations in lateral distance. A similar control response occurs when the road transitions back from curved to straight. The steering angle gradually returns to zero on the straight segment, resulting in a steady lateral distance of 1.5 meters once again.

The aggressiveness of the controller is determined by the tuning of the PID constants. From the figures, it is evident that the control actions during road transitions are smooth, and the errors are reduced to zero within a short period, indicating effective control performance.

5 Conclusion and Outlook

This project successfully developed a modular, simulation-based framework for trajectory generation and lane control, designed to support the prototyping and validation of Advanced Driver Assistance Systems (ADAS). Utilizing Python for algorithm development and WinFACT for simulating vehicle dynamics via a single-track model, the framework achieves seamless integration between control logic and dynamic simulation environments. It allows for the creation of customizable trajectory and velocity profiles through configurable geometric elements such as straight lines, clothoids, and circular arcs. These elements are modular and centrally managed, significantly enhancing flexibility and maintainability.

The system supports real-time computation of lateral deviation and vehicle velocity at each simulation step, enabling detailed analysis of control performance and path-following behaviour. A key strength of the framework is the effective synchronization between Python and WinFACT, allowing for interactive communication between trajectory planning and vehicle simulation without manual data exchange. Simulation results confirm the effectiveness of the lane control strategy in maintaining stable lane keeping.

The robustness and adaptability of the framework are evident in its ability to generate and simulate precise single-lane trajectories with effective control response. Its modular architecture provides a solid foundation for future development. Potential enhancements include support for multi-lane trajectory generation, dynamic lane-changing behaviours, and the integration of advanced decision-making and control strategies. Incorporating more complex vehicle models, such as full dynamic representations, would further increase simulation fidelity and applicability to real-world driving scenarios.

Additionally, integration with real-time sensor data or digital twin environments could enable Hardware-in-the-Loop (HiL) testing, expediting the transition from simulation to deployment. The inclusion of machine learning or predictive control techniques may also improve system adaptability in uncertain or rapidly changing environments. Overall, this project establishes a robust, flexible, and scalable platform for accelerated ADAS development, providing a safe environment for testing and extending autonomous vehicle technologies.

References

[1] Jaswanth Nidamanuri; Chinmayi Nibhanupudi; Rolf Assfalg; Hrishikesh Venkataraman: A Progressive Review: Emerging Technologies for ADAS Driven Solutions. In: IEEE: Transactions on Intelligent Vehicles (2022).

[2] Felipe Jiménez; José Eugenio Naranjo; José Javier Anaya: Advanced Driver Assistance System for road environments to improve safety and efficiency. In: Elsevier: Transportation Research Procedia (2022).

[3] M. Kathiresh; R. Neelaveni: Automotive Embedded Systems. 1st ed. Springer, 2021. URL: `https://link.springer.com/content/pdf/10.1007/978-3-030-59897-6`.

[4] Mark de Berg; Marc van kreveld; M.Overmars; O.Schwarzkopf: Computational Geometry. 3rd ed. Springer, 2008. URL: `https://link.springer.com/content/pdf/10.1007/978-3-030-59897-6`.

[5] Nicolas Montes; Marta C. Mora; Josep Tornero: Trajectory Generation based on Rational Bezier Curves as Clothoids. In: IEEE: Intelligent Vehicles Symposium (2007).

[6] Philip Polack;Florent Altché; Brigitte d'Andréa-Novel; Arnaud de La Fortelle: The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles. In: IEEE: Intelligent Vehicles Symposium (2017).

[7] Peter J. Olver; Chehrzad Shakiban: Applied Linear Algebra. 2nd ed. Springer Nature, 2018. URL: `https://link.springer.com/book/10.1007/978-3-319-91041-3`.