# Realistic Intersection Generation: Using Clothoid Spirals

Will Bloomfield

July 2024

## 1 Introduction

We present a method for procedurally generating road intersection layouts, including realistic geometric details and lane configurations. This approach emphasizes the use of Clothoid Spirals in road generation.

## 2 Methods for Intersection Generation

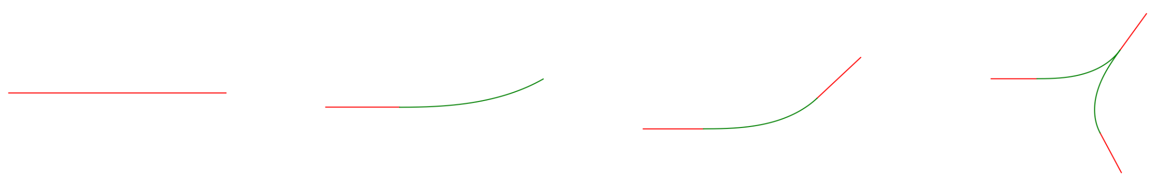
Generating road intersections requires a two-phase process. Phase 1 involves generating incident roads using helper roads created with Clothoid Curves. In Phase 2, we create connection roads based on a set of link configuration rules.

### 2.1 Phase 1: Incident Road Generation

In Phase 1 [4.1], we generate the headings and starting points for all the incident roads. To achieve this, we use helper roads to place the incident roads one by one. Additionally, we determine the number of lanes for each incident road and can insert partitions between the lanes if desired.

An outline of the steps in Phase 1:

1. Create the initial incident road [4.2].
2. Generate a helper road starting from the initial incident road [4.3].
3. Check if the end point of the helper road is in the vicinity of the placed incident roads. If the distance is too close, repeat step 2.
4. If the end point passes the check in step 3, place the incident road at the end of the helper path using the coordinates and heading from the helper path.
5. Repeat steps 2, 3, and 4 for the desired number of incident roads, changing the starting point of the helper road in step 2 to the new incident road.



(a) Step 1 (b) Step 2 (c) Step 3-4 (d) Step 5

Figure 1: Outline of Steps in Phase 1

### 2.1.1 The Helper Path: Clothoid Curve Fitting [4.3]

To construct our helper roads, we will use the simplest form of the clothoid curve, as described below. We only require the initial coordinates, heading, desired initial curvature, and the curvature rate of change.

**Definition 1** (Clothoid Curve)**.**

$$x(0) = x_0, \quad y(0) = y_0, \quad \theta(0) = \theta_0$$

$$\begin{aligned}
\frac{dx}{ds} &= \cos(\theta), \\
\frac{dy}{ds} &= \sin(\theta), \\
\frac{d\theta}{ds} &= \kappa_0 + \kappa_1 s.
\end{aligned}$$

*where $\kappa_0$ is the initial curvature, $\kappa_1$ is the curvature rate of change and, $s$ is the arc length. Given our Initial conditions $(x_0, y_0)$ and the heading $\theta_0$ taken from our Incident Road.*

*Remark* 1. This set of differential equations describes the motion of a particle in a plane where $\theta$ is the angle with the horizontal axis and $\kappa_0$ and $\kappa_1$ are constants related to the curvature of the path.

*Remark* 2. $\theta$ must be in the range $[-\pi, \pi]$

### 2.1.2 Adding Lanes

Given a lane width, one can easily add lanes by keeping the existing curves as the midpoints. Using the heading, we find the two perpendicular angles and then transform the initial point half of the lane width in the new direction. [4.1]
To enhance realism, we allow for random lane widths for each incident road. Since most lanes do not have exactly the same width, our lane widths are generated to simulate this variability.

### 2.1.3 Incident Road Variations

**Multiple lanes**
For multiple lanes, we randomly generate an integer in the range of 1 to 3, which corresponds to the number of lanes on the incident road. By keeping the initial lane central, we first add one lane to the left and then a second lane to the right. The code can be found here. [4.2]

**Partitions**
Similarly, we randomly decide whether an incident road will have partitions. This decision applies to all lanes on the given incident road. The code for this is also found in the Initial Incident Road Generation. [4.2]

(a) Fig 2.a (b) Fig 2.b (c) Fig 2.c (d) Fig 2.d

Figure 2: Incident Roads with multiple lanes and partitions

## 2.2 Phase 2: Generating Connection Roads

With the incident roads placed, we move on to Connections Roads. Each incident road has an initial position $(x_0, y_0)$, angle in $v_1$ and angle out $v_1$. We will use a more generalised clothoid curve to connect the incident roads. We will also confront the lane configuration and the boundary conditions of the junction.

An outline of the steps in Phase 2:

1. For each incident road, calculate the lane configuration for the remaining incident roads [4.4].
2. Calculate the clothoid curve for each lane in the configuration [4.5].
3. Repeat steps 1 and 2 for all incident roads.
4. Create the boundaries for the junction [4.6].

### 2.2.1 Connection Road: Clothoid Fitting [4.5]

**Definition 2** (Generalised Clothoid Curve). *Consider the curve $\gamma : \mathbb{R} \to \mathbb{R}^2$ defined by $\gamma(s) = (x(s), y(s))$, where the functions $x(s)$ and $y(s)$ are given by:*

$$\begin{aligned} x(s) &= x_0 + \int_0^s \cos\left(\frac{1}{2}\kappa_1 t^2 + \kappa_0 t + v_0\right) \mathrm{d}t, \\ y(s) &= y_0 + \int_0^s \sin\left(\frac{1}{2}\kappa_1 t^2 + \kappa_0 t + v_0\right) \mathrm{d}t \end{aligned} \tag{1}$$

*where $s$ is the arc length $s \in \mathbb{R}$, $\kappa_0$ is the initial curvature $\kappa_0 > 0$, $\kappa_1$ is the rate of change of curvature $\kappa_1 \in \mathbb{R}$ and $v_0$ is the initial angle measured from the positive x-axis $v_0 \in [-\pi, \pi]$.*

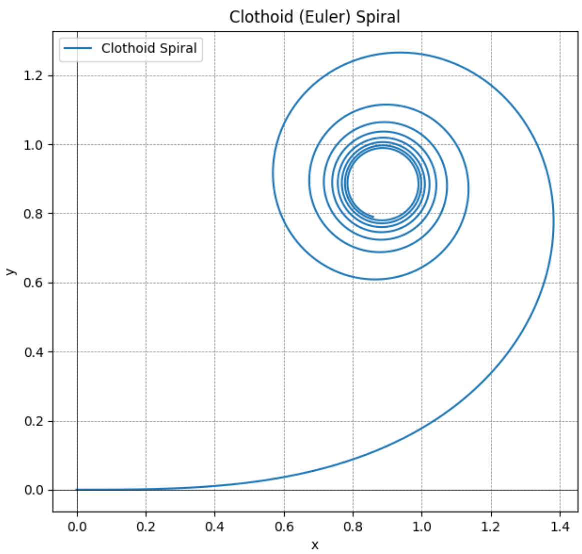



Figure 3: Clothoid (Euler) Spiral for large s.

*Remark* 3.

The Euler Spiral has linearly varying curvature, $\kappa(s) = \kappa_1 s + \kappa_1$. Fix a unit vector $a \in \mathbb{R}^2$, $a = (1, 0)$. Let $t = \dot{\gamma}(s)$ and let $\varphi(s)$ be the angle between $a$ and $t$ measured anticlockwise. Then

$$\kappa(s) = \frac{d\varphi}{ds} \tag{2}$$

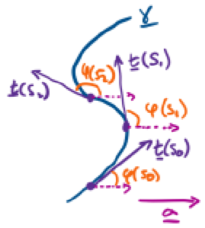


Figure 4: The definition of $\varphi$

*Problem* 1. Given two points $(x_0, y_0)$, $(x_1, y_1)$ and two angles $v_0$ (angle out of $(x_0, y_0)$ ), $v_1$ (angle in for $(x_1, y_1)$), find a clothoid segment of the form (1) which satisfies :

$$\begin{aligned} x(0) = x_0, \quad y(0) = y_0, \quad v(0) = v_0, \\ x(L) = x_1, \quad y(L) = y_1, \quad v(L) = v_1. \end{aligned} \tag{3}$$

for an unknown $L \in \mathbb{R}^+ \setminus \{0\}$

**Solution 1.** We wish to apply the constraints(3) to the curve described by equation (1). First, given the boundary conditions $x(0) = x_0$ and $x(L) = x_1$, we have:

$$x_1 = x_0 + \int_0^L \cos\left(\frac{1}{2}\kappa_1 s^2 + \kappa_0 s + v_0\right) ds \tag{4}$$

Similarly, for the boundary conditions $y(0) = y_0$ and $y(L) = y_1$, we obtain:

$$y_1 = y_0 + \int_0^L \sin\left(\frac{1}{2}\kappa_1 s^2 + \kappa_0 s + v_0\right) ds \tag{5}$$

Finally, considering the constraints $v(0) = v_0$ and $v(L) = v_1$, and using the motivation from remark 3, we have:

$$v_1 = \frac{1}{2}\kappa_1 L^2 + \kappa_0 L + v_0 \tag{6}$$

We now have three unknowns $L$, $\kappa_0$, and $\kappa_1$ that we need to solve for. The solution to problem 1 is the zero of the following nonlinear system of equations.

$$F(L, \kappa_0, \kappa_1) = \begin{pmatrix} x_1 - x_0 - \int_0^L \cos\left(\frac{1}{2}\kappa_1 s^2 + \kappa_0 s + v_0\right) ds \\ y_1 - y_0 - \int_0^L \sin\left(\frac{1}{2}\kappa_1 s^2 + \kappa_0 s + v_0\right) ds \\ v_1 - \left(\frac{1}{2}\kappa_1 L^2 + \kappa_0 L + v_0\right) \end{pmatrix} \tag{7}$$

So we need to find the values such that $F(L, \kappa_0, \kappa_1) = 0$. We will now reformulate the system to reduce the number of unknowns, to make solving the system less computationally heavy.

**Reformulation of the Problem**
We can reduce the dimension of the nonlinear system (7) by introducing the parametrization $s = \tau L$, where $\tau \in [0, 1]$.

$$F\left(L, \frac{B}{L}, \frac{2A}{L^2}\right) = \begin{pmatrix} \Delta x - L\int_0^1 \cos(A\tau^2 + B\tau + v_0)\, d\tau \\ \Delta y - L\int_0^1 \sin(A\tau^2 + B\tau + v_0)\, d\tau \\ v_1 - (A + B + v_0) \end{pmatrix} \tag{8}$$

where $A = \frac{1}{2}\kappa_1 L^2$, $B = L\kappa_0$, $\Delta x = x_1 - x_0$, and $\Delta y = y_1 - y_0$.

Notice the third equation in (8) is linear. As we are solving for $F\left(L, \frac{B}{L}, \frac{2A}{L^2}\right) = 0$, consider:

$$\begin{aligned} &\implies v_1 - A - B - v_0 = 0 \\ &\implies B = v_1 - v_0 - A \\ &\implies B = \Delta v - A, \quad \text{where } \Delta v = v_1 - v_0 \end{aligned} \tag{9}$$

Thus, one can choose $B$ so that the third dimension is always zero:

$$B = \Delta v - A, \quad \text{where} \quad \Delta v = v_1 - v_0. \tag{10}$$

This allows the system to be reduced to a system of two equations with two unknowns, $A$ and $L$.

$$G(L, A) = \begin{pmatrix} \Delta x - L\int_0^1 \cos\left(A\tau^2 + (\Delta v - A)\tau + v_0\right) d\tau \\ \Delta y - L\int_0^1 \sin\left(A\tau^2 + (\Delta v - A)\tau + v_0\right) d\tau \end{pmatrix} \tag{11}$$

For further simplification we can use polar coordinates for $\Delta x \in \mathbb{R}, \Delta y \in \mathbb{R}$ and $L > 0$

$$\Delta x = r\cos\phi, \qquad \Delta y = r\sin\phi, \qquad r = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{12}$$

using (11) we define two new nonlinear functions $f(L, A)$ and $g(A)$:

$$f(L,A) = G(L,A) \cdot \begin{pmatrix} \cos\phi \\ \sin\phi \end{pmatrix}, \qquad g(A) = \frac{1}{L} G(L,A) \cdot \begin{pmatrix} \sin\phi \\ -\cos\phi \end{pmatrix}. \tag{13}$$

Using the identity $\sin(\alpha - \beta) = \sin\alpha\cos\beta - \cos\alpha\sin\beta$, we can rewrite $g(A)$:

$$\begin{aligned}
&\text{Let } \theta = A\tau^2 + (\Delta v - A)\tau + v_0. \\
&g(A) = \frac{1}{L} \begin{pmatrix} r\cos\phi - L\int_0^1 \cos\theta \, d\tau \\ r\sin\phi - L\int_0^1 \sin\theta \, d\tau \end{pmatrix} \cdot \begin{pmatrix} \sin\phi \\ -\cos\phi \end{pmatrix} \\
&= \frac{1}{L}\left[ r\cos\phi\sin\phi - L\int_0^1 (\sin\phi\cos\theta)\, d\tau - r\cos\phi\sin\phi + L\int_0^1 (\cos\phi\sin\theta)\, d\tau \right] \\
&= \frac{1}{L}\int_0^1 (\cos\phi\sin\theta - \sin\phi\cos\theta)\, d\tau \\
&= \int_0^1 \sin(\theta - \phi)\, d\tau \\
&= \int_0^1 \left( \sin(A\tau^2 + (\Delta v - A)\tau + v_0 - \phi \right)\, d\tau.
\end{aligned} \tag{14}$$

So now we have rewritten $g(A)$ in the form:

$$g(A) = \int_0^1 \sin(A\tau^2 + (\Delta v - A)\tau + \Delta\phi) d\tau \tag{15}$$

where $\Delta\phi = v_0 - \phi$

Similar one can use the identity $\cos(\alpha - \beta) = cos\alpha\cos\beta + \sin\alpha\sin\beta$ to rewrite $f(L,A)$ :

$$\begin{aligned}
&\text{Let } \theta = A\tau^2 + (\Delta v - A)\tau + v_0. \\
&g(A) = \begin{pmatrix} r\cos\phi - L\int_0^1 \cos\theta \, d\tau \\ r\sin\phi - L\int_0^1 \sin\theta \, d\tau \end{pmatrix} \cdot \begin{pmatrix} \cos\phi \\ \sin\phi \end{pmatrix} \\
&= \left[ r\cos^2\phi - L\int_0^1 (\cos\phi\cos\theta)\, d\tau + r\sin^2\phi - L\int_0^1 (\sin\phi\sin\theta)\, d\tau \right] \\
&= r(\cos^2\phi + \sin^2\phi) - \int_0^1 (\sin\phi\sin\theta + \cos\phi\cos\theta)\, d\tau \\
&= r - \int_0^1 \cos(\theta - \phi)\, d\tau \\
&= r - \int_0^1 \left( \cos(A\tau^2 + (\Delta v - A)\tau + v_0 - \phi \right)\, d\tau.
\end{aligned} \tag{16}$$

So we now rewritten $f(L,A)$ in the form :

$$f(L,A) = r - Lh(A), \tag{17}$$

$$h(A) = \int_0^1 \cos(A\tau^2 + (\Delta v - A)\tau + \Delta\phi)\, d\tau \tag{18}$$

$$\Delta\phi = v_0 - \phi \tag{19}$$

**Lemma 1.**

$$g(A) = 0, \rightarrow h(A) \neq 0. \tag{20}$$

*Proof.* Proof can be found in "fast and accurate Clothoid fitting" ☐

**Lemma 2.** The solutions of the nonlinear system (6) are given by

$$L = \frac{r}{h(A)}, \quad \kappa_0 = \frac{\Delta v - A}{L}, \quad \kappa_1 = \frac{2A}{L^2} \tag{21}$$

where A is the root of $g(A)$ defined in (15) and $h(A)$ is defined in (18)

*Proof.* Let L,A satisfy (13) and (21).Then $f(L,A) = 0$ and hence $G(L,A) = 0$ from lemma 1 when $g(A) = 0$ then $h(A) \neq 0$ and thus L is well defined. ☐

Hence our interpolation problem is reduced to a single equation that can be solved numerically with the Newton Raphson Method.

### 2.2.2 Connection Road: Clothoid Fitting Algorithm [4.5]

**Algorithm 1** Clothoid Curve Generation

**Require:** $x_0, y_0, x_1, y_1, v_0, v_1$

Normalize $v_0, v_1$
Compute $\Delta x, \Delta y, \Delta v$
Compute $\phi = \arctan\left(\frac{\Delta y}{\Delta x}\right)$ , $r$ and $\Delta\phi$
Normalize $\Delta v, \Delta\phi$
Define $g(A)$
Define $dG(A)$
Solve $g(A) = 0$ for $A$ using the Newton-Raphson method
Define $h(A)$
Compute $L, \kappa_0, \kappa_1$
Compute $x\_values$ and $y\_values$ using Definition 2 (1)
**Return** $x\_values, y\_values$

**Algorithm 2** Normalize($\theta$)

while $\theta > +\pi$ do $\theta - 2\pi$
while $\theta < -\pi$ do $\theta + 2\pi$
return $\theta$;

### 2.2.3 Lane configuration 4.4

An intersection does not require every lane to connect. In our model, the lanes have no specific directions, and each incident road has a maximum of 3 lanes. Therefore, we hard coded the logic for which lanes connect to which. Lane configuration can be a complex problem that could warrant a separate paper dedicated to the general issue, so we did not explore it further.

We can see some of our basic lane configuration below:

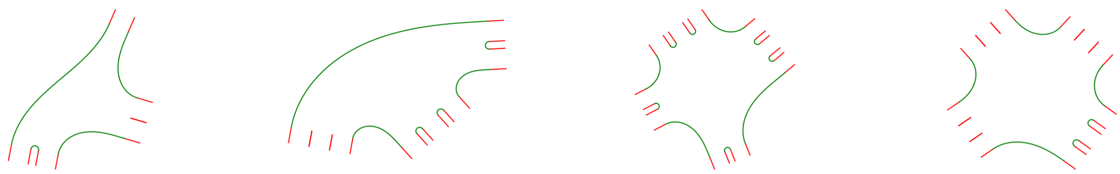
(a) Fig 2.a (b) Fig 2.b (c) Fig 2.c (d) Fig 2.d

Figure 5: Junctions with Connection Roads Showing

### 2.2.4 Boundary Generation [4.6]

We also generate the boundary curves for the junctions. This involves calculating clothoid curves in a clockwise direction, starting from the far-left lane of one incident road and connecting to the far-right lane of the adjacent incident road. This process is repeated to define the complete boundary of the junction.

When partitions are present, we use clothoid curves to create smooth, curved boundaries. Below, you can see examples of junctions where only the boundary curves have been generated.



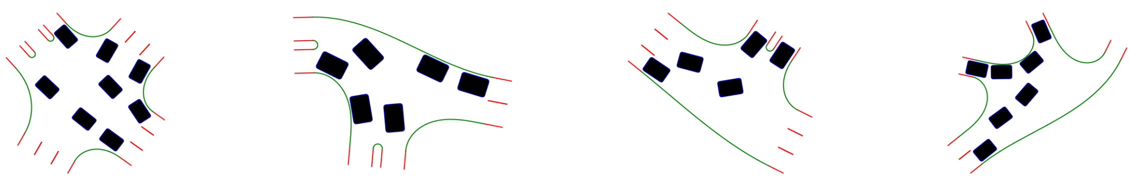
(a) Fig 2.a (b) Fig 2.b (c) Fig 2.c (d) Fig 2.d

Figure 6: Junctions with Boundaries Showing

## 2.3 Adding Static Vehicles

As an additional feature of the project, we introduced static vehicles on the connecting lanes. To ensure realistic traffic behavior in the absence of traffic lights, vehicles are generated only on one selected incident road, limiting their movement to this road. Vehicles are randomly placed along the path while maintaining a minimum distance between them to prevent overlap.

Furthermore, each vehicle's orientation is aligned with the heading of the connecting path, ensuring that the vehicles face the correct direction relative to their position on the road.



(a) Fig 2.a (b) Fig 2.b (c) Fig 2.c (d) Fig 2.d

Figure 7: Junctions with static traffic

# 3 Conclusion and Future Work

This project effectively demonstrates a method for procedurally generating road intersections with a focus on realism and functionality. By employing a two-phase approach, we have successfully integrated realistic geometric details and lane configurations into the intersection design.

In Phase 1, the initial incident roads are placed using helper roads defined by clothoid curves. This allows for random, yet realistic lane placement and configuration, including random lane widths and optional partitions, enhancing the realism of the road network. In Phase 2, we ensure that connection roads align accurately with the incident roads by calculating lane configurations and generating boundary curves. The use of clothoid curves to define junction boundaries provides smooth transitions between lanes, even in the presence of partitions. Additionally, the inclusion of static vehicles on the connecting lanes adds a layer of realism to traffic behavior. By randomly placing vehicles and aligning their orientation with the heading of the connecting paths, the model simulates a more lifelike traffic environment.

Overall, the project addresses the complexities of road intersection design and offers a robust framework for generating realistic and functional road networks. The approach can serve as a foundation for further exploration into more complex traffic simulations and road design challenges.

For future work, developing a more robust lane configuration function would be beneficial, especially for handling scenarios with more than three lanes, such as four lanes. One initial idea is to start with our existing framework for two lanes and then divide them to create four lanes. Additionally, the current vehicle placement function is relatively basic and could be enhanced, particularly by improving the proximity conditions between vehicles. Currently, many configurations are randomized by design. Adding more control over how junctions are generated could also be advantageous for achieving more precise and realistic results.

# References


[1] Miguel E. Vazquez-Mendez and G. Casal. *The clothoid computation: a simple and efficient numerical algorithm*, Journal of Surveying Engineering, 2016.

[2] Enrico Bertolazzi and Marco Frego. *Fast and accurate clothoid fitting*, ResearchGate, 2012.

[3] Brustad, T.F.and Dalmo, R. *Railway Transition Curves: A Review of the State-of-the-Art and Future Research.*, Infrastructures, 2020.

[4] "Euler spiral". *https://en.wikipedia.org/wiki/Euler_spiral*, Wikipedia, 2024.

[5] Golam Md Muktadir, Abdul Jawad, Aleksey Shepelev, Ishaan Parajape and Jim Whithead. *Realistic Road Generation: Intersections*, ResearchGate, 2022.

# 4 Appendices

The code for this project was written in Python. It requires the following packages:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint, quad
from scipy.optimize import minimize
from typing import List, Tuple
import random
```

## 4.1 A.1 Generating Incident Roads

```python
def Incident_Road_Gen(Number_of_incident_Roads: int) -> List:
    Roads = []  # List to store road details
    Helper_paths = []  # List to store helper paths
    Roads.append(Initial_incident_Road())  # Add initial incident road
    Number_of_Incident_Roads = 0

    i = 1  # Start from 1 to avoid index error
    while i < Number_of_incident_Roads:
        temp = []  # Temporary storage for a new incident road
        t = np.linspace(0, 2, 21)  # Time vector

        # Generate helper path based on the last road's end point and angle
        Helper_paths.append(Helper_Path_Gen(Roads[i-1][1][0][0], Roads[i-1][1][1][0], Roads[i-1][-2], Length_of_Roads))

        x0 = Helper_paths[i-1][0][-1]  # x-coordinate of the new road start
        y0 = Helper_paths[i-1][1][-1]  # y-coordinate of the new road start
        Angle_in = normalize_angle(Helper_paths[i-1][2])  # Incoming angle
        Angle_out = normalize_angle(Helper_paths[i-1][2] - np.pi)  # Outgoing angle

        Theta_L = Angle_out + np.pi / 2  # Left angle
        Theta_R = Angle_out - np.pi / 2  # Right angle

        delta_L_x = np.cos(Theta_L)  # x-component for left road
        delta_L_y = np.sin(Theta_L)  # y-component for left road

        delta_R_x = np.cos(Theta_R)  # x-component for right road
        delta_R_y = np.sin(Theta_R)  # y-component for right road

        delta_x = np.cos(Angle_in)  # x-component for the incident road
        delta_y = np.sin(Angle_in)  # y-component for the incident road

        x_coords = list(x0 + t * delta_x)  # x-coordinates of the new road
        y_coords = list(y0 + t * delta_y)  # y-coordinates of the new road

        temp.append([x_coords, y_coords])  # Append new road coordinates

        # Determine partition and lane width
        Partition_of_Inicident_Roads = np.random.uniform(Partition_of_Inicident_Roads_min, Partition_of_Inicident_Roads_max)
        k = random.randint(0, 1)
        if k == 1:
            A = np.random.uniform(lane_width_min, lane_width_max) + Partition_of_Inicident_Roads
            l = True
        else:
            A = np.random.uniform(lane_width_min, lane_width_max)
            l = False

        # Generate additional incident roads if needed
        Number_of_Incident_Roads = random.randint(Min_num_of_Incident_Roads - 1, Max_num_Incident_Roads - 1)
        if Number_of_Incident_Roads == 1:
            temp.append([list((x0 + A * delta_L_x) + t * delta_x), list((y0 + A * delta_L_y) + t * delta_y)])
        elif Number_of_Incident_Roads == 2:
            temp.append([list((x0 + A * delta_L_x) + t * delta_x), list((y0 + A * delta_L_y) + t * delta_y)])
            temp.append([list((x0 + A * delta_R_x) + t * delta_x), list((y0 + A * delta_R_y) + t * delta_y)])
        else:
            temp.append([list((x0) + t * delta_x), list((y0) + t * delta_y)])
```

```
        temp.append(l)  # Add partition flag
        temp.append(A - Partition_of_Inicident_Roads if k == 1 else A)  # Add lane width
        temp.append(Angle_in)  # Add incoming angle
        temp.append(Angle_out)  # Add outgoing angle
        temp.append(Number_of_Incident_Roads + 1)  # Add total number of incident roads

        Roads.append(temp)  # Append new road to the list

        # Check for proximity with existing roads and remove if too close
        should_pop = False
        for j in range(len(Roads) - 1):
            if (np.abs(Roads[-1][1][0][0] - Roads[j][1][0][0]) <=
                Min_seperation_of_Incident_roads and
                np.abs(Roads[-1][1][1][0] - Roads[j][1][1][0]) <=
                    Min_seperation_of_Incident_roads):
                Roads.pop()
                Helper_paths.pop()
                should_pop = True
                break

        if not should_pop:
            i += 1  # Increment index if road is not removed

    return Roads  # Return the list of roads
```

## 4.2 A.2 Initial Incident Road Generator

```
def Initial_incident_Road() -> Tuple[List[List[float]], List[List[float]], float, float, int]:
    # Initialize variables
    Incident_Road_Temp = []
    Number_of_Incident_Roads = 0
    x0 = np.random.uniform(25, 50)  # Random starting x-coordinate
    y0 = np.random.uniform(25, 50)  # Random starting y-coordinate
    t = np.linspace(0, 2, 21)

    Angle_out = np.random.uniform(-np.pi, np.pi)  # Random angle
    Angle_in = normalize_angle(Angle_out + np.pi)  # Adjusted angle

    Theta_L = Angle_out + np.pi / 2  # Left angle
    Theta_R = Angle_out - np.pi / 2  # Right angle

    delta_L_x = np.cos(Theta_L)  # x-direction cosine for left
    delta_L_y = np.sin(Theta_L)  # y-direction sine for left

    delta_R_x = np.cos(Theta_R)  # x-direction cosine for right
    delta_R_y = np.sin(Theta_R)  # y-direction sine for right

    delta_x = np.cos(Angle_in)  # x-direction cosine for incident road
    delta_y = np.sin(Angle_in)  # y-direction sine for incident road

    x_coords = list(x0 + t * delta_x)  # x-coordinates of the road
    y_coords = list(y0 + t * delta_y)  # y-coordinates of the road

    Incident_Road_Temp.append([x_coords, y_coords])  # Append primary road coordinates

    # Determine number of incident roads (1, 2, or 3)
    Number_of_Incident_Roads = random.randint(Min_num_of_Incident_Roads - 1,
        Max_num_Incident_Roads - 1)

    # Partition width
    Partition_of_Inicident_Roads = np.random.uniform(Partition_of_Inicident_Roads_min,
        Partition_of_Inicident_Roads_max)

    # Randomly choose whether to use partition
    k = random.randint(0, 1)

    if k == 1:
        A = np.random.uniform(lane_width_min, lane_width_max) + Partition_of_Inicident_Roads
        l = True
    else:
        A = np.random.uniform(lane_width_min, lane_width_max)
        l = False

    # Append additional incident roads based on the number
```

```python
    if Number_of_Incident_Roads == 1:
        Incident_Road_Temp.append([list((x0 + A * delta_L_x) + t * delta_x), list((y0 + A * delta_L_y) + t * delta_y)])
    elif Number_of_Incident_Roads == 2:
        Incident_Road_Temp.append([list((x0 + A * delta_L_x) + t * delta_x), list((y0 + A * delta_L_y) + t * delta_y)])
        Incident_Road_Temp.append([list((x0 + A * delta_R_x) + t * delta_x), list((y0 + A * delta_R_y) + t * delta_y)])
    else:
        Incident_Road_Temp.append([list((x0) + t * delta_x), list((y0) + t * delta_y)])

    Incident_Road_Temp.append(1)  # Add partition flag

    if k == 1:
        Incident_Road_Temp.append(A - Partition_of_Inicident_Roads)  # Lane width with partition
    else:
        Incident_Road_Temp.append(A)  # Lane width without partition

    Incident_Road_Temp.append(Angle_in)  # Incoming angle
    Incident_Road_Temp.append(Angle_out)  # Outgoing angle
    Incident_Road_Temp.append(Number_of_Incident_Roads + 1)  # Total number of incident roads

    return Incident_Road_Temp
```

## 4.3 A.3 Helper Road Generator

```python
def clothoid_ode_rhs(state, s, kappa0, kappa1):
    x, y, theta = state[0], state[1], state[2]
    # Return derivatives for clothoid curve
    return [np.cos(theta), np.sin(theta), kappa0 + kappa1 * s]

def eval_clothoid(x0, y0, theta0, kappa0, kappa1, s):
    # Solve the ODE for the clothoid curve
    return odeint(clothoid_ode_rhs, [x0, y0, theta0], s, args=(kappa0, kappa1))

def Helper_Path_Gen(x0: float, y0: float, theta0: float, L: float) -> Tuple[List[float], List[float], float]:
    kappa0, kappa1 = Curvature_of_Incident_Road_placment, Curvature_of_Incident_Road_placment
    s = np.linspace(0, L, 1000)  # Generate s values

    sol = eval_clothoid(x0, y0, theta0, kappa0, kappa1, s)  # Compute clothoid path

    xs, ys, thetas = sol[:, 0], sol[:, 1], sol[:, 2]  # Extract coordinates and angles
    return xs, ys, thetas[-1]  # Return path and final angle
```

## 4.4 A.4 Lane configuration

```python
def Connection_road_gen(Incident_roads, Incident_road_index):
    """
    Generates connection roads between a specified incident road and other incident roads.

    Parameters:
    Incident_roads (list): List of roads where each road contains data about its segments and characteristics.
    Incident_road_index (int): Index of the road from which to generate paths.

    Returns:
    list: List of generated connection roads.
    """

    Roads, I = Incident_roads, Incident_road_index

    Connection_Roads = []  # List to store the generated connection roads

    for j in range(len(Incident_roads)):
        if I == j:  # Skip the road if it's the same as the incident road
            break

        # Case where both roads have type 3
        if Roads[I][-1] == 3 and Roads[j][-1] == 3:
            Connection_Roads.append(Clothoid_Curve(Roads[I][1][0][0], Roads[I][1][1][0], Roads[j][2][0][0], Roads[j][2][1][0], Roads[I][-2], Roads[j][-3]))
```

```
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][2][0][0], Roads[I][2][1][0], Roads[j][1][0][0], Roads[j][1][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where incident road is type 3 and other road is type 2
        if Roads[I][-1] == 3 and Roads[j][-1] == 2:
            Connection_Roads.append(Clothoid_Curve(Roads[I][1][0][0], Roads[I][1][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][2][0][0], Roads[I][2][1][0], Roads[j][1][0][0], Roads[j][1][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where incident road is type 3 and other road is type 1
        if Roads[I][-1] == 3 and Roads[j][-1] == 1:
            Connection_Roads.append(Clothoid_Curve(Roads[I][1][0][0], Roads[I][1][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][2][0][0], Roads[I][2][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where incident road is type 2 and other road is type 3
        if Roads[I][-1] == 2 and Roads[j][-1] == 3:
            Connection_Roads.append(Clothoid_Curve(Roads[I][1][0][0], Roads[I][1][1][0], Roads[j][2][0][0], Roads[j][2][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where both roads have type 2
        if Roads[I][-1] == 2 and Roads[j][-1] == 2:
            Connection_Roads.append(Clothoid_Curve(Roads[I][1][0][0], Roads[I][1][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][1][0][0], Roads[j][1][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where incident road is type 2 and other road is type 1
        if Roads[I][-1] == 2 and Roads[j][-1] == 1:
            Connection_Roads.append(Clothoid_Curve(Roads[I][1][0][0], Roads[I][1][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where both roads have type 1
        if Roads[I][-1] == 1 and Roads[j][-1] == 1:
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where incident road is type 1 and other road is type 3
        if Roads[I][-1] == 1 and Roads[j][-1] == 3:
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][2][0][0], Roads[j][2][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][1][0][0], Roads[j][1][1][0], Roads[I][-2], Roads[j][-3]))

        # Case where incident road is type 1 and other road is type 2
        if Roads[I][-1] == 1 and Roads[j][-1] == 2:
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][0][0][0], Roads[j][0][1][0], Roads[I][-2], Roads[j][-3]))
            Connection_Roads.append(Clothoid_Curve(Roads[I][0][0][0], Roads[I][0][1][0], Roads[j][1][0][0], Roads[j][1][1][0], Roads[I][-2], Roads[j][-3]))

    return Connection_Roads  # Return the list of generated connection roads
```

## 4.5 A.5 Clothoid Curve Gen

```
def Clothoid_Curve(a, b, c, d, heading0, heading1):
    x0, y0, x1, y1 = a, b, c, d  # Start and end coordinates
    v0, v1 = heading0, heading1  # Start and end headings
    Delta_x, Delta_y = x1 - x0, y1 - y0  # Difference in coordinates
    Delta_v = normalize_angle(v1 - v0)  # Change in heading
    phi = np.arctan2(Delta_y, Delta_x)  # Angle of the line connecting start and end
```

```python
    r = np.sqrt(Delta_x**2 + Delta_y**2)  # Distance between start and end
    Delta_Phi = normalize_angle(v0 - phi)  # Difference between initial heading and line angle

    # Function to compute the integral for G(A)
    def G(A):
        def integrand(tau):
            return np.sin(A * tau**2 + (Delta_v - A) * tau + Delta_Phi)
        integral = quad(integrand, 0, 1)[0]
        return integral

    # Function to compute the derivative of the integral for G(A)
    def dG(A):
        def integrand(tau):
            return np.cos(A * tau**2 + (Delta_v - A) * tau + Delta_Phi) * (tau**2 - tau)
        integral = quad(integrand, 0, 1)[0]
        return integral

    # Newton's method to find the root of G(A) derivative
    def newton(f, df, x0):
        iterates = [x0]
        for i in range(10):
            iterates.append(iterates[-1] - f(iterates[-1]) / df(iterates[-1]))
        return iterates

    A = newton(G, dG, 1)[-1]  # Find optimal A

    # Function to compute the integral for H(A)
    def H(A):
        def integrand(tau):
            return np.sin(A * tau**2 + (Delta_v - A) * tau + Delta_Phi + (np.pi/2))
        integral = quad(integrand, 0, 1)[0]
        return integral

    L = r / H(A)  # Length of the clothoid
    k0 = (Delta_v - A) / L  # Initial curvature
    k1 = (2 * A) / L**2  # Curvature rate

    # Generate clothoid curve
    def clothoid_curve(x0, y0, theta0, L, kappa0, kappa1, num_points=1000):
        s_vals = np.linspace(0, L, num_points)  # Parameter values
        x_vals = np.zeros(num_points)  # x coordinates
        y_vals = np.zeros(num_points)  # y coordinates
        for i, s in enumerate(s_vals):
            def integrand_x(tau):
                return np.cos(0.5 * kappa1 * tau**2 + kappa0 * tau + theta0)
            def integrand_y(tau):
                return np.sin(0.5 * kappa1 * tau**2 + kappa0 * tau + theta0)
            x_vals[i] = x0 + quad(integrand_x, 0, s)[0]
            y_vals[i] = y0 + quad(integrand_y, 0, s)[0]
        return [x_vals, y_vals]

    return clothoid_curve(x0, y0, v0, L, k0, k1)  # Return the clothoid curve
```

## 4.6 A.6 Boundary Gen

```python
def Boundry_gen(Incident_boundry_Roads, Incident_Roads):
    # Initialize an empty list to store the generated boundary curves.
    Boundry = []

    # Reference to the list of boundary roads provided as input.
    Roads = Incident_boundry_Roads

    # Iterate through each pair of adjacent roads in Incident_Roads.
    for k in range(len(Incident_Roads) - 1):
        # Extract the type of the current and next road segments.
        a = Incident_Roads[k][-1]  # Type of the current road segment (1, 2, or 3)
        b = Incident_Roads[k+1][-1]  # Type of the next road segment (1, 2, or 3)

        # Generate boundary curves based on the types of the current and next road segments.
        if a == 3 and b == 3:
            Boundry.append(Clothoid_Curve(Roads[k][1][0][0][0], Roads[k][1][0][1][0], Roads[k
                +1][2][1][0][0], Roads[k+1][2][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 3 and b == 2:
```

```
            Boundry.append(Clothoid_Curve(Roads[k][1][0][0][0], Roads[k][1][0][1][0], Roads[k+1][0][1][0][0], Roads[k+1][0][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 3 and b == 1:
            Boundry.append(Clothoid_Curve(Roads[k][1][0][0][0], Roads[k][1][0][1][0], Roads[k+1][0][1][0][0], Roads[k+1][0][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 2 and b == 3:
            Boundry.append(Clothoid_Curve(Roads[k][1][0][0][0], Roads[k][1][0][1][0], Roads[k+1][2][1][0][0], Roads[k+1][2][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 2 and b == 2:
            Boundry.append(Clothoid_Curve(Roads[k][1][0][0][0], Roads[k][1][0][1][0], Roads[k+1][0][1][0][0], Roads[k+1][0][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 2 and b == 1:
            Boundry.append(Clothoid_Curve(Roads[k][1][0][0][0], Roads[k][1][0][1][0], Roads[k+1][0][1][0][0], Roads[k+1][0][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 1 and b == 3:
            Boundry.append(Clothoid_Curve(Roads[k][0][0][0][0], Roads[k][0][0][1][0], Roads[k+1][2][1][0][0], Roads[k+1][2][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 1 and b == 2:
            Boundry.append(Clothoid_Curve(Roads[k][0][0][0][0], Roads[k][0][0][1][0], Roads[k+1][0][1][0][0], Roads[k+1][0][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

        if a == 1 and b == 1:
            Boundry.append(Clothoid_Curve(Roads[k][0][0][0][0], Roads[k][0][0][1][0], Roads[k+1][0][1][0][0], Roads[k+1][0][1][1][0], Roads[k][0][-1], Roads[k+1][0][-2]))

    # Handle the boundary between the last road and the first road (cyclic connection).
    i = len(Incident_boundry_Roads) - 1
    j = 0

    # Extract the type of the last and first road segments.
    a = Incident_Roads[i][-1]  # Type of the last road segment
    b = Incident_Roads[j][-1]  # Type of the first road segment

    # Generate boundary curves for the cyclic connection.
    if a == 3 and b == 3:
        Boundry.append(Clothoid_Curve(Roads[i][1][0][0][0], Roads[i][1][0][1][0], Roads[j][2][1][0][0], Roads[j][2][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 3 and b == 2:
        Boundry.append(Clothoid_Curve(Roads[i][1][0][0][0], Roads[i][1][0][1][0], Roads[j][0][1][0][0], Roads[j][0][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 3 and b == 1:
        Boundry.append(Clothoid_Curve(Roads[i][1][0][0][0], Roads[i][1][0][1][0], Roads[j][0][1][0][0], Roads[j][0][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 2 and b == 3:
        Boundry.append(Clothoid_Curve(Roads[i][1][0][0][0], Roads[i][1][0][1][0], Roads[j][2][1][0][0], Roads[j][2][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 2 and b == 2:
        Boundry.append(Clothoid_Curve(Roads[i][1][0][0][0], Roads[i][1][0][1][0], Roads[j][0][1][0][0], Roads[j][0][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 2 and b == 1:
        Boundry.append(Clothoid_Curve(Roads[i][1][0][0][0], Roads[i][1][0][1][0], Roads[j][0][1][0][0], Roads[j][0][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 1 and b == 3:
        Boundry.append(Clothoid_Curve(Roads[i][0][0][0][0], Roads[i][0][0][1][0], Roads[j][2][1][0][0], Roads[j][2][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 1 and b == 2:
        Boundry.append(Clothoid_Curve(Roads[i][0][0][0][0], Roads[i][0][0][1][0], Roads[j][0][1][0][0], Roads[j][0][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))

    if a == 1 and b == 1:
        Boundry.append(Clothoid_Curve(Roads[i][0][0][0][0], Roads[i][0][0][1][0], Roads[j][0][1][0][0], Roads[j][0][1][1][0], Roads[i][0][-1], Roads[j][0][-2]))
```

```python
    # Iterate through each road in Incident_Roads to check for specific conditions and generate additional boundary curves.
    for w in range(len(Incident_Roads)):
        if Incident_Roads[w][-1] == 3 and Incident_Roads[w][-5] == True:
            Boundry.append(Clothoid_Curve(Roads[w][0][0][0][0], Roads[w][0][0][1][0], Roads[w][1][1][0][0], Roads[w][1][1][1][0], Roads[w][0][-1], Roads[w][0][-2]))
            Boundry.append(Clothoid_Curve(Roads[w][2][0][0][0], Roads[w][2][0][1][0], Roads[w][0][1][0][0], Roads[w][0][1][1][0], Roads[w][0][-1], Roads[w][0][-2]))

        if Incident_Roads[w][-1] == 2 and Incident_Roads[w][-5] == True:
            Boundry.append(Clothoid_Curve(Roads[w][0][0][0][0], Roads[w][0][0][1][0], Roads[w][1][1][0][0], Roads[w][1][1][1][0], Roads[w][0][-1], Roads[w][0][-2]))
            Boundry.append(Clothoid_Curve(Roads[w][1][0][0][0], Roads[w][1][0][1][0], Roads[w][0][1][0][0], Roads[w][0][1][1][0], Roads[w][0][-1], Roads[w][0][-2]))

        if Incident_Roads[w][-1] == 1 and Incident_Roads[w][-5] == True:
            Boundry.append(Clothoid_Curve(Roads[w][0][0][0][0], Roads[w][0][0][1][0], Roads[w][1][1][0][0], Roads[w][1][1][1][0], Roads[w][0][-1], Roads[w][0][-2]))
            Boundry.append(Clothoid_Curve(Roads[w][1][0][0][0], Roads[w][1][0][1][0], Roads[w][0][1][0][0], Roads[w][0][1][1][0], Roads[w][0][-1], Roads[w][0][-2]))

    return Boundry
```