Proceedings of Machine Learning Research 1–33

Iterative Best Response of Zero-Sum Racing Game

Clément Suttor CLEMENT.SUTTOR@EPFL.CH

Yo-Shiun Cheng YO-SHIUN.CHENG@EPFL.CH

Abstract

During this project for the course ME-429 MULTIAGENT DECISION-MAKING AND CONTROL, instructed by Dr. Maryam Kamgarpour, we develop a control system in which two autonomous racing cars compete head-to-head. Each car aims to outpace the other by casting their interaction as a zero-sum, multi-stage, dynamic feedback game.

Keywords: Iterative best response, zero-sum game, and feedback game.

1. Introduction

To make this idea concrete, we simplify each vehicle's motion using the *bicycle model*, where a single front wheel and a single rear wheel capture acceleration and steering dynamics. At every instant, each car selects an acceleration and a steering angle; these choices affect its own trajectory and, through the shared track and collision constraints, influence the opponent's possible moves. By planning over a short horizon, solving a quadratic program at each step, and then applying only the first control action before re-planning, both cars continuously adapt to each other—just like real racers glance at each other before deciding whether to overtake or defend.

This formulation is motivated by the need for safe and robust real-time decision making in competitive driving: each autonomous car must anticipate its rival's tactics, respect track and collision constraints, and still pursue its own racing goals. Our main objective is to merge this optimization framework (quadratic programming) with simple game-theoretic ideas so that, at each time step, the resulting pair of control sequences forms a *saddle point*: neither car can unilaterally improve its outcome.

Game-theoretic planning for competitive autonomy has received growing attention in recent years. Wang *et al.* propose a receding-horizon game-theoretic planner for two-car racing scenarios that represents trajectories as piecewise polynomials, incorporates bicycle kinematics, and adds a sensitivity term to account for collision avoidance and opponent yielding; their Sensitivity-Enhanced IBR algorithm was validated in both numerical simulations and full-scale experiments. More recently, Zhang *et al.* analyze zero-sum linear quadratic (LQ) games as nonconvex–nonconcave saddle-point problems and prove that simple policy-optimization schemes (nested gradient, natural policy gradient, etc.) converge globally to Nash equilibria with sublinear and locally linear rates. In contrast, our work bridges these lines by (i) formulating the two-car racing duel as a zero-sum, multi-stage dynamic feedback game in Frenet coordinates with realistic non-holonomic bicycle dynamics, (ii) solving it online via iterative best–response quadratic programs that enforce track-boundary and collision constraints directly, and (iii) demonstrating convergence properties in simulation.

2. Problem Setup

2.1. Notation

- $x = [s,\, d,\, e_\psi,\, v]^\top$: state (longitudinal progress, lateral deviation, heading error, speed).
- $u = [a,\, \delta]^\top$: control input (acceleration, steering angle).
- N: prediction horizon (number of steps).
- Δt: time step.
- $\alpha_{goal}, \alpha_{opt}, \alpha_{next}, \alpha_u$: cost weights.
- $s_{\text{goal}}, d_{\text{opt}_\text{i}}, s_{\text{next}}$: longitudinal progress of the target, offset from the optimal trajectory for player i, longitudinal progress of the next way-point on the optimal trajectory.

2.2. Dynamics and Constraints

2.2.1. VEHICLE DYNAMICS

We model each vehicle with a simple bicycle kinematic model, tracking its position, heading, speed, and steering. These continuous- and discrete-time formulations, the Frenet-coordinate transformation, and their linearization are provided in full in the Appendix A."

2.2.2. CONSTRAINTS

These linear Frenet-frame dynamics are then enforced as constraints in our quadratic programs. At each step k the Frenet-frame state evolves under the (linearized) discrete dynamics

$$x_{k+1} = A_k\, x_k \;+\; B_k\, u_{i,k},$$

subject to:

State and Input Constraints:

$$\begin{aligned} &0 \le v_k \le v_{\max}, \\ &-a_{\max} \le a_{i,k} \le a_{\max}, \\ &-\delta_{\max} \le \delta_{i,k} \le \delta_{\max}, \end{aligned}$$

Track Boundary Constraint:

$$-\frac{\text{track\_width}}{2} \le d_k \le \frac{\text{track\_width}}{2},$$

Collision Avoidance Constraint:

Elliptical safety zone

2.3. Players, Strategy Sets and Cost Functions

We have two players, $i \in \{1, 2\}$:

$$U_i = \{u_{i,0}, u_{i,1}, \dots, u_{i,N-1}\}, \quad u_{i,k} \in \mathcal{U} = [0, a_{\max}] \times [-\delta_{\max}, \delta_{\max}].$$

We split the per-stage cost into four terms:

$$J_{\text{goal}}(s_1, s_2) = \alpha_{\text{goal}}\,(s_{\text{goal}} - s_1)^2 \;-\; \alpha_{\text{goal}}\,(s_{\text{goal}} - s_2)^2, \tag{1}$$

$$J_{\text{opt}}(d_1, d_2) = \alpha_{\text{opt}}\,(d_{\text{opt}_1} - d_1)^2 \;-\; \alpha_{\text{opt}}\,(d_{\text{opt}_2} - d_2)^2, \tag{2}$$

$$J_{\text{next}}(s_1, s_2) = \alpha_{\text{next}}\,(s_{next} - s_1)^2 \;-\; \alpha_{\text{next}}\,(s_{next} - s_2)^2, \tag{3}$$

$$J_u(u_1, u_2) = \alpha_u u_1^2 \;+\; \alpha_u u_2^2, \tag{4}$$

where J_{goal} is the cost w.r.t. the goal, J_{opt} is the cost w.r.t. the optimal trajectory, J_{next} is the cost w.r.t. the next way point on the optimal trajectory, and J_u is the cost on control effort.

The stage cost for Player 1 is then

$$J \;=\; J_{\text{goal}}(s_1, s_2) \;+\; J_{\text{opt}}(d_1, d_2) \;+\; J_{\text{next}}(s_1, s_2) \;+\; J_u(u_1, u_2)$$

and the total cost is

$$\mathcal{J}_1 = \sum_{k=0}^{N-1} J_k \;+\; J_{\text{terminal}}.$$

Player 1 seeks to *minimize* a total cost $\mathcal{J}_1$; Player 2 seeks to *maximize* the same (so $\mathcal{J}_2 = -\mathcal{J}_1$). This makes the game zero-sum.

2.4. Game Classification

- *Zero-sum*: $\mathcal{J}_2 = -\mathcal{J}_1$.
- *Multi-stage*, because decisions are made for $k = 0, \ldots, N-1$.
- *Dynamic*, since state updates couple decisions over time.
- *Feedback*, as both players re-solve at each k based on the current x_k.

3. Analysis

3.1. Iterative Best–Response Algorithm

To compute the saddle-point controls at each time step, we employ an *iterative best–response* scheme. Starting from an initial guess for the opponent's trajectory, each player alternately solves a quadratic program (QP) to optimize its own cost while holding the other fixed. We repeat until neither player can significantly improve. (Details shown in Appendix B.)

3.2. Experimental Setup

All code is written in Python and organized into the following modules:

- `car.py` / `car_dynamics.py`: defines the `Car` class and the RK4-discretized bicycle model dynamics. (Appendix C and D)
- `frenet.py`: routines for computing Frenet coordinates (s, d) relative to a spline–interpolated centerline. (Appendix F)

- `optimal_trajectory.py`: generates a smooth, time-optimal lateral offset $d_{\text{opt}}(s)$ via `scipy.optimize.minimize` (Appendix G)
- `racetrack.py`: builds an "U-shaped" track with the `Track` class and solves for waypoints and headings. (Appendix E)
- `ZS_Controller.py`: implements the zero-sum best-response QPs for both cars using `cvxpy` and `OSQP`. (Appendix H)

3.2.1. SIMULATION WORKFLOW

To put all the things together, this is how we do it:

Track Generation We call

$$\texttt{track} = \texttt{generate\_track(Length=100, Width=6)}$$

to create an U-shaped track. We extract `centerline` $= [x_m, y_m]$ and `headings` $= \theta_m$ from the `Track` object, then build the arc-length vector `centerline_frenet` by cumulative sum of Euclidean distances.

Optimal Trajectories For each car $i \in \{1, 2\}$ we compute

$$d^i_{\text{opt}} = \texttt{generate\_optimal\_traj(centerline\_frenet, track.width, } a^i_{\text{lat\_max}}\texttt{)}$$

which yields a smooth lateral offset that approximately minimizes lap time.

Car Initialization Two `Car` objects are created with different maximum acceleration and maximum velocity to break symmetry:

$$\begin{aligned}
\texttt{car1} &= \texttt{Car}([x_0,\, y_0 + 2,\, \theta_0,\, 5.0],\, v_{\max} = 75,\, a_{\max} = 10,\, a_{\text{lat_max}} = g), \\
\texttt{car2} &= \texttt{Car}([x_0,\, y_0 - 2,\, \theta_0,\, 5.0],\, v_{\max} = 65,\, a_{\max} = 12,\, a_{\text{lat_max}} = 2g),
\end{aligned}$$

where (x_0, y_0, θ_0) is the start of the centerline.

Closed-Loop Simulation We set

$$\Delta t = 0.05 \text{ s}, \quad N = 20, \quad T = \frac{10 \text{ s}}{\Delta t} = 200 \text{ steps},$$

At each step t:

1. Compute Frenet positions s_i, d_i for both cars.
2. Determine next reference s_{next_i} by advancing one waypoint.
3. Call
$$u_1, u_2 = \texttt{zero\_sum\_best\_response}(\ldots, N, \Delta t, \ldots),$$
which returns the first control inputs for each car.
4. Apply `Car.set_control` and `Car.update`.

Plotting After simulation, we plot the track boundaries and the two car histories using `matplotlib.pyplot`.

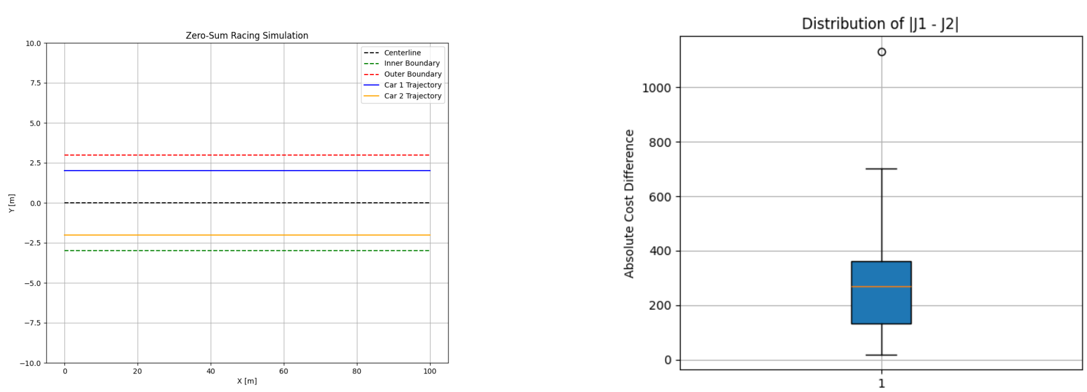
4. Simulation Results

Despite extensive attempts, we were only able to obtain viable results for a **straight-line track**. We experimented with modifying the framework from Cartesian to Frenet coordinates in the Quadratic Program (QP), adjusting the cost function, and relaxing constraints. Although relaxing the constraints enabled the solver to compute a solution, the resulting vehicle behavior was incorrect. Nevertheless, the straight-line case allows us to extract meaningful insights about our implementation.

Figure 1(a) shows the trajectories of the two cars. As expected, both vehicles accelerate forward at maximum speed toward the finish line which is an optimal behavior in this simplified setting. This confirms that the saddle-point controller successfully computed optimal inputs under these conditions.

More interestingly, we can use this scenario to analyze the convergence of our algorithm toward a *saddle-point equilibrium* in the zero-sum dynamic game. According to [2], unconstrained policy optimization converges to equilibrium for a certain class of linear-quadratic dynamic games. We wanted to see if these results applied to our best-response algorithm despite having constrains in the QPs. Figure 1(b) shows a boxplot of the the cost difference $|J_1 - J_2|$ distribution over timesteps of the simulation. The mean and median cost differences are 296.9 and 269.4, respectively. Given that the average total cost per player is on the order of ~ 3000, this implies a relative cost gap $|\bar{V} - \underline{V}|$ of only $\sim 10\%$.

Combined with the fact that the computed control inputs align with the expected optimal behavior, these results suggest that our best-response algorithm approximates well the saddle-point equilibrium at each time step in this scenario.



(a) Car Trajectories on the Track

(b) Distribution of $|J_1 - J_2|$ across timesteps

Figure 1: Simulation Results for the Straight Line Track

5. Conclusion

In this project, we wanted to address the problem of real-time decision-making for autonomous vehicles engaged in competitive racing, formulated as a zero-sum, multi-stage dynamic feedback game. Our goal was to merge game-theoretic planning with optimization techniques to enable two vehicles to anticipate each other's strategies and make rational, competitive driving decisions.

We successfully developed a simulation framework using the iterative best-response approach, applied to cars modeled by bicycle dynamics in Frenet coordinates. The optimal control strategies were derived from quadratic programs that enforced track boundaries and collision avoidance constraints on top of the usual state and input constraints. While the controller struggled to converge on more complex tracks, the straight-line case provided critical validation: both vehicles exhibited optimal behavior and more importantly our algorithm demonstrated convergence toward saddle-point equilibrium solutions.

These results demonstrate that iterative best-response (IBR) methods can effectively approximate saddle-point equilibria in constrained zero-sum dynamic games. Nevertheless, several important questions remain open. Notably, the observed instability of the framework on complex, curved tracks suggests a need for more robust formulations. Improvements may involve refining the constraint handling, enhancing the linearization of vehicle dynamics, or adopting more sophisticated numerical solvers to ensure convergence under tighter feasibility conditions.

Moreover, the modular nature of the proposed framework allows for the investigation and comparison of alternative solutions within the same zero-sum game-theoretic setting. For instance, a natural extension would be to implement a Stackelberg (leader–follower) formulation, where one agent commits to a strategy that the other responds to optimally.

Investigating these alternative strategies may yield valuable insights into robustness, computational efficiency, and applicability across more diverse and realistic racing scenarios.

6. Partnership work and resources

We contributed equally, moving from a grid-based zero-sum racing setup to a continuous-track formulation to tackle collision handling. We briefly split—Clément refining the continuous controller and Yo-Shiun building a first-price auction backup project—then reunited under TA guidance for debugging.

We've leveraged AI throughout our workflow to accelerate debugging and enhance writing: by feeding error messages and solver failure codes into a large language model, we iteratively refined our vehicle dynamics and QP formulations, rapidly identifying linearization mistakes and solver parameter tweaks. Additionally, AI-powered translation tools seamlessly converted Chinese drafts into polished English, ensuring clear documentation among all the project.

Acknowledgments

We'd like to thank our TAs for their invaluable insights and guidance throughout this project, which helped us refine our approach and stay on track.

References

[1] Mingyu Wang et al. "Game Theoretic Planning for Self-Driving Cars in Competitive Scenarios".
[2] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. "Policy optimization provably converges to Nash equilibria in zero-sum linear quadratic games".

Appendix A. Vehicle Dynamics

To describe each car's motion, we use the *bicycle model*, which reduces the four wheels to a single front wheel and a single rear wheel. Let the state vector at time t be

$$x(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \\ v(t) \end{bmatrix},$$

where

- x, y are the car's position coordinates in the plane,
- θ is the heading angle,
- v is the forward speed.

The control inputs are

$$u(t) = \begin{bmatrix} a(t) \\ \delta(t) \end{bmatrix},$$

where a is the longitudinal acceleration and δ is the steering angle of the front wheel.

Continuous-Time Dynamics

The continuous-time equations of motion are

$$\dot{x} = v \cos\theta, \tag{5}$$
$$\dot{y} = v \sin\theta, \tag{6}$$
$$\dot{\theta} = \frac{v}{L} \tan\delta, \tag{7}$$
$$\dot{v} = a, \tag{8}$$

where L is the wheelbase (distance between front and rear wheel centers).

Frenet Coordinate Transformation

To simplify planning along a curved track, we express the vehicle state in *Frenet coordinates* (s, d), where

- s is the longitudinal distance along a reference path $r(s) = \big(x_r(s),\, y_r(s)\big)$,
- d is the signed lateral offset from that path,
- $\theta_r(s)$ is the tangent angle of the path at s,
- $\kappa_r(s)$ is the curvature of the path at s.

Define the heading error

$$e_\psi = \theta - \theta_r(s).$$

Then the continuous-time dynamics in Frenet frame become:

$$\dot{s} = \frac{v\cos(e_\psi)}{1 - \kappa_r(s)\,d}, \tag{9}$$
$$\dot{d} = v\sin(e_\psi), \tag{10}$$
$$\dot{e}_\psi = \frac{v}{L}\tan\delta \;-\; \frac{\kappa_r(s)\,v\cos(e_\psi)}{1 - \kappa_r(s)\,d}, \tag{11}$$
$$\dot{v} = a. \tag{12}$$

Here, the denominator $1-\kappa_r(s)\,d$ accounts for the fact that moving off the centerline effectively shortens (or lengthens) the path curvature. These equations define

$$\dot{x} = f(x, u),$$

with

$$x = \begin{bmatrix} s \\ d \\ e_\psi \\ v \end{bmatrix}, \quad u = \begin{bmatrix} a \\ \delta \end{bmatrix}.$$

Discretization and Linearization in Frenet Frame

We discretize $\dot{x} = f(x, u)$ with RK4 to obtain

$$x_{k+1} = g(x_k, u_k).$$

Then around the current operating point (x_k, u_k) we form the linear approximation

$$x_{k+1} \approx A_k\,x_k \;+\; B_k\,u_k,$$

where

$$A_k = \frac{\partial g}{\partial x}\Big|_{(x_k,u_k)}, \quad B_k = \frac{\partial g}{\partial u}\Big|_{(x_k,u_k)}.$$

Appendix B. Algorithm Description

Algorithm 1 Iterative Best–Response at time step k

Require: current state $x_{f,k}$

1: Initialize

$$U_2^{(0)} \leftarrow \text{zero}, \quad U_1^{(0)} \leftarrow \text{zero}.$$

2: **for** $i = 1, \ldots, i_{\max}$ **do**

3: **Player 1 (minimizer) update:**

$$U_1^{(i)} = \arg\min_{U_1} J_1\big(U_1, U_2^{(i-1)}\big) \quad \text{s.t. dynamics \& constraints}$$

4: Warm-start QP solver with $U_1^{(i-1)}$

5: **Player 2 (maximizer) update:**

$$U_2^{(i)} = \arg\max_{U_2} J_1\big(U_1^{(i)}, U_2\big) = \arg\min_{U_2} -J_1\big(U_1^{(i)}, U_2\big) \quad \text{s.t. dynamics \& constraints}$$

6: Warm-start QP solver with $U_2^{(i-1)}$

7: **if** $\|U_1^{(i)} - U_1^{(i-1)}\|_\infty < \epsilon$ **and** $\|U_2^{(i)} - U_2^{(i-1)}\|_\infty < \epsilon$ **then**

8: **break**

9: **end if**

10: **end for**

11: **Return** $U_1^* = U_1^{(i)},\ U_2^* = U_2^{(i)}$

Details and Parameters

- ϵ is the convergence tolerance (e.g. 10^{-6}).
- $i_{\max}$ is a safeguard on iterations (e.g. 5).
- *Warm-starting* the QP solvers with the previous iterate greatly reduces solve time.
- Once (U_1^*, U_2^*) is found, only the first control inputs $(u_{1,k}^*, u_{2,k}^*)$ are applied; then the process repeats at the next time step $k + 1$.

This procedure ensures that at each discretized time k, the pair (U_1^*, U_2^*) approximates the saddle-point of the finite-horizon game, yielding a feedback policy that continuously reacts to the opponent's moves.

Appendix C. Car Class Code

car.py

```
import numpy as np
from car_dynamics import bicycle_dynamics_discrete
```

```python
from frenet import get_frenet_coords

class Car:
    def __init__(self, initial_state, v_max, a_max, a_lat_max,
        max_steering_angle=np.pi/2, dt=0.1, L=2.5):
        self.state = np.array(initial_state, dtype=float)
        self.control_inputs = np.array([0.0, 0.0])
        self.dt = dt
        self.L = L
        self.history = [self.state.copy()]
        self.v_max = v_max
        self.a_max =  a_max
        self.a_lat_max = a_lat_max
        self.max_steering_angle = max_steering_angle

    def set_control(self, a, delta):
        """Set current control inputs."""
        self.control_inputs = np.array([a, delta])

    def update(self):
        """Advance the state using bicycle dynamics."""
        self.state = bicycle_dynamics_discrete(self.state, self.
            control_inputs, self.dt, self.L)
        self.history.append(self.state.copy())

    def get_state(self):
        """Return the current state."""
        return self.state

    def get_control_inputs(self):
        """Return the current control inputs."""
        return self.control_inputs

    def get_frenet_coords(self, centerline, headings):
        """Return current (s, d) in Frenet frame."""
        x, y = self.state[0], self.state[1]
        return get_frenet_coords(x, y, centerline, headings)

    def compute_delta_psi(self, centerline, headings):
        """Compute delta_psi for frenet dynamics"""
        x, y, theta = self.state[0], self.state[1], self.state[2]
        diffs = centerline - np.array([x, y])
        idx = np.argmin(np.sum(diffs**2, axis=1))
        psi_ref = headings[idx]
        delta_psi = theta - psi_ref
        delta_psi = (delta_psi + np.pi) % (2 * np.pi) - np.pi
        return delta_psi
```

Appendix D. Car Dynamics Code

car_dynamics.py

```python
import numpy as np

def bicycle_dynamics_continuous(state, control, L=2.5):
    """Continuous-time bicycle model dynamics."""
    x, y, theta, v = state
    a, delta = control

    x_dot = v * np.cos(theta)
    y_dot = v * np.sin(theta)
    theta_dot = v / L * np.tan(delta)
    v_dot = a

    return np.array([x_dot, y_dot, theta_dot, v_dot])


def frenet_bicycle_dynamics(x, u, kappa, L=2.5):
    """Continuous-time bicycle model dynamics in Frenet frame."""
    s, d, v, delta_psi = x
    a, delta = u

    denom = max(1.0 - kappa * d, 1e-5)

    s_dot = v * np.cos(delta_psi) / denom
    d_dot = v * np.sin(delta_psi)
    v_dot = a
    delta_psi_dot = v / L * np.tan(delta) - kappa * v * np.cos(delta_psi
        ) / denom

    return np.array([s_dot, d_dot, v_dot, delta_psi_dot])


def rk4_integration(state, control, dt, dynamics):
    """RK4 integration step."""
    k1 = dynamics(state, control)
    k2 = dynamics(state + 0.5 * dt * k1, control)
    k3 = dynamics(state + 0.5 * dt * k2, control)
    k4 = dynamics(state + dt * k3, control)

    return state + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)


def bicycle_dynamics_discrete(state, control, dt, L=2.5):
    """Bicycle model dynamics discretized using RK4"""
    x_plus = rk4_integration(state, control, dt, lambda s, u:
        bicycle_dynamics_continuous(s, u, L))
    return x_plus
```

```python
def frenet_dynamics_discrete(x, u, kappa_func, dt, L=2.5):
    """Frenet bicycle dynamics discretized using RK4"""
    def dynamics(x_local, u_local):
        s = x_local[0]
        kappa = kappa_func(s)
        return frenet_bicycle_dynamics(x_local, u_local, kappa, L)
    x_plus = rk4_integration(x, u, dt, dynamics)
    return x_plus
```

Appendix E. Track Generation Code

racetrack.py

```python
from trackgen import Track
from math import pi
import numpy as np

def generate_track(Length = 1000, Width = 6):
    ''' Generates the track following the example template provided on the github of the trackgen package : https://github.com/mopg/trackgen'''
    # ## Oval
    # # Where are the corners?
    # # crns = np.array( [False,True,False,True], dtype=bool )
    # crns = np.array( [False,True,False], dtype=bool )

    # # Change in angle (needs to be zero for straight)
    # # delTh = np.array( [0,pi,0,pi], dtype=float )
    # delTh = np.array( [0,pi/2,0], dtype=float )

    # # length parameter initial guess (radius for corner, length for straight)
    # # lpar = np.array( [Length/2,Length/5,Length/2,Length/5], dtype=float )
    # lpar = np.array( [Length/2,Length/5,Length/2], dtype=float )

    # # Solve
    # track = Track( length = Length, width = Width, left = True, crns = crns )
    # sol = track.solve( lpar, delTh, case = 2 )

    ## Straight line
    track = Track(
        width =  Width,
        crns = np.array([False]),
        lpar = np.array([Length]),
        delTh = np.array([0.0]),
    )
    track.compTrackXY()

```

```python
    xe, ye, thcum = track.endpoint()

    print( "End point    = (%4.3f, %4.3f)" % (xe, ye) )
    print( "Final angle =  %4.3f" % (thcum) )

    return track
```

Appendix F. Usefull Fuctions in the Frenet Framework

frenet.py

```python
import numpy as np

def get_frenet_coords(x, y, centerline, headings):
    '''Computes the Frenet coordinates (s,d) of a point in cartesian
       coordinates (x,y) w.r.t to the centerline'''
    # Compute closest point on the centerline
    dx = centerline[:,0] - x
    dy = centerline[:,1] - y
    distances = np.hypot(dx,dy)
    idx = np.argmin(distances)

    # Compute arc length s
    s = np.sum(np.hypot(np.diff(centerline[:idx+1, 0]), np.diff(
       centerline[:idx+1, 1])))

    # Compute deviation d
    path_heading = headings[idx]
    normal = np.array([-np.sin(path_heading), np.cos(path_heading)])
    rel_pos = np.array([x, y]) - centerline[idx]
    d = np.dot(rel_pos, normal)

    return s, d

def compute_curvature(centerline):
    '''Computes the curvature Kappa along the centerline'''
    x = centerline[:, 0]
    y = centerline[:, 1]

    dx = np.gradient(x)
    dy = np.gradient(y)
    ddx = np.gradient(dx)
    ddy = np.gradient(dy)

    num = dx * ddy - dy * ddx
    denom_raw = (dx**2 + dy**2) ** 1.5
    # Avoid division by 0
    denom = np.where(np.abs(denom_raw) < 1e-6, 1e-6, denom_raw)
    kappa = np.divide(num, denom)
    return kappa
```

Appendix G. Optimal Trajectory Computation Code

optimal_trajectory.py

```python
import numpy as np
from scipy.optimize import minimize

def lap_time_cost_with_smoothness(d, s, a_lat_max=9.0, w_smooth=0.5):
    """Computes the cost that is used in the optimization problem to compute the optimal lateral offsets from the centerline."""
    # Estimation of the lap time
    ds = np.gradient(s)
    d2_ds2 = np.gradient(np.gradient(d, s), s)
    curvature = np.abs(d2_ds2)
    curvature = np.clip(curvature, 1e-4, None)

    v_max = np.sqrt(a_lat_max / curvature)
    segment_times = ds / v_max
    lap_time = np.sum(segment_times)

    # Penalize fast changes in lateral position to have a smooth optimal trajectory
    smoothness = np.sum(np.diff(d, 2)**2)

    total_cost = lap_time + w_smooth * smoothness

    return total_cost

def generate_optimal_traj(centerline_frenet, track_width, a_lat_max=9.0):
    """Computes the optimal lateral offsets from the centerline to minimize lap time."""
    N = len(centerline_frenet)
    max_dev = track_width / 2 * 0.95
    np.random.seed(42)
    initial_d = np.random.randn(N)*max_dev
    bounds = [(-max_dev, max_dev)] * N # Ensures that the optimal lateral offsets remain inside of the track

    res = minimize(
        lap_time_cost_with_smoothness,
        initial_d,
        args=(centerline_frenet, a_lat_max),
        bounds=bounds,
        method='L-BFGS-B'
    )

    return res.x
```

Appendix H. Best Iterative Response Controller Code

ZS_controller.py

```
import numpy as np
import cvxpy as cp
from car import Car
from car_dynamics import bicycle_dynamics_discrete
from car_dynamics import frenet_dynamics_discrete
from frenet import get_frenet_coords
from scipy.signal import cont2discrete
from scipy.interpolate import interp1d

def linearize_bicycle_dynamics(x_ref, u_ref, L=2.5):
    '''Computes the A and B matrices corresponding to the linearized bicycle dynamics around (x_ref, u_ref)'''
    _, _, theta, v = x_ref
    a, delta = u_ref

    A = np.zeros((4, 4))
    B = np.zeros((4, 2))

    # Compute jacobian w.r.t to state to get A
    A[0, 2] = -v * np.sin(theta)
    A[0, 3] = np.cos(theta)
    A[1, 2] = v * np.cos(theta)
    A[1, 3] = np.sin(theta)
    A[2, 3] = 1.0 / L * np.tan(delta)

    # Compute jacobian w.r.t to input to get B
    B[2, 1] = v / L / (np.cos(delta)**2)
    B[3, 0] = 1.0

    return A, B

def linearize_frenet_dynamics(x_ref, u_ref, kappa, L=2.5):
    '''Computes the A and B matrices corresponding to the linearized bicycle dynamics in frenet coordinates around (x_ref, u_ref, kappa_ref)'''
    s, d, v, delta_psi = x_ref
    a, delta = u_ref

    # Compute constants used in the jacobian
    denom = 1.0 - kappa * d
    denom = np.clip(denom, 1e-6, None)  # prevent division by 0
    cos_dpsi = np.cos(delta_psi)
    sin_dpsi = np.sin(delta_psi)
    cos_dpsi = np.clip(cos_dpsi, -1.0, 1.0)

    A = np.zeros((4, 4))
    B = np.zeros((4, 2))

```

```
    # Compute jacobian w.r.t to state to get A
    A[0, 1] = v * kappa * cos_dpsi / (denom ** 2)
    A[0, 2] = cos_dpsi / denom
    A[0, 3] = -v * sin_dpsi / denom
    A[1, 2] = sin_dpsi
    A[1, 3] = v * cos_dpsi
    A[3, 1] = -v * kappa**2 * cos_dpsi / (denom ** 2)
    A[3, 2] = delta / L - kappa * cos_dpsi / denom
    A[3, 3] = v * kappa * sin_dpsi / denom

    # Compute jacobian w.r.t to input to get B
    B[2, 0] = 1.0
    B[3, 1] = v / (L * (np.cos(delta) ** 2))

    return A, B

def discretize_linear_dynamics(A, B, dt):
    '''Discretize matrices A and B given dt using Zero-Order-Hold'''
    C = np.eye(A.shape[0])
    D = np.zeros(B.shape)
    system = (A, B, C, D)
    A_d, B_d, _, _, _ = cont2discrete(system, dt, method='zoh')
    return A_d, B_d


## Solve the problem in (x,y) Coordinates
def solve_zero_sum_qp_P1(
    car1, car2,
    d_opt1, d_opt2, centerline_frenet,
    centerline, headings, track_width, s_goal, s_next_1, s_next_2,
    P2_control_input,
    N=10, dt=0.1, d_g=1.0, d_u=1.0, d_t=0.2, d_next=2.0, d_c=100.0
):
    n, m = 4, 2
    X1 = cp.Variable((n, N+1))
    U1 = cp.Variable((m, N))
    U2 = P2_control_input
    slack = cp.Variable(N)
    x1_0 = car1.get_state()
    x2_0 = car2.get_state()
    s1_0, d1_0 = car1.get_frenet_coords(centerline, headings)

    constraints = [
        X1[:, 0] == x1_0,
    ]

    cost = 0.0

    # Assume horizon is short enough to linearize around x_0
    A1, B1 = linearize_bicycle_dynamics(x1_0, [0.0, 0.0])
    A1_d, B1_d = discretize_linear_dynamics(A1, B1, dt)
```

```python

    # Create interpolators locally
    d_opt_interp1 = interp1d(centerline_frenet, d_opt1, kind='linear',
        fill_value='extrapolate')
    d_opt_interp2 = interp1d(centerline_frenet, d_opt2, kind='linear',
        fill_value='extrapolate')

    # Compute desired lateral offset
    d_goal1 = d_opt_interp1(s1_0)

    # Compute parameters for frenet coordinates linearization of car 1
    idx1 = np.argmin(np.abs(centerline_frenet - s1_0))
    theta_c1 = headings[idx1]
    t_hat1 = np.array([np.cos(theta_c1), np.sin(theta_c1)])
    n_hat1 = np.array([-np.sin(theta_c1), np.cos(theta_c1)])
    p1_0 = x1_0[0:2]

    # Rotation matrix for whole horizon (assume horizon short enough for
        the cars to keep the same orientation during whole length)
    theta_rel = 0.5 * (x1_0[2] + x2_0[2])  # average heading of the cars
    R = np.array([[np.cos(theta_rel), np.sin(theta_rel)],
                  [-np.sin(theta_rel), np.cos(theta_rel)]])

    # Initialize state variables for car 2
    x2_k = x2_0

    for k in range(N):

        # Constraints on dynamics
        constraints += [X1[:, k+1] == A1_d @ X1[:, k] + B1_d @ U1[:, k]]

        # Box constraints on speed, acceleration, and steering
        constraints += [X1[3, k] <= car1.v_max, X1[3, k] >= 0.0]
        constraints += [U1[0, k] <= car1.a_max, U1[0, k] >= -car1.a_max]
        constraints += [U1[1, k] <= car1.max_steering_angle, U1[1, k] >=
            -car1.max_steering_angle]

        # Collision constraints using an ellipse rotated in the
            direction of the cars (both cars cannot be in the ellipse at
            the same time)
        delta = X1[0:2, k] - x2_k[0:2]
        rel_rot = R @ delta
        collision_expr = cp.quad_form(rel_rot, np.diag([1/2.0**2,
            1/1.0**2]))
        constraints += [collision_expr + slack[k] >= 1.0]
        cost += d_c * slack[k]

        # Slack variables always positive
        constraints += [slack[k] >= 0]

        # Linear approximation of frenet coordinates for X1[0:2,k]
```

```
        s1 = s1_0 + t_hat1 @ (X1[0:2, k] - p1_0)
        d1 = d1_0 + n_hat1 @ (X1[0:2, k] - p1_0)

        # Constraints on lateral deviation to prevent the car from going outside the track
        constraints += [d1 <= track_width/2*0.95, d1 >= -track_width/2*0.95]

        # Use real frenet coordiantes for car 2
        s2, d2 = get_frenet_coords(x2_k[0], x2_k[1], centerline, headings)

        # Update optimal trajectory goal
        d_goal2 = d_opt_interp2(s2)

        # Goal cost
        cost += d_g * cp.square(s_goal - s1) - d_g * cp.square(s_goal - s2)

        # Optimal trajectory cost
        cost += d_t * cp.square(d_goal1 - d1) - d_t * cp.square(d_goal2 - d2)

        # Input cost on acceleration
        cost += d_u * cp.square(U1[0, k]) - d_u * cp.square(U2[0, k])

        # Next waypoint cost
        cost += d_next * cp.square(s_next_1 - s1) - d_next * cp.square(s_next_2 - s2)

        # Update state of car 2
        x2_k = bicycle_dynamics_discrete(x2_k, U2[:,k], dt)

    # Linear approximation of frenet coordinates for  X[0:2,N]
    s1_N = s1_0 + t_hat1 @ (X1[0:2, N] - p1_0)
    d1_N = d1_0 + n_hat1 @ (X1[0:2, N] - p1_0)

    # Real frenet coordinates for X2_N
    s2_N, d2_N = get_frenet_coords(x2_k[0], x2_k[1], centerline, headings)

    # Goal cost
    cost += d_g * cp.square(s_goal - s1_N) - d_g * cp.square(s_goal - s2_N)

    # Optimal trajectory cost
    cost += d_t * cp.square(d_goal1 - d1_N) - d_t * cp.square(d_goal2 - d2_N)

    # Next waypoint cost
```

```
    cost += d_next * cp.square(s_next_1 - s1_N) - d_next * cp.square(
        s_next_2 - s2_N)

    # Problem solving
    prob = cp.Problem(cp.Minimize(cost), constraints)
    prob.solve(solver=cp.OSQP)

    # Check if solver succeeded
    if prob.status not in [cp.OPTIMAL, cp.OPTIMAL_INACCURATE]:
        print("[Warning] Car 1 QP solver failed:", prob.status)
        return np.zeros((2, N))

    return U1.value, prob.value

def solve_zero_sum_qp_P2(
    car1, car2,
    d_opt1, d_opt2, centerline_frenet,
    centerline, headings, track_width, s_goal, s_next_1, s_next_2,
    P1_control_input,
    N=10, dt=0.1, d_g=1.0, d_u=1.0, d_t=0.2, d_next=2.0, d_c=100.0
):
    n, m = 4, 2
    X2 = cp.Variable((n, N+1))
    U2 = cp.Variable((m, N))
    U1 = P1_control_input
    slack = cp.Variable(N)
    x1_0 = car1.get_state()
    x2_0 = car2.get_state()
    s2_0, d2_0 = car2.get_frenet_coords(centerline, headings)

    constraints = [
        X2[:, 0] == x2_0,
    ]

    cost = 0.0

    # Assume horizon is short enough to linearize around x_0
    A2, B2 = linearize_bicycle_dynamics(x2_0, [0.0, 0.0])
    A2_d, B2_d = discretize_linear_dynamics(A2, B2, dt)

    # Create interpolators locally
    d_opt_interp1 = interp1d(centerline_frenet, d_opt1, kind='linear',
        fill_value='extrapolate')
    d_opt_interp2 = interp1d(centerline_frenet, d_opt2, kind='linear',
        fill_value='extrapolate')

    # Compute desired lateral offset
    d_goal2 = d_opt_interp2(s2_0)

    # Compute parameters for frenet coordinates linearization of car 2
    idx2 = np.argmin(np.abs(centerline_frenet - s2_0))
```

```
    theta_c2 = headings[idx2]
    t_hat2 = np.array([np.cos(theta_c2), np.sin(theta_c2)])
    n_hat2 = np.array([-np.sin(theta_c2), np.cos(theta_c2)])
    p2_0 = x2_0[0:2]

    # Rotation matrix for whole horizon (assume horizon short enough for the cars to keep the same orientation during whole length)
    theta_rel = 0.5 * (x1_0[2] + x2_0[2])  # average heading of the cars
    R = np.array([[np.cos(theta_rel), np.sin(theta_rel)],
                  [-np.sin(theta_rel), np.cos(theta_rel)]])

    # Initialize state variables for car 1
    x1_k = x1_0

    for k in range(N):

        # Constraints on dynamics
        constraints += [X2[:, k+1] == A2_d @ X2[:, k] + B2_d @ U2[:, k]]

        # Box constraints on speed, acceleration, and steering
        constraints += [X2[3, k] <= car2.v_max, X2[3, k] >= 0.0]
        constraints += [U2[0, k] <= car2.a_max, U2[0, k] >= -car2.a_max]
        constraints += [U2[1, k] <= car2.max_steering_angle, U2[1, k] >= -car2.max_steering_angle]

        # Collision constraints using an ellipse rotated in the direction of the cars (both cars cannot be in the ellipse at the same time)
        delta = x1_k[0:2] - X2[0:2, k]
        rel_rot = R @ delta
        collision_expr = cp.quad_form(rel_rot, np.diag([1/2.0**2, 1/1.0**2]))
        constraints += [collision_expr + slack[k] >= 1.0]
        cost += d_c * slack[k]

        # Slack variables always positive
        constraints += [slack[k] >= 0]

        # Linear approximation of frenet coordinates for X2[0:2,k]
        s2 = s2_0 + t_hat2 @ (X2[0:2, k] - p2_0)
        d2 = d2_0 + n_hat2 @ (X2[0:2, k] - p2_0)


        # Constraints on lateral deviation to prevent the car from going outside the track
        constraints += [d2 <= track_width/2*0.95, d2 >= -track_width/2*0.95]

        # Use real frenet coordiantes for car 2
        s1, d1 = get_frenet_coords(x1_k[0], x1_k[1], centerline, headings)
```

```
        # Update optimal trajectory goal
        d_goal1 = d_opt_interp1(s1)

        # Goal cost
        cost += d_g * cp.square(s_goal - s1) - d_g * cp.square(s_goal -
            s2)

        # Optimal trajectory cost
        cost += d_t * cp.square(d_goal1 - d1) - d_t * cp.square(d_goal2
            - d2)

        # Input cost on acceleration
        cost += d_u * cp.square(U1[0, k]) - d_u * cp.square(U2[0, k])

        # Next waypoint cost
        cost += d_next * cp.square(s_next_1 - s1) - d_next * cp.square(
            s_next_2 - s2)

        # Update state of car 1
        x1_k = bicycle_dynamics_discrete(x1_k, U1[:,k], dt)

    # Linear approximation of frenet coordinates for X2[0:2,N]
    s2_N = s2_0 + t_hat2 @ (X2[0:2, N] - p2_0)
    d2_N = d2_0 + n_hat2 @ (X2[0:2, N] - p2_0)

    # Real frenet coordinates for X2_N
    s1_N, d1_N = get_frenet_coords(x1_k[0], x1_k[1], centerline,
        headings)

    # Goal cost
    cost += d_g * cp.square(s_goal - s1_N) - d_g * cp.square(s_goal -
        s2_N)

    # Optimal trajectory cost
    cost += d_t * cp.square(d_goal1 - d1_N) - d_t * cp.square(d_goal2 -
        d2_N)

    # Next waypoint cost
    cost += d_next * cp.square(s_next_1 - s1_N) - d_next * cp.square(
        s_next_2 - s2_N)

    # Solve the problem
    prob = cp.Problem(cp.Minimize(-cost), constraints)
    prob.solve(solver=cp.OSQP)

    # Check if solver succeeded
    if prob.status not in [cp.OPTIMAL, cp.OPTIMAL_INACCURATE]:
        print("[Warning] Car 2 QP solver failed:", prob.status)
        return np.zeros((2, N))

```

```python
    return U2.value, prob.value

def zero_sum_best_response(
    car1, car2,
    d_opt1, d_opt2, centerline_frenet,
    centerline, headings, track_width, s_goal, s_next_1, s_next_2,
    N=10, dt=0.1, d_g=1.0, d_u=1.0, d_t=20.0, d_next=20.0, d_c=100.0,
    max_iters=5, epsilon=1e-6
):
    np.random.seed(42)
    m = 2
    U1 = np.zeros((m, N))
    U2 = np.zeros((m, N))

    for i in range(max_iters):
        print(f"  Iter {i+1}/{max_iters}")

        # Minimize cost for Car 1 given U2
        U1_new, cost_J1 = solve_zero_sum_qp_P1(
            car1, car2,
            d_opt1, d_opt2, centerline_frenet,
            centerline, headings, track_width, s_goal, s_next_1,
                s_next_2,
            U2, N, dt, d_g, d_u, d_t, d_next, d_c
        )

        # Maximize cost for Car 2 given U1
        U2_new, cost_J2 = solve_zero_sum_qp_P2(
            car1, car2,
            d_opt1, d_opt2, centerline_frenet,
            centerline, headings, track_width, s_goal, s_next_1,
                s_next_2,
            U1_new, N, dt, d_g, d_u, d_t, d_next, d_c
        )

        # Check for convergence
        delta_U1 = np.linalg.norm(U1_new - U1)
        delta_U2 = np.linalg.norm(U2_new - U2)
        if delta_U1 < epsilon and delta_U2 < epsilon:
            break

        U1, U2 = U1_new, U2_new

    # Print cost value to chack if its a saddle_point equilibrium
    print("Cost for player 1: ", cost_J1)
    print("Cost for player 2: ", cost_J2)

    return U1[:, 0], U2[:, 0]

## Solve the problem in Frenet Coordinates
def solve_zero_sum_qp_P1_frenet(
```

```
    car1, car2,
    d_opt1, d_opt2, centerline_frenet, kappa_interp, kappa_ref_1,
    centerline, headings, track_width, s_goal, s_next_1, s_next_2,
    P2_control_input,
    N=10, dt=0.1, d_g=1.0, d_u=1.0, d_t=0.2, d_next=2.0, d_c=100.0
):
    # Init decision variables
    n, m = 4, 2
    X1 = cp.Variable((n, N+1))
    U1 = cp.Variable((m, N))
    U2 = P2_control_input
    slack = cp.Variable(N)

    # Initial state of the cars
    s1_0, d1_0 = car1.get_frenet_coords(centerline, headings)
    s2_0, d2_0 = car2.get_frenet_coords(centerline, headings)
    state1_0 = car1.get_state()
    state2_0 = car2.get_state()
    dpsi1_0 = car1.compute_delta_psi(centerline, headings)
    dpsi2_0 = car2.compute_delta_psi(centerline, headings)
    x1_0 = np.array([s1_0, d1_0, state1_0[2], dpsi1_0])
    x2_0 = np.array([s2_0, d2_0, state2_0[2], dpsi2_0])

    constraints = [
        X1[:, 0] == x1_0,
    ]

    cost = 0.0

    # Assume horizon is short enough to  linearize around x_0
    A1, B1 = linearize_frenet_dynamics(x1_0, [0.0, 0.0], kappa_ref_1)
    A1_d, B1_d = discretize_linear_dynamics(A1, B1, dt)

    # Create interpolators locally
    d_opt_interp1 = interp1d(centerline_frenet, d_opt1, kind='linear',
        fill_value='extrapolate')
    d_opt_interp2 = interp1d(centerline_frenet, d_opt2, kind='linear',
        fill_value='extrapolate')

    # Compute desired lateral offset
    d_goal1 = d_opt_interp1(s1_0)

    # Initialize state variables for car 2
    x2_k = x2_0

    for k in range(N):

        # Constraints on dynamics
        constraints += [X1[:, k+1] == A1_d @ X1[:, k] + B1_d @ U1[:, k]]

        # Frenet coordinates for car 2
```

```
        s2 = x2_k[0]
        d2 = x2_k[1]

        # Box constraints on speed, acceleration, steering and track width
        constraints += [X1[3, k] <= car1.v_max, X1[3, k] >= 0.0]
        constraints += [X1[1, k] <= track_width/2*0.95, X1[1, k] >= -track_width/2*0.95]
        constraints += [U1[0, k] <= car1.a_max, U1[0, k] >= -car1.a_max]
        constraints += [U1[1, k] <= car1.max_steering_angle, U1[1, k] >= -car1.max_steering_angle]

        # Collision constraints
        # rel = cp.vstack([X1[0,k] - s2, X1[1,k] - d2])
        # Q = np.diag([1/2.0**2, 1/1.0**2])
        # collision_expr = cp.quad_form(rel, Q)
        # constraints += [collision_expr + slack[k] >= 1.0]
        # constraints += [slack[k] >= 0]
        # cost += d_c * slack[k]

        # Update optimal trajectory goal
        d_goal2 = d_opt_interp2(s2)

        # Goal cost
        cost += d_g * cp.square(s_goal - X1[0,k]) - d_g * cp.square(s_goal - s2)

        # Optimal trajectory cost
        cost += d_t * cp.square(d_goal1 - X1[1,k]) - d_t * cp.square(d_goal2 - d2)

        # Input cost on acceleration
        cost += d_u * cp.square(U1[0, k]) - d_u * cp.square(U2[0, k])

        # Next waypoint cost
        cost += d_next * cp.square(s_next_1 - X1[0,k]) - d_next * cp.square(s_next_2 - s2)

        # Update state of car 2
        x2_k = frenet_dynamics_discrete(x2_k, U2[:,k], kappa_interp, dt)

    # Frenet coordinates for car 2 at last step
    s2_N = x2_k[0]
    d2_N = x2_k[1]

    # Update optimal trajectory goal at last step
    d_goal2 = d_opt_interp2(s2_N)

    # Goal cost
    cost += d_g * cp.square(s_goal - X1[0,N]) - d_g * cp.square(s_goal - s2_N)
```

```
    # Optimal trajectory cost
    cost += d_t * cp.square(d_goal1 - X1[1,N]) - d_t * cp.square(d_goal2
         - d2_N)

    # Next waypoint cost
    cost += d_next * cp.square(s_next_1 - X1[0,N]) - d_next * cp.square(
        s_next_2 - s2_N)

    # Problem solving
    prob = cp.Problem(cp.Minimize(cost), constraints)
    prob.solve(solver=cp.OSQP)

    # Check if solver succeeded
    if prob.status not in [cp.OPTIMAL, cp.OPTIMAL_INACCURATE]:
        print("[Warning] Car 1 QP solver failed:", prob.status)
        return np.zeros((2, N))

    return U1.value, prob.value

def solve_zero_sum_qp_P2_frenet(
    car1, car2,
    d_opt1, d_opt2, centerline_frenet, kappa_interp, kappa_ref_2,
    centerline, headings, track_width, s_goal, s_next_1, s_next_2,
    P1_control_input,
    N=10, dt=0.1, d_g=1.0, d_u=1.0, d_t=0.2, d_next=2.0, d_c=100.0
):
    # Init decision variables
    n, m = 4, 2
    X2 = cp.Variable((n, N+1))
    U2 = cp.Variable((m, N))
    U1 = P1_control_input
    slack = cp.Variable(N)

    # Initial state of the cars
    s1_0, d1_0 = car1.get_frenet_coords(centerline, headings)
    s2_0, d2_0 = car2.get_frenet_coords(centerline, headings)
    state1_0 = car1.get_state()
    state2_0 = car2.get_state()
    dpsi1_0 = car1.compute_delta_psi(centerline, headings)
    dpsi2_0 = car2.compute_delta_psi(centerline, headings)
    x1_0 = np.array([s1_0, d1_0, state1_0[2], dpsi1_0])
    x2_0 = np.array([s2_0, d2_0, state2_0[2], dpsi2_0])

    constraints = [
        X2[:, 0] == x2_0,
    ]

    cost = 0.0

    # Assume horizon is short enough to  linearize around x_0
```

```
    A2, B2 = linearize_frenet_dynamics(x2_0, [0.0, 0.0], kappa_ref_2)
    A2_d, B2_d = discretize_linear_dynamics(A2, B2, dt)

    # Create interpolators locally
    d_opt_interp1 = interp1d(centerline_frenet, d_opt1, kind='linear',
        fill_value='extrapolate')
    d_opt_interp2 = interp1d(centerline_frenet, d_opt2, kind='linear',
        fill_value='extrapolate')

    # Compute desired lateral offset
    d_goal2 = d_opt_interp1(s2_0)

    # Initialize state variables for car 2
    x1_k = x1_0

    for k in range(N):

        # Constraints on dynamics
        constraints += [X2[:, k+1] == A2_d @ X2[:, k] + B2_d @ U2[:, k]]

        # Frenet coordinates for car 1
        s1 = x1_k[0]
        d1 = x1_k[1]

        # Box constraints on speed, acceleration, steering and track
            width
        constraints += [X2[3, k] <= car2.v_max, X2[3, k] >= 0.0]
        constraints += [X2[1, k] <= track_width/2*0.95, X2[1, k] >= -
            track_width/2*0.95]
        constraints += [U2[0, k] <= car2.a_max, U2[0, k] >= -car2.a_max]
        constraints += [U2[1, k] <= car2.max_steering_angle, U2[1, k] >=
             -car2.max_steering_angle]

        # Collision constraints
        # rel = cp.vstack([X2[0,k] - s1, X2[1,k] - d1])
        # Q = np.diag([1/2.0**2, 1/1.0**2])
        # collision_expr = cp.quad_form(rel, Q)
        # constraints += [collision_expr + slack[k] >= 1.0]
        # constraints += [slack[k] >= 0]
        # cost += d_c * slack[k]

        # Update optimal trajectory goal
        d_goal1 = d_opt_interp2(s1)

        # Goal cost
        cost += d_g * cp.square(s_goal - s1) - d_g * cp.square(s_goal -
            X2[0,k])

        # Optimal trajectory cost
        cost += d_t * cp.square(d_goal1 - d1) - d_t * cp.square(d_goal2
            - X2[1,k])
```

```
        # Input cost on acceleration
        cost += d_u * cp.square(U1[0, k]) - d_u * cp.square(U2[0, k])

        # Next waypoint cost
        cost += d_next * cp.square(s_next_1 - s1) - d_next * cp.square(
            s_next_2 - X2[0,k])

        # Update state of car 2
        x1_k = frenet_dynamics_discrete(x1_k, U1[:,k], kappa_interp, dt)

    # Frenet coordinates for car 2 at last step
    s1_N = x1_k[0]
    d1_N = x1_k[1]

    # Update optimal trajectory goal at last step
    d_goal1 = d_opt_interp2(s1_N)

    # Goal cost
    cost += d_g * cp.square(s_goal - s1_N) - d_g * cp.square(s_goal - X2
        [0,N])

    # Optimal trajectory cost
    cost += d_t * cp.square(d_goal1 - d1_N) - d_t * cp.square(d_goal2 -
        X2[1,N])

    # Next waypoint cost
    cost += d_next * cp.square(s_next_1 - s1_N) - d_next * cp.square(
        s_next_2 - X2[0,N])

    # Problem solving
    prob = cp.Problem(cp.Minimize(-cost), constraints)
    prob.solve(solver=cp.OSQP)

    # Check if solver succeeded
    if prob.status not in [cp.OPTIMAL, cp.OPTIMAL_INACCURATE]:
        print("[Warning] Car 2 QP solver failed:", prob.status)
        return np.zeros((2, N))

    return U2.value, prob.value


def zero_sum_best_response_frenet(
    car1, car2,
    d_opt1, d_opt2, centerline_frenet, kappa_interp, kappa_ref_1,
        kappa_ref_2,
    centerline, headings, track_width, s_goal, s_next_1, s_next_2,
    N=10, dt=0.1, d_g=10.0, d_u=1.0, d_t=30.0, d_next=20.0, d_c=100.0,
    max_iters=5, epsilon=1e-6
):
    np.random.seed(42)
```

```
    m = 2  # dimension of control input
    U1 = np.zeros((m, N))
    U2 = np.random.randn(m, N)
    U2[0, :] *= car2.a_max
    U2[1, :] *= car2.max_steering_angle

    for i in range(max_iters):
        print(f"  Iter {i+1}/{max_iters}")

        # Minimize cost for Car 1 given U2
        U1_new, cost_J1 = solve_zero_sum_qp_P1_frenet(
            car1, car2,
            d_opt1, d_opt2, centerline_frenet, kappa_interp, kappa_ref_1
                ,
            centerline, headings, track_width, s_goal, s_next_1,
                s_next_2,
            U2, N, dt, d_g, d_u, d_t, d_next, d_c
        )

        # Step 2: Maximize cost for Car 2 given U1
        U2_new, cost_J2 = solve_zero_sum_qp_P2_frenet(
            car1, car2,
            d_opt1, d_opt2, centerline_frenet, kappa_interp, kappa_ref_2
                ,
            centerline, headings, track_width, s_goal, s_next_1,
                s_next_2,
            U1_new, N, dt, d_g, d_u, d_t, d_next, d_c
        )

        # Check if solution converged
        delta_U1 = np.linalg.norm(U1_new - U1)
        delta_U2 = np.linalg.norm(U2_new - U2)
        if delta_U1 < epsilon and delta_U2 < epsilon:
            break

        U1, U2 = U1_new, U2_new

    # Print cost value to chack if its a saddle_point equilibrium
    print("Cost for player 1: ", cost_J1)
    print("Cost for player 2: ", -cost_J2)

    return U1[:, 0], U2[:, 0], cost_J1, -1*cost_J2
```

Appendix I. Simulation Code

simulation.py

```
import math
import numpy as np
import matplotlib.pyplot as plt
```

```
from scipy.interpolate import interp1d
from racetrack import generate_track
from optimal_trajectory import generate_optimal_traj
from car import Car
from ZS_Controller import zero_sum_best_response_frenet
from frenet import compute_curvature

def resample_centerline(centerline, num_points=500):
    '''Function used to sample additional points along the centerline
        since trackgen only returns the minimal amount of points
        necessary to describe the track'''
    ds = np.hypot(np.diff(centerline[:, 0]), np.diff(centerline[:, 1]))
    s = np.insert(np.cumsum(ds), 0, 0.0)
    s_uniform = np.linspace(0, s[-1], num_points)
    interp_x = interp1d(s, centerline[:, 0], kind='linear')
    interp_y = interp1d(s, centerline[:, 1], kind='linear')
    centerline_dense = np.stack([interp_x(s_uniform), interp_y(s_uniform
        )], axis=1)
    return centerline_dense, s_uniform

# Generate the track
num_points = 1000
Length, Width = 100, 6
track = generate_track(Length, Width)
track.plot()
centerline_raw = np.column_stack((track.xm, track.ym))

# Remove duplicate (x,y) points from the centerline (trackgen can
    duplicate points sometimes and they can cause numerical errors)
_, unique_indices = np.unique(centerline_raw, axis=0, return_index=True)
centerline_raw = centerline_raw[np.sort(unique_indices)]

# Resample the centerline
centerline, centerline_frenet = resample_centerline(centerline_raw,
    num_points)

# Compute centerline headings
diffs = np.diff(centerline, axis=0)
headings = np.arctan2(diffs[:, 1], diffs[:, 0])
headings = np.append(headings, headings[-1])

# Initialize cars
g = 9.81
car1 = Car([track.xm[0], track.ym[0]+2, headings[0], 5.0], v_max=65.0,
    a_max=10.0, a_lat_max=1*g)
car2 = Car([track.xm[0], track.ym[0]-2, headings[0], 5.0], v_max=65.0,
    a_max=10.0, a_lat_max=2*g)

# Generate the optimal trajectory for each car given the track
d_opt1 = generate_optimal_traj(centerline_frenet, track.width, car1.
    a_lat_max)
```

```
d_opt2 = generate_optimal_traj(centerline_frenet, track.width, car2.a_lat_max)

# Compute the curvature along the centerline (Artificial, only works for the U shaped track)
kappa = np.zeros_like(headings)
straight_angles = [0, np.pi, -np.pi]
tolerance = 1e-2
# Curvature is zero for straight lines (heading is 0 or pi or -pi)
curved_mask = ~np.isclose(headings % (2*np.pi), straight_angles[0], atol=tolerance) & \
              ~np.isclose(headings % (2*np.pi), straight_angles[1], atol=tolerance) & \
              ~np.isclose(headings % (2*np.pi), straight_angles[2], atol=tolerance)

# kappa[curved_mask] = 1.0 / 40.0  # curvature of a semi-circle is simply 1/R

# Curvature interpolation function used in the discrete dynamics
kappa_interp = interp1d(centerline_frenet, kappa, kind='linear', bounds_error=False, fill_value="extrapolate")

# Useful prints for debugging
# print("centerline :", centerline)
# print("headings :", headings)
# print("centerline_frenet :", centerline_frenet)
# print("dopt1 :", d_opt1)
# print("dopt2 :", d_opt2)
# print("curvature:", kappa)

## Simulation
# Simulation parameters init
s_goal = centerline_frenet[-1]
N = 20
dt = 0.05
T = 40

# Cost lists init
J1 = []
J2 = []

# Loop
for t in range(T):
    print(f"Step {t+1}/{T}")

    # Get current Frenet coordinates
    s1, _ = car1.get_frenet_coords(centerline, headings)
    s2, _ = car2.get_frenet_coords(centerline, headings)

    # Find the closest index in the centerline
```

```
    idx_closest_1 = np.argmin(np.abs(centerline_frenet - s1))
    idx_closest_2 = np.argmin(np.abs(centerline_frenet - s2))

    # Move couple steps forward
    idx_next_1 = min(idx_closest_1 + int(math.floor(num_points/10)), len(centerline_frenet) - 1)
    idx_next_2 = min(idx_closest_2 + int(math.floor(num_points/10)), len(centerline_frenet) - 1)

    # Next reference point in s
    s_next_1 = centerline_frenet[idx_next_1]
    s_next_2 = centerline_frenet[idx_next_2]

    # Compute centerline curvature at current car position
    kappa_ref_1 = kappa_interp(s1)
    kappa_ref_2 = kappa_interp(s2)

    # Compute optimal input for both cars using the itterative best response approach
    u1, u2, cost_J1, cost_J2 = zero_sum_best_response_frenet(
        car1, car2,
        d_opt1, d_opt2, centerline_frenet, kappa_interp, kappa_ref_1, kappa_ref_2,
        centerline, headings, Width, s_goal, s_next_1, s_next_2,
        N=N, dt=dt
    )
    print("u1: ", u1)
    print("u2: ", u2)
    car1.set_control(*u1)
    car2.set_control(*u2)
    car1.update()
    car2.update()
    J1.append(cost_J1)
    J2.append(cost_J2)

# Statistics on cost to assess if saddle point equilibrium is reached or not
J1_array = np.array(J1)
J2_array = np.array(J2)

# Compute absolute difference
diff = np.abs(J1_array - J2_array)

# Compute statistics
mean_diff = np.mean(diff)
median_diff = np.median(diff)
std_diff = np.std(diff)
min_diff = np.min(diff)
max_diff = np.max(diff)

# Print statistics
```

```
print("Statistics of |J1 - J2|:")
print(f"Mean    : {mean_diff:.4f}")
print(f"Median : {median_diff:.4f}")
print(f"Std Dev: {std_diff:.4f}")
print(f"Min     : {min_diff:.4f}")
print(f"Max     : {max_diff:.4f}")

# Create box plot
plt.figure(figsize=(6, 5))
plt.boxplot(diff, vert=True, patch_artist=True)
plt.title("Distribution of |J1 - J2|")
plt.ylabel("Absolute Cost Difference")
plt.grid(True)
plt.show()

# Plotting results
h1 = np.array(car1.history)
h2 = np.array(car2.history)

plt.figure(figsize=(10, 8))
plt.plot(track.xm, track.ym, 'k--', label='Centerline')
plt.plot(track.xb1, track.yb1, 'g--', label='Inner Boundary')
plt.plot(track.xb2, track.yb2, 'r--', label='Outer Boundary')
plt.plot(h1[:, 0], h1[:, 1], 'b-', label='Car 1 Trajectory')
plt.plot(h2[:, 0], h2[:, 1], 'orange', label='Car 2 Trajectory')
plt.ylim(-10, 10)
plt.legend()
plt.title("Zero-Sum Racing Simulation")
plt.xlabel("X [m]")
plt.ylabel("Y [m]")
plt.grid(True)
plt.show()
```