MECE652: Autonomous Driving & Navigation
Assignment 1

February 15, 2025

Dikshant
1877098

---

1. For the front-steered mobile robot, the configuration is described by $q = [x, y, \theta, \phi]$, in which the position of the point at the rear axle is denoted by x and y, the front steering angle is shown by ϕ, and the heading angle is θ (i.e., the yaw rate is $\dot{\theta}$). Assume there is rolling without slipping, thus, the sideways velocity of the rear wheels is zero.

    1. Prove that the rolling without slipping constraints lead to the following set of constraints:
    $$\dot{y}\cos\theta - \dot{x}\sin\theta = 0$$
    $$\dot{x}\sin(\theta+\phi) - \dot{y}\cos(\theta+\phi) - \dot{\theta}d\cos\phi = 0$$

    2. Obtain the allowable velocity vectors as a combination of the basis vectors to derive $\dot{q} = g_1(q)u_1 + \cdots + g_m(q)u_m$ where each velocity vector $\dot{q}$ satisfies $w_i(q)\dot{q} = 0$, and elaborate on control inputs and physical representation of the motion for each g_j controlled by $u_1, \cdots, u_m$

Ans: The configuration of a front-steered mobile robot is given by:

$$q = [x, y, \theta, \phi]$$

1. **Deriving the Rolling Constraints:**

    We assume that rolling without slipping occurs, meaning the sideways velocity of the rear wheels is zero. The no-slip condition at the rear wheels imposes the constraint that motion can only occur along the direction of the rear wheel:

    $$\dot{y}\cos\theta - \dot{x}\sin\theta = 0. \tag{1}$$

    The front wheel is steered by an angle ϕ relative to the heading θ. The velocity at the front wheel is obtained by considering the velocity at the rear

and adding the contribution due to rotation. The front wheel's position is at a distance d from the rear axle, so the velocity at the front axle is:

$$x_f = x + dcos\theta$$

$$y_f = y + dsin\theta$$

Taking time derivate, we get:

$$\dot{x_f} = \dot{x} - dsin\theta$$

$$\dot{y_f} = \dot{y} + dcos\theta$$

Since the front wheel moves in the direction of $\theta + \phi$, its lateral velocity component must be zero:

$$\dot{x_f}\sin(\theta+\phi) - \dot{y_f}\cos(\theta+\phi) = 0$$

Substituting, $\dot{x_f}$ and $\dot{y_f}$, we will get:

$$(\dot{x} - dsin\theta)\sin(\theta+\phi) - (\dot{y} + dcos\theta)\cos(\theta+\phi) = 0$$

Rearranging these terms, we will get our final constraint:

$$\dot{x}\sin(\theta+\phi) - \dot{y}\cos(\theta+\phi) - d\dot{\theta}\cos\phi = 0. \tag{2}$$

2. **Finding the Basis for Allowable Velocities:**

   Rewriting the above constraints in matrix form:

$$\begin{bmatrix} -\sin\theta & \cos\theta & 0 & 0 \\ \sin(\theta+\phi) & -\cos(\theta+\phi) & -d\cos\phi & 0 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{3}$$

   These constraints limit the degrees of freedom of the system, allowing only two independent velocity components.

   Since the system has two constraints and four generalized coordinates, there are two independent velocity components. We express the velocity as:

$$\dot{q} = g_1(q)u_1 + g_2(q)u_2, \tag{4}$$

   where $g_1(q)$ and $g_2(q)$ are independent velocity vectors.

The basis vectors must satisfy $A(q)g_i(q) = 0$. Thus, following two basis vector works:

$$g_1(q) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, g_2(q) = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{1}{d}\tan\phi \\ 0 \end{bmatrix}$$

First Basis Vector $g_1(q)$

$$A(q)g_1 = \begin{bmatrix} -\sin\theta \cdot 0 + \cos\theta \cdot 0 + 0 \cdot 0 + 0 \cdot 1 \\ \sin(\phi+\theta) \cdot 0 - \cos(\phi+\theta) \cdot 0 - d\cos\phi \cdot 0 + 0 \cdot 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Second Basis Vector $g_2(q)$

Substituting into constraint (1),

$$\sin\theta \cdot \cos\theta - \cos\theta \cdot \sin\theta + 0 \cdot \frac{1}{d}\tan\phi + 0 \cdot 0 = 0$$

and substituting into constraint (2),

$$\sin(\phi+\theta)\cos\theta - \cos(\phi+\theta)\sin\theta - d\cos\phi \cdot \frac{1}{d}\tan\phi = 0$$

Final Velocity Expression:

The system dynamics can be expressed as:

$$\dot{q} = g_1(q)u_1 + g_2(q)u_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} u_1 + \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{1}{d}\tan\phi \\ 0 \end{bmatrix} u_2$$

where, u_1 represents the steering rate $\dot{\phi}$ and u_1 represents the forward velocity of the robot.

Physical Interpretation

- The first term describes changes in the steering angle, which affects turning.
- The second term in the velocity equation describes motion along the current heading direction.

2. For an autonomous vehicle platform with the geometrical parameters of mass: 1780 kg; front axles to CG: 1.32 m; rear axles to CG: 1.46 m; moment of inertia I_z about

the vertical axis: 4400 kg. m ; front cornering stiffness $C_{\alpha f}$: 70,500 N/rad; rear cornering stiffness $C_{\alpha r}$: 70,500 N/rad on a dry surface.

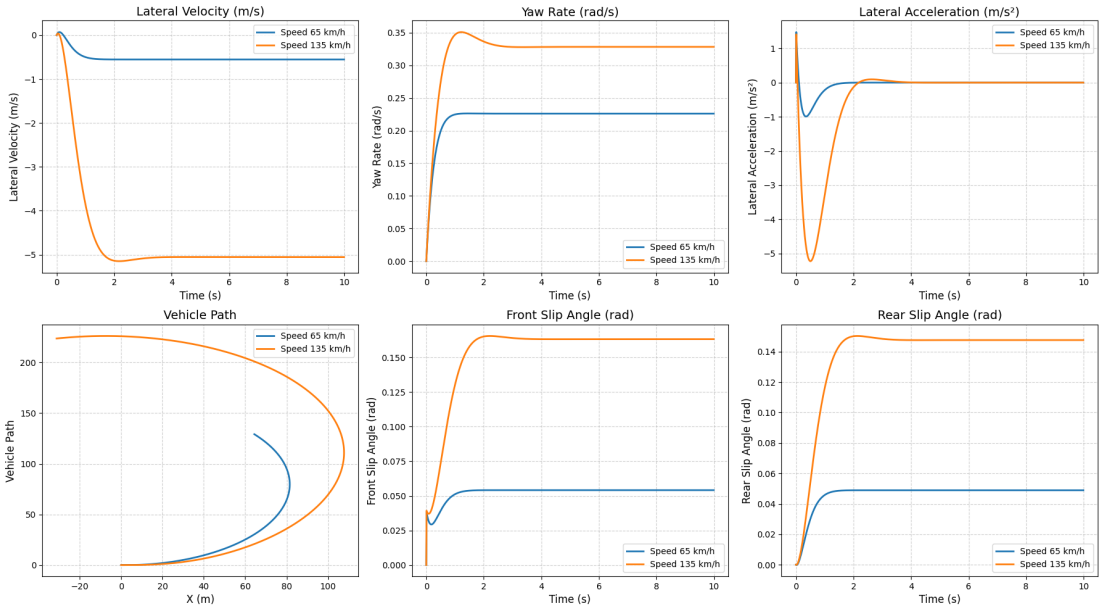
(a) Simulate conventional vehicle handling responses by using a bicycle model for longitudinal speeds of 65 and 135 kp/h (note: you need to change the speed to m/s for the stability and response analysis). For both of these maneuvers, set the front steering input angle to δ 0.04 rad. Compare lateral velocity (in the body frame), yaw rate, lateral acceleration (in the body frame), vehicle path (in the World frame), and tire slip angle responses for the maneuvers (i.e., through plots with time in the horizontal axes).

(b) Discuss on the stability of the system and pole locations, and calculate understeer coefficient. Comments on the results and discuss validity of the bicycle model for the maneuvers considering bicycle model linearity assumptions

(c) Redo all components requested in items i) and ii) above with a bilinear tire force model with the same slope for the linear part (i.e., $C_{\alpha f} = C_{\alpha r}$ = 70,500 N/rad), and saturation after $\alpha_s = 7$ deg for the nominal normal/vertical load.

Ans:

(a) `simulation time = 10s, dt = 0.01`



(b) Poles for speed 65 km/h:

`[-3.91252233+1.4070185j, -3.91252233-1.4070185j]`

Poles for speed 135 km/h:

`[-1.88380705+1.47719347j, -1.88380705-1.47719347j]`

Understeer Coefficient: Under the constant-speed assumption, the stability of the time-invariant system suggests that

$$l + K_{us}^2 > 0$$

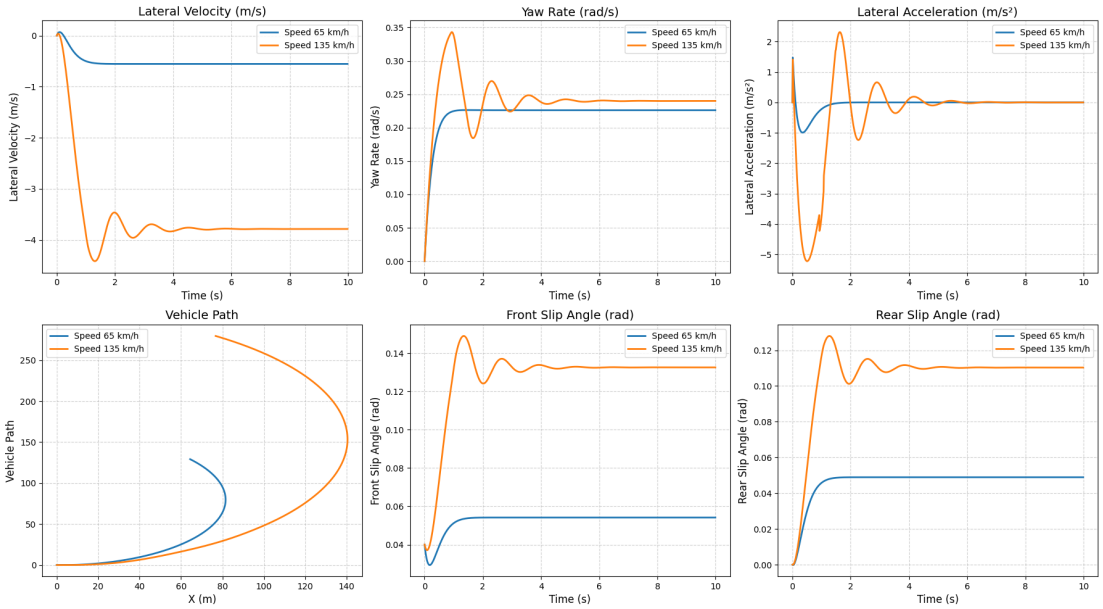
$$K_{us} = (-m(aC_{\alpha f} - bC_{\alpha r}))/lC_{\alpha f}C_{\alpha r}$$

Plugging in the values, we will get: ($K \approx 0.001272$)

The understeer coefficient ($K \approx 0.001272\,\mathrm{rad/m/s^2}$) indicates that the vehicle exhibits **understeer**. The stability of the system is confirmed by the poles of the state matrix, which have negative real parts and are complex conjugates, indicating damped oscillatory behavior in response to steering inputs.

The bicycle model is a useful tool for analyzing steady-state cornering and transient maneuvers under small slip angles and constant speed conditions. However, its linearity assumptions - such as linear tire forces, small slip angles, and neglect of roll/pitch dynamics - limit its accuracy for aggressive maneuvers or high slip angles where non-linear effects like tire saturation and load transfer become significant.

(c) `simulation time = 10s, dt = 0.01`



The understeer coefficient K_{us} does not change with the introduction of a bilinear tire model, but its applicability is limited to the linear region. In the saturated region (large slip angles), the lateral forces F_{yf} and F_{yr} are no longer proportional to the slip angles. The concept of cornering stiffness $(C_{\alpha f}, C_{\alpha r})$ no longer applies because the tire forces are saturated. Thus in this region, the vehicle's handling behavior depends on the maximum lateral forces rather than the cornering stiffness, and the understeer coefficient K_{us} is no longer directly meaningful.

Switching to a bilinear tire model will also change the poles of the system in the saturated region. In the linear region, poles will be same but in the saturated region they will differ. As we can see from the plots, the system is exhibiting reduced stability and a different dynamic behavior compared to the linear tire model. The effect is more evident when the vehicle has a much higher speed (135 km/h). Thus, it's important to consider nonlinear tire behavior for accurate vehicle dynamics analysis, particularly in aggressive maneuvers or high-speed scenarios.

Code

Code can be accessed from Assignment 1 solution and here as well:
2.i

```
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt

class VehicleModel:
    def __init__(self, mass, a, b, I_z, C_af, C_ar, delta, dt=0.01,
        sim_time=10):
        self.m = mass
        self.a = a
        self.b = b
        self.I_z = I_z
        self.C_af = C_af
        self.C_ar = C_ar
        self.delta = delta
        self.dt = dt
        self.t = np.arange(0, sim_time, dt)
```

```
def create_system(self, speed):
    u = speed * (1000 / 3600)
    A = np.array([[-(self.C_af + self.C_ar) / (u * self.m), -((self.a
        * self.C_af - self.b * self.C_ar) / (u * self.m)) - u],
                  [-(self.a * self.C_af - self.b * self.C_ar) / (u *
                      self.I_z), -((self.a ** 2) * self.C_af + (self.b
                      ** 2) * self.C_ar) / (u * self.I_z)]])
    B = np.array([[self.C_af / self.m], [self.a * (self.C_af / self.
        I_z)]])
    C = np.eye(2)
    D = np.zeros((2, 1))
    return signal.StateSpace(A, B, C, D), A

def simulate(self, speed):
    sys, A = self.create_system(speed)
    sys_d = signal.cont2discrete((sys.A, sys.B, sys.C, sys.D), self.dt
        )
    delta_input = np.ones_like(self.t) * self.delta
    _, y, _ = signal.dlsim(sys_d, delta_input)
    return y, A

def compute_trajectory(self, y, speed):
    u = speed * (1000 / 3600)
    theta, X, Y, a_lat, alpha_f, alpha_r = np.zeros((6, len(self.t)))
    y = np.array(y).reshape(len(self.t), -1)

    for n in range(1, len(self.t)):
        theta[n] = y[n, 1] * self.dt + theta[n-1]
        X[n] = (u * np.cos(theta[n]) + y[n, 0] * np.sin(theta[n])) *
            self.dt + X[n-1]
        Y[n] = (u * np.sin(theta[n]) - y[n, 0] * np.cos(theta[n])) *
            self.dt + Y[n-1]
        a_lat[n] = (y[n, 0] - y[n-1, 0]) / self.dt
        alpha_f[n] = self.delta - (y[n, 0] + self.a * y[n, 1]) / u
        alpha_r[n] = (self.b * y[n, 1] - y[n, 0]) / u

    return X, Y, theta, a_lat, alpha_f, alpha_r

def plot_results(self, speeds):
    fig, axs = plt.subplots(2, 3, figsize=(18, 10))
    labels = ["Lateral␣Velocity␣(m/s)", "Yaw␣Rate␣(rad/s)", "Lateral␣
```

```
            Acceleration␣(m/s^2)",
                 "Vehicle␣Path", "Front␣Slip␣Angle␣(rad)", "Rear␣Slip␣
                     Angle␣(rad)"]

        for speed in speeds:
            y, _ = self.simulate(speed)
            X, Y, theta, a_lat, alpha_f, alpha_r = self.compute_trajectory(
                y, speed)
            data = [y[:, 0], y[:, 1], a_lat, (X, Y), alpha_f, alpha_r]

            for j, d in enumerate(data):
                ax = axs[j // 3, j % 3]
                if j == 3:
                    ax.plot(d[0], d[1], label=f"Speed␣{speed}␣km/h",
                        linewidth=2)
                else:
                    ax.plot(self.t, d, label=f"Speed␣{speed}␣km/h",
                        linewidth=2)
                ax.set_title(labels[j], fontsize=14)
                ax.set_xlabel("Time␣(s)" if j != 3 else "X␣(m)", fontsize
                    =12)
                ax.set_ylabel(labels[j], fontsize=12)
                ax.grid(True, linestyle='--', alpha=0.6)
                ax.legend()
                ax.tick_params(axis='both', which='major', labelsize=10)

        plt.tight_layout()
        plt.savefig("q2_results_1.png")
        plt.show()

    def check_stability(self, speeds):
        for speed in speeds:
            _, A = self.simulate(speed)
            poles = np.linalg.eigvals(A)
            print(f"Poles␣for␣speed␣{speed}␣km/h:␣{poles}")
            stability = "Stable" if np.all(np.real(poles) < 0) else "
                Unstable"
            print(f"System␣at␣{speed}␣km/h␣is␣{stability}\n")

if __name__ == "__main__":
    vehicle = VehicleModel(mass=1780, a=1.32, b=1.46, I_z=4400, C_af
        =70500, C_ar=70500, delta=0.04)
```

```
    speeds = [65, 135]
    vehicle.plot_results(speeds)
    vehicle.check_stability(speeds)
```

2.iii Bilinear tire force model

```
import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt

class VehicleModel:
    def __init__(self, mass, a, b, I_z, C_af, C_ar, delta, dt=0.01,
        sim_time=10):
        self.m = mass
        self.a = a
        self.b = b
        self.I_z = I_z
        self.C_af = C_af
        self.C_ar = C_ar
        self.delta = delta
        self.dt = dt
        self.t = np.arange(0, sim_time, dt)
        self.alpha_s = 0.1222 # 7' in rad
        self.F_yf = self.C_af * self.alpha_s
        self.F_yr = self.C_ar * self.alpha_s

    def create_system(self, speed):
        u = speed * (1000 / 3600)
        A = np.array([[-(self.C_af + self.C_ar) / (u * self.m), -((self.a
            * self.C_af - self.b * self.C_ar) / (u * self.m)) - u],
                    [-(self.a * self.C_af - self.b * self.C_ar) / (u *
                        self.I_z), -((self.a ** 2) * self.C_af + (self.b
                        ** 2) * self.C_ar) / (u * self.I_z)]])
        B = np.array([[self.C_af / self.m], [self.a * (self.C_af / self.
            I_z)]])
        C = np.eye(2)
        D = np.zeros((2, 1))
        return signal.StateSpace(A, B, C, D)

    def create_additional_systems(self, speed, condition):
        u = speed * (1000 / 3600)
        if condition == "front_saturated":
            A = np.array([[-self.C_ar / (u * self.m), -((self.b * self.C_ar
```

```
            ) / (u * self.m)) - u],
                    [-(-self.b * self.C_ar) / (u * self.I_z), -((self.
                        b ** 2 * self.C_ar) / (u * self.I_z))]])
        B = np.array([[0], [0]])
    elif condition == "rear_saturated":
        A = np.array([[-self.C_af / (u * self.m), -((self.a * self.C_af
            ) / (u * self.m)) - u],
                    [-(self.a * self.C_af) / (u * self.I_z), -((self.a
                         ** 2 * self.C_af) / (u * self.I_z))]])
        B = np.array([[self.C_af / self.m], [self.a * (self.C_af / self
            .I_z)]])
    elif condition == "both_saturated":
        A = np.array([[0, -u],
                    [0, 0]])
        B = np.array([[0], [0]])
    C = np.eye(2)
    D = np.zeros((2, 1))
    return signal.StateSpace(A, B, C, D)

def simulate(self, speed):
    sys = self.create_system(speed)
    sys_d = sys.to_discrete(self.dt)
    x = np.zeros((2, len(self.t)))
    theta = np.zeros(len(self.t))
    X = np.zeros(len(self.t))
    Y = np.zeros(len(self.t))
    a_lat = np.zeros(len(self.t))
    alpha_f = np.zeros(len(self.t))
    alpha_r = np.zeros(len(self.t))

    alpha_f[0] = self.delta
    alpha_r[0] = 0

    for n in range(1, len(self.t)):
        if alpha_f[n-1] < self.alpha_s and alpha_r[n-1] < self.alpha_s:
            x[:, n] = sys_d.A @ x[:, n-1] + sys_d.B.flatten() * self.
                delta
            c = np.zeros(2)
        elif alpha_f[n-1] >= self.alpha_s and alpha_r[n-1] < self.
            alpha_s:
            # Front saturation
            sys_additional = self.create_additional_systems(speed, "
```

```
                front_saturated")
            sys_d_additional = sys_additional.to_discrete(self.dt)
            c = np.array([self.F_yf / self.m, self.a * self.F_yf / self
                .I_z])
            x[:, n] = sys_d_additional.A @ x[:, n-1] + c * self.dt
        elif alpha_f[n-1] < self.alpha_s and alpha_r[n-1] >= self.
            alpha_s:
            # Rear saturation
            sys_additional = self.create_additional_systems(speed, "
                rear_saturated")
            sys_d_additional = sys_additional.to_discrete(self.dt)
            c = np.array([self.F_yr / self.m, -self.b * self.F_yr /
                self.I_z])
            x[:, n] = sys_d_additional.A @ x[:, n-1] + sys_d_additional
                .B.flatten() * self.delta + c * self.dt
        else:
            # Both saturation
            sys_additional = self.create_additional_systems(speed, "
                both_saturated")
            sys_d_additional = sys_additional.to_discrete(self.dt)
            c = np.array([(self.F_yf + self.F_yr) / self.m, (self.a *
                self.F_yf - self.b * self.F_yr) / self.I_z])
            x[:, n] = sys_d_additional.A @ x[:, n-1] + c * self.dt

        theta[n] = x[1, n] * self.dt + theta[n-1]
        X[n] = (speed * (1000 / 3600) * np.cos(theta[n]) + x[0, n] * np
            .sin(theta[n])) * self.dt + X[n-1]
        Y[n] = (speed * (1000 / 3600) * np.sin(theta[n]) - x[0, n] * np
            .cos(theta[n])) * self.dt + Y[n-1]
        a_lat[n] = (x[0, n] - x[0, n-1]) / self.dt
        alpha_f[n] = self.delta - (x[0, n] + self.a * x[1, n]) / (speed
             * (1000 / 3600))
        alpha_r[n] = (self.b * x[1, n] - x[0, n]) / (speed * (1000 /
            3600))

    return x, theta, X, Y, a_lat, alpha_f, alpha_r

def plot_results(self, speeds):
    fig, axs = plt.subplots(2, 3, figsize=(18, 10))
    labels = ["Lateral Velocity (m/s)", "Yaw Rate (rad/s)", "Lateral
        Acceleration (m/s^2)",
             "Vehicle Path", "Front Slip Angle (rad)", "Rear Slip
```

```
                    Angle (rad)"]

        for speed in speeds:
            x, theta, X, Y, a_lat, alpha_f, alpha_r = self.simulate(speed)
            data = [x[0, :], x[1, :], a_lat, (X, Y), alpha_f, alpha_r]

            for j, d in enumerate(data):
                ax = axs[j // 3, j % 3]
                if j == 3:
                    ax.plot(d[0], d[1], label=f"Speed {speed} km/h",
                        linewidth=2)
                else:
                    ax.plot(self.t, d, label=f"Speed {speed} km/h",
                        linewidth=2)
                ax.set_title(labels[j], fontsize=14)
                ax.set_xlabel("Time (s)" if j != 3 else "X (m)", fontsize
                    =12)
                ax.set_ylabel(labels[j], fontsize=12)
                ax.grid(True, linestyle='--', alpha=0.6)
                ax.legend()
                ax.tick_params(axis='both', which='major', labelsize=10)

        plt.tight_layout()
        plt.savefig("q2_results_3.png")
        plt.show()

    def check_stability(self, speeds):
        for speed in speeds:
            sys = self.create_system(speed)
            poles = np.linalg.eigvals(sys.A)
            print(f"Poles for speed {speed} km/h: {poles}")
            stability = "Stable" if np.all(np.real(poles) < 0) else "
                Unstable"
            print(f"System at {speed} km/h is {stability}\n")

if __name__ == "__main__":
    vehicle = VehicleModel(mass=1780, a=1.32, b=1.46, I_z=4400, C_af
        =70500, C_ar=70500, delta=0.04)
    speeds = [65, 135]
    vehicle.plot_results(speeds)
    vehicle.check_stability(speeds)
```