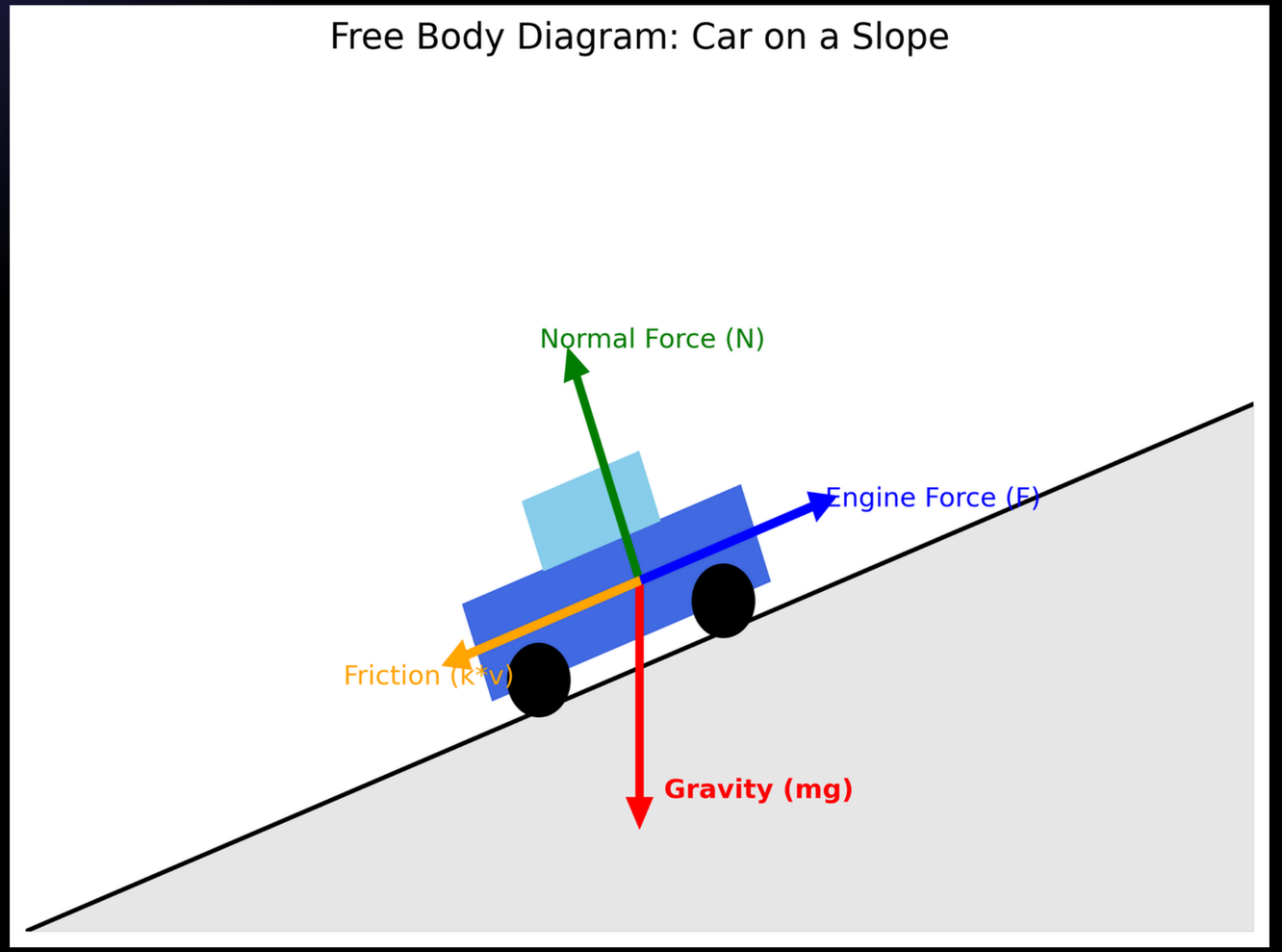
INTRODUCTION:

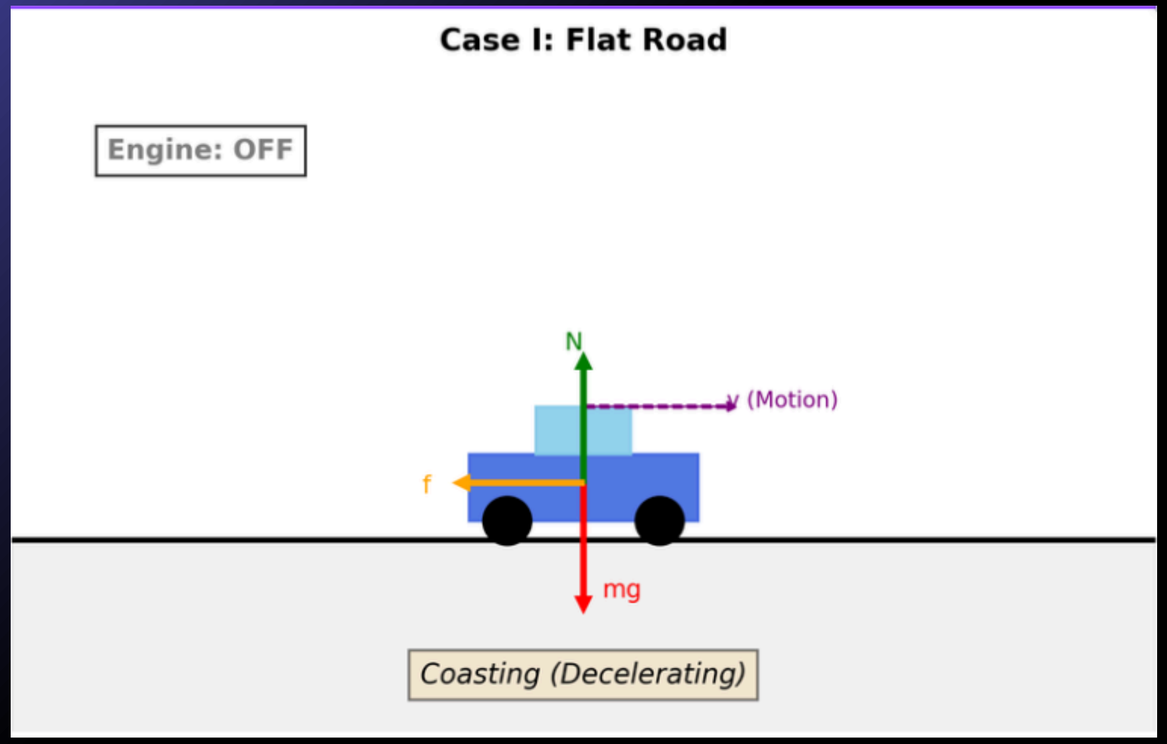
In this project, we designed, modeled, and secured a continuous-time dynamic car-control system. The objective was to integrate principles of control engineering with cybersecurity by modeling the vehicle's motion, designing a PID controller to regulate velocity, and finally, simulating cyber-attacks to analyze system resilience. We used MATLAB/Simulink and Python to simulate the physical plant and the controller logic

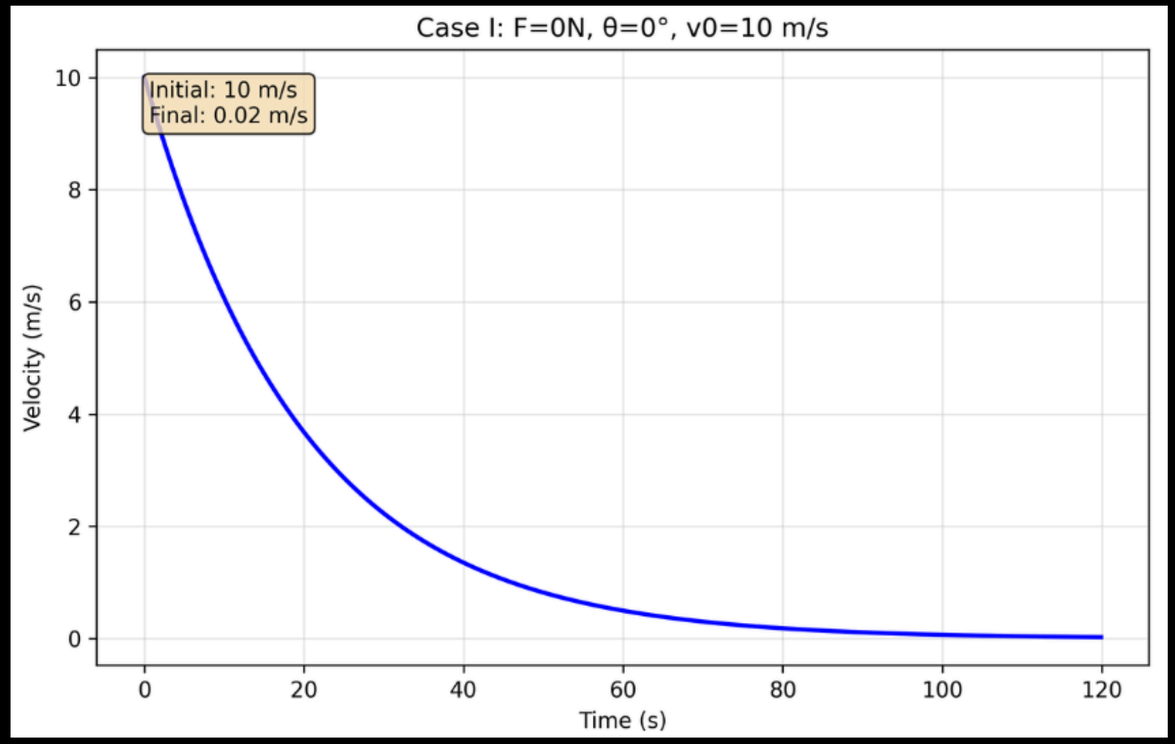
Dynamic Car Modeling

In this phase, we modeled the car to see how it moves with different forces and road slopes. We used Python to simulate four cases:

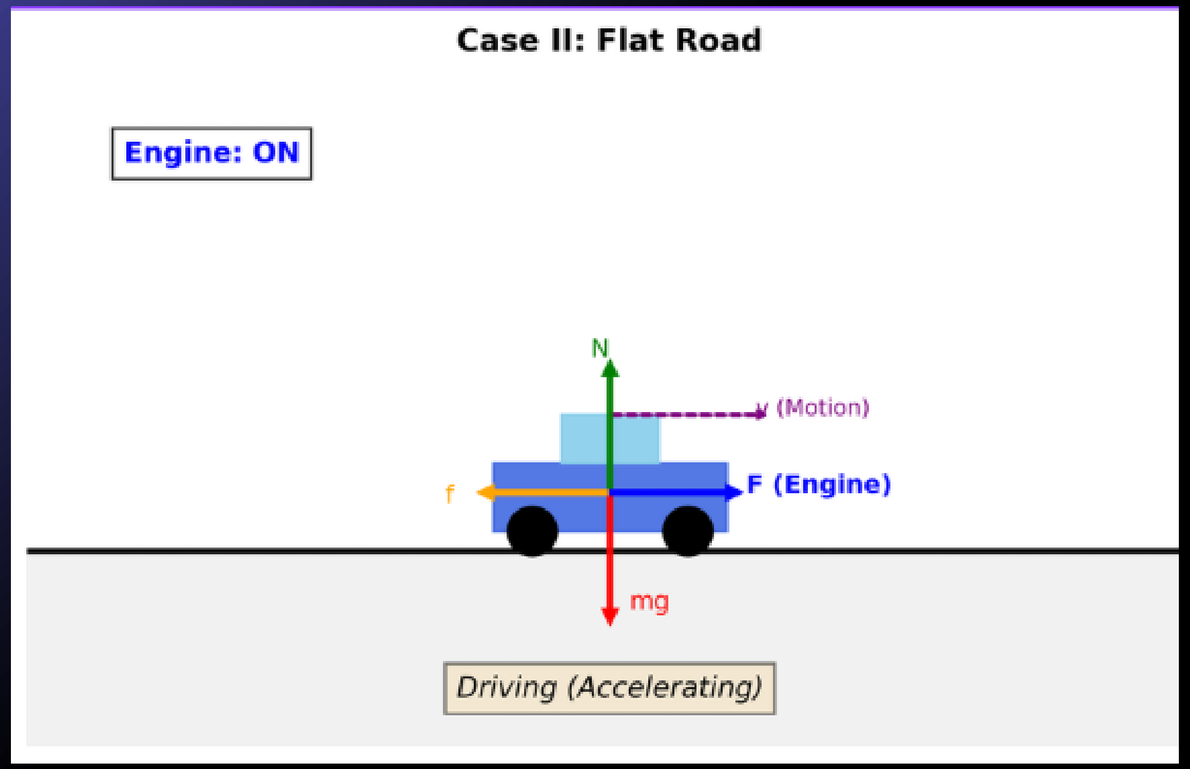


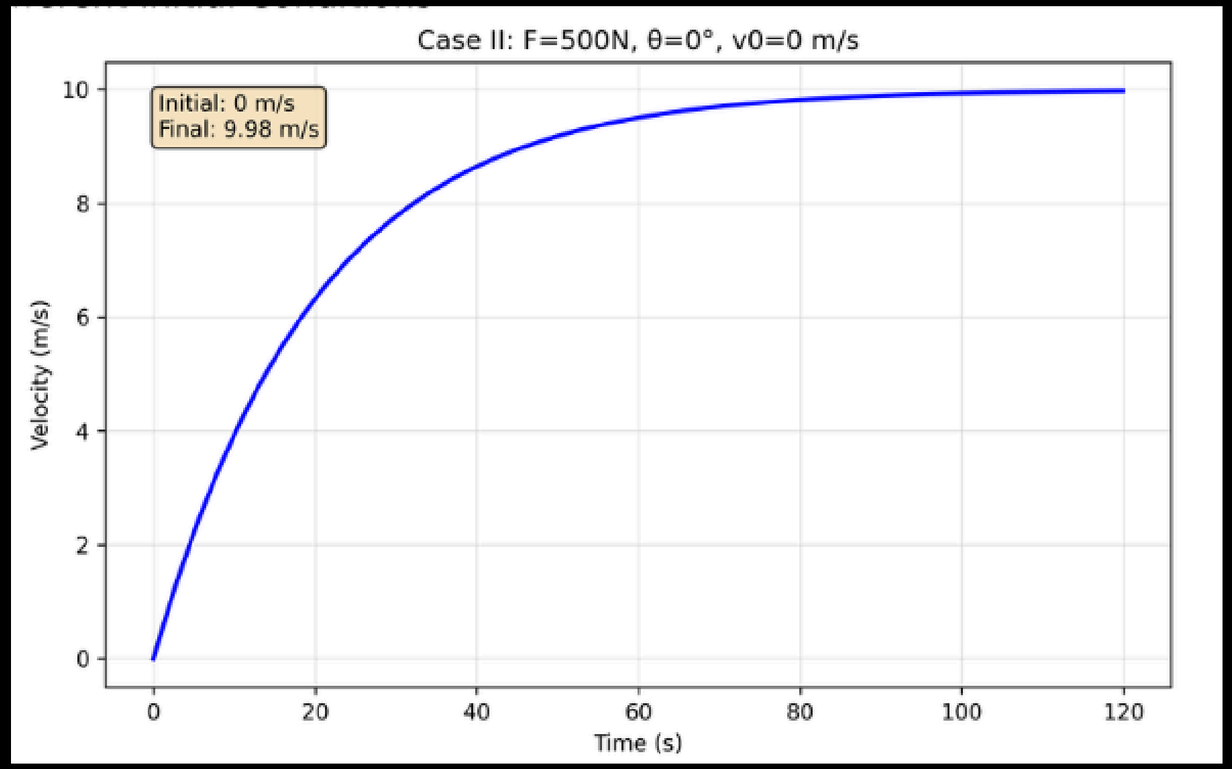
Case I (Flat Road): With the engine off (F=0), the car slowed down until it stopped. This happens because friction is always pushing back against the car.





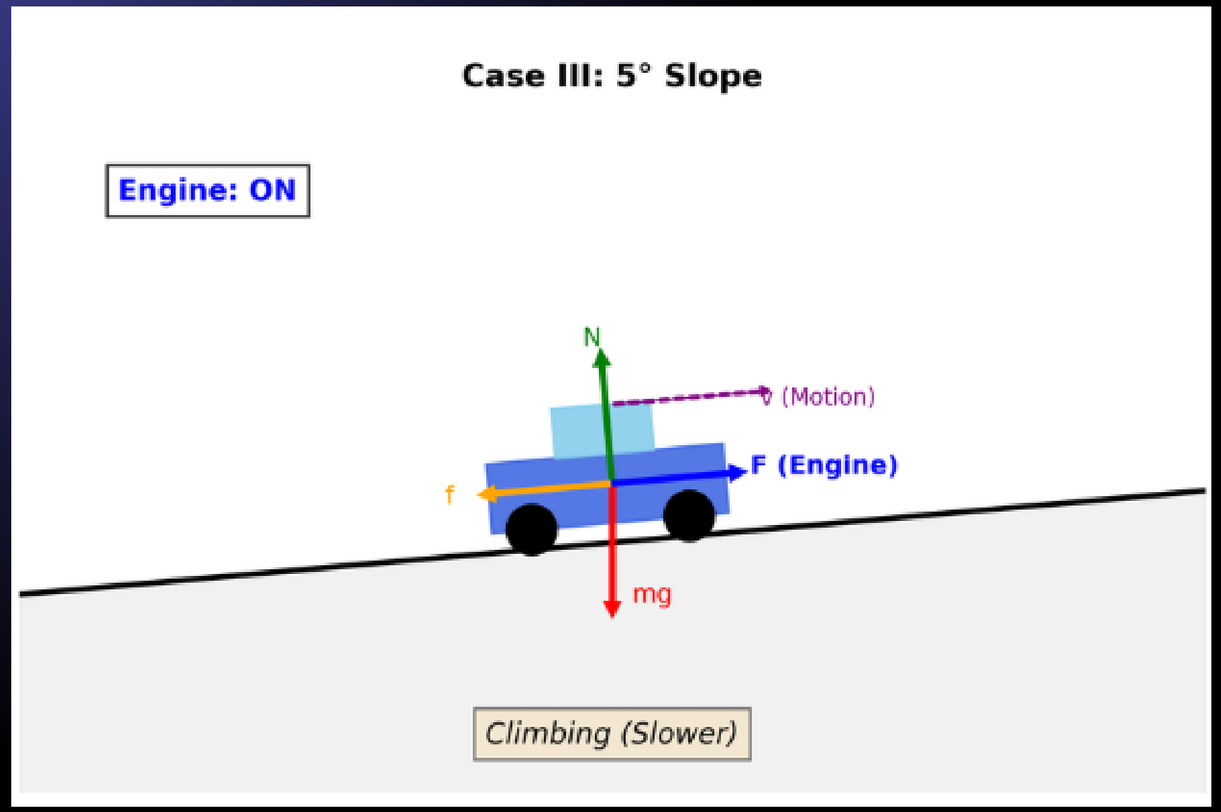
Case II (Flat Road): We turned on the engine to 500 N. The car speed went up and stayed at 10 m/s because the engine push was equal to the friction push.

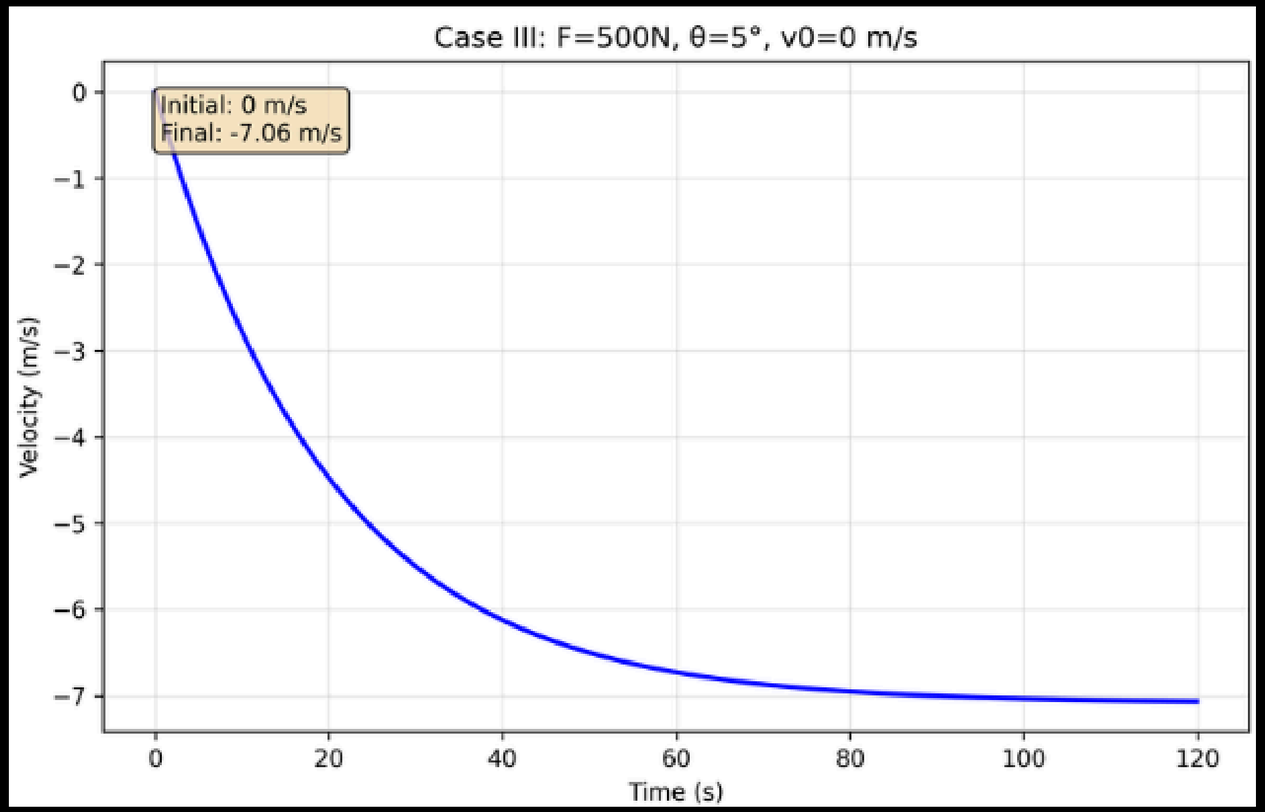




Case III & IV (Hilly Road):

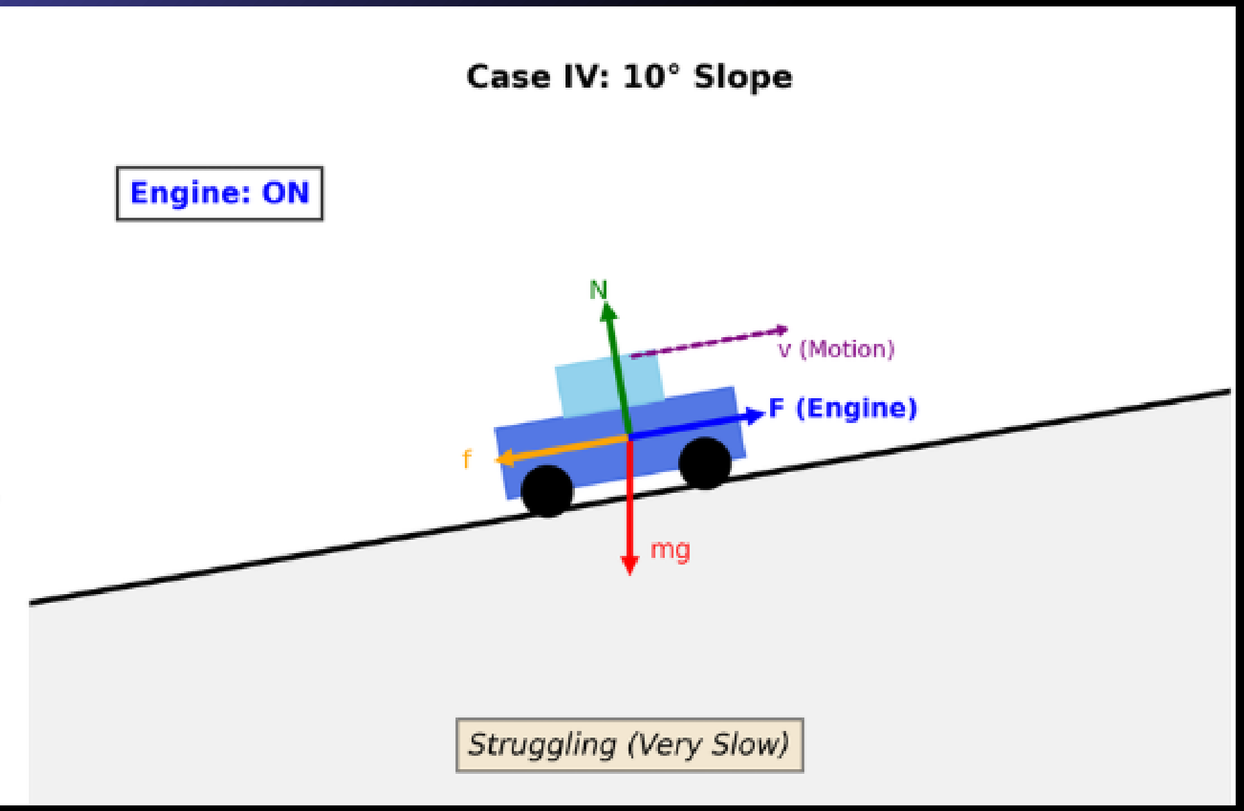
Case III: the car became very slow, even though the engine was still pushing with 500 N.

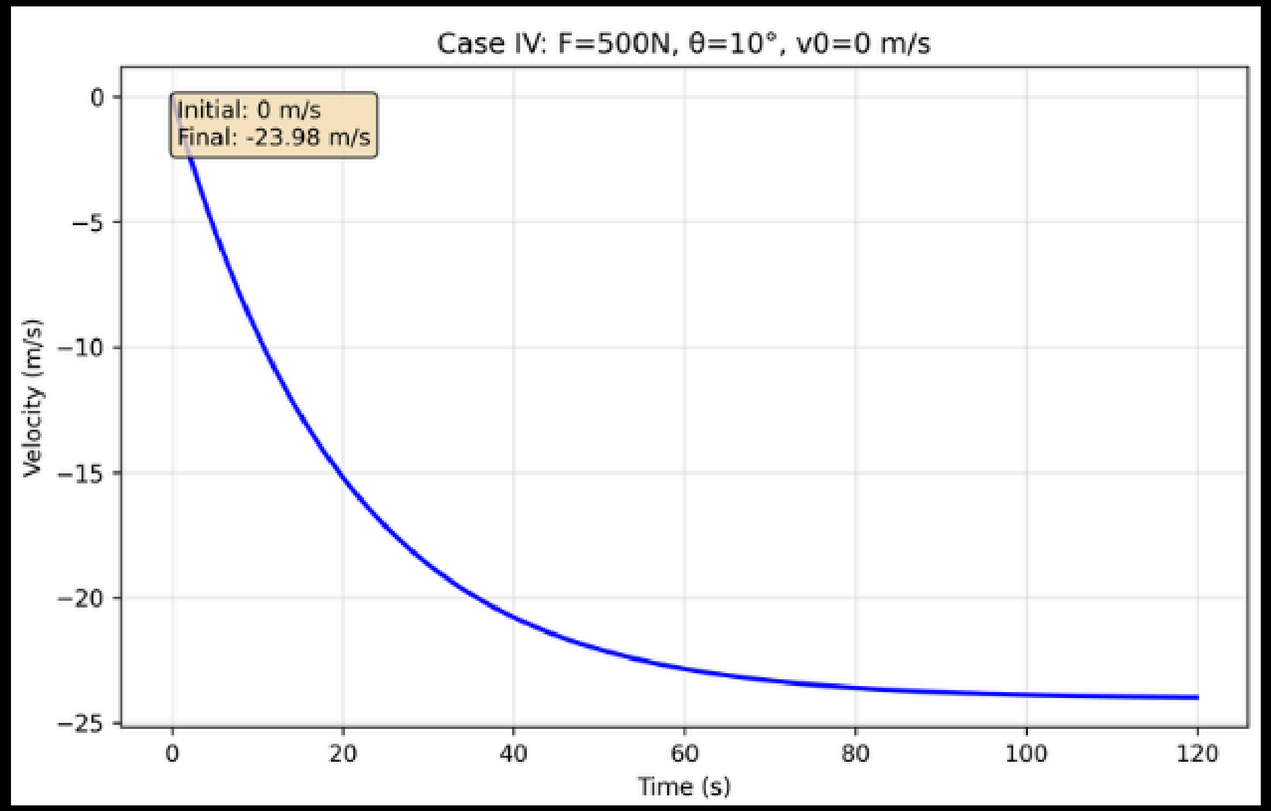




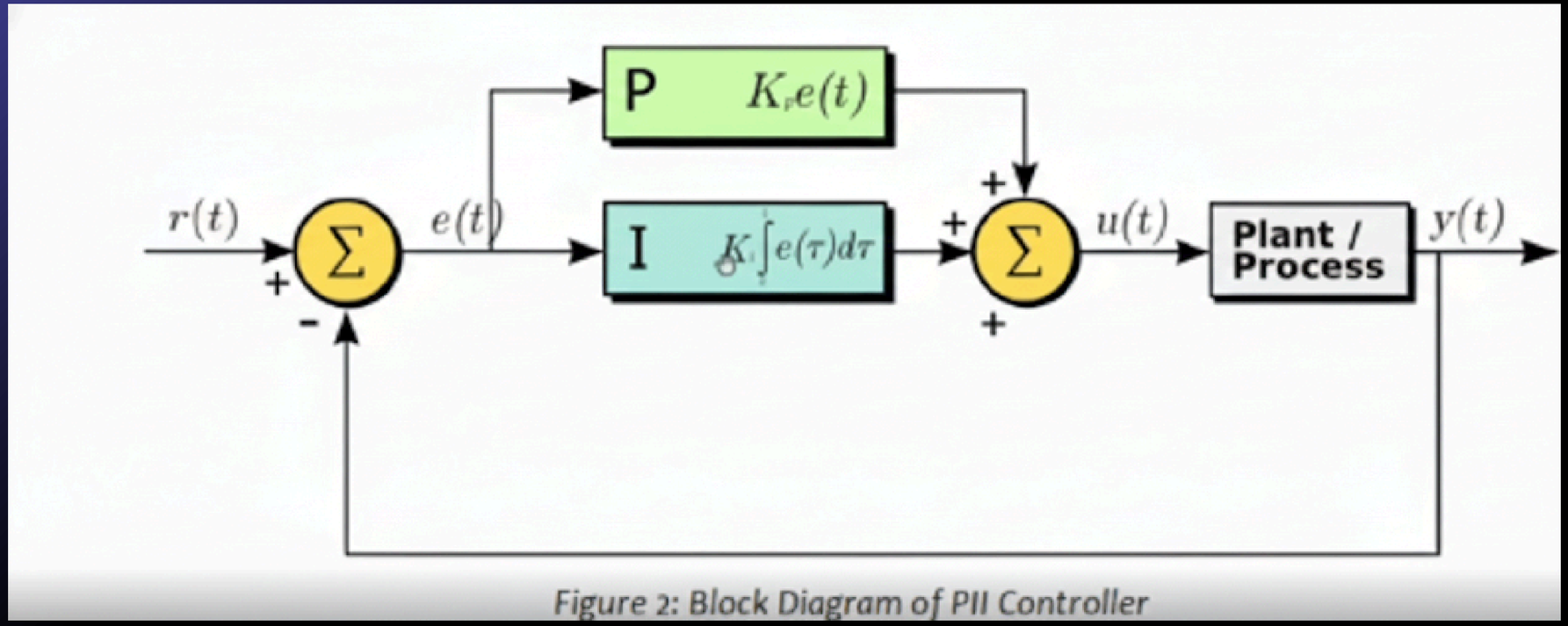
Case III & IV (Hilly Road):

Case IV: On the steep 10-degree hill, the car could barely move.





Task 2.0:Controller Block Diagrams (P and PI)



The diagram shows how the P and PI controllers work.

The error is found by comparing the reference speed with the actual speed.

The P part uses Kp * $e(t)$, and the PI controller also adds an integral term to reduce long-term error.

The control signal drives the plant, and the output speed is fed back to keep correcting the error.

Task 2.1: Derivation of P and PI Controller Equations

The error is defined as → $e(t)=R-v(t)$

```
for i = 1:N
    e = R_local - v;
```

The difference between the desired speed and the actual speed.

The P controller uses → $u(t)=K_p^* e(t)$

The P controller is applied when Ki = 0

```
for i = 1:N
    e = R_local - v;
    if Ki == 0
        u_cmd = Kp * e;
```

The controller output is directly proportional to the error

The PI controller uses → $u(t)=K_p^* e(t) + K_i * \int e(t)dt$

The PI controller adds the integral accumulation when Ki > 0

```
if (u_cmd ~= u_sat) && (Ki ~= 0)
    integral = integral - e * dt_local;
    u_cmd = Kp * e + Ki * integral;
```

The integral term helps remove steady-state error.

2.3: Controller Implementation (MATLAB)

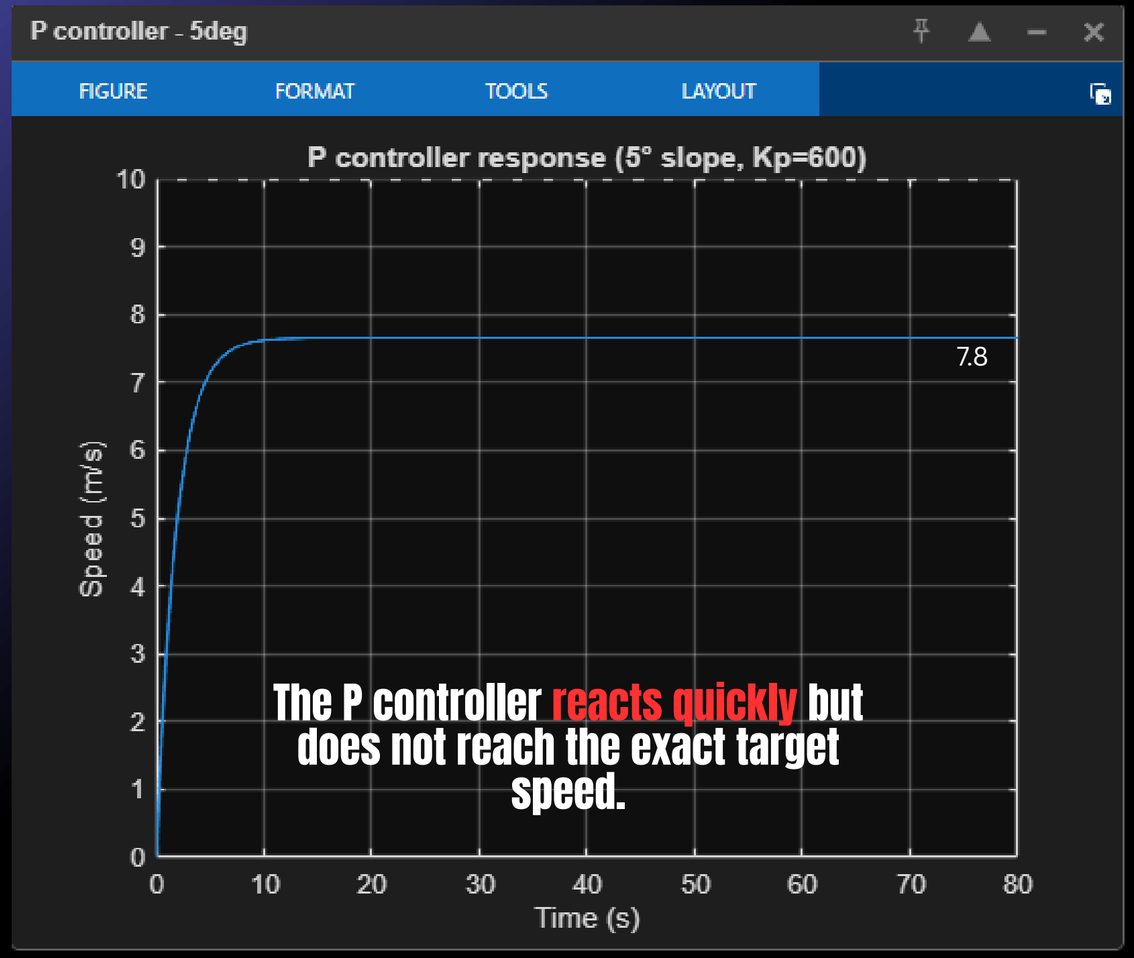
Run_controller in matlab

```
%% ============================
function [v_hist, u_hist] = run_controller(Kp, Ki, theta_profile, R_local, dt_local, m_local, g_local, b_local, u_min_local, u_max_local)
    N = length(theta_profile);
    v = 0;
    integral = 0;
    v_hist = zeros(1,N);
    u_hist = zeros(1,N);
    for i = 1:N
        e = R_local - v;
        if Ki == 0
            u_cmd = Kp * e;
        else
            integral = integral + e * dt_local;
            u_cmd = Kp * e + Ki * integral;
        end
```

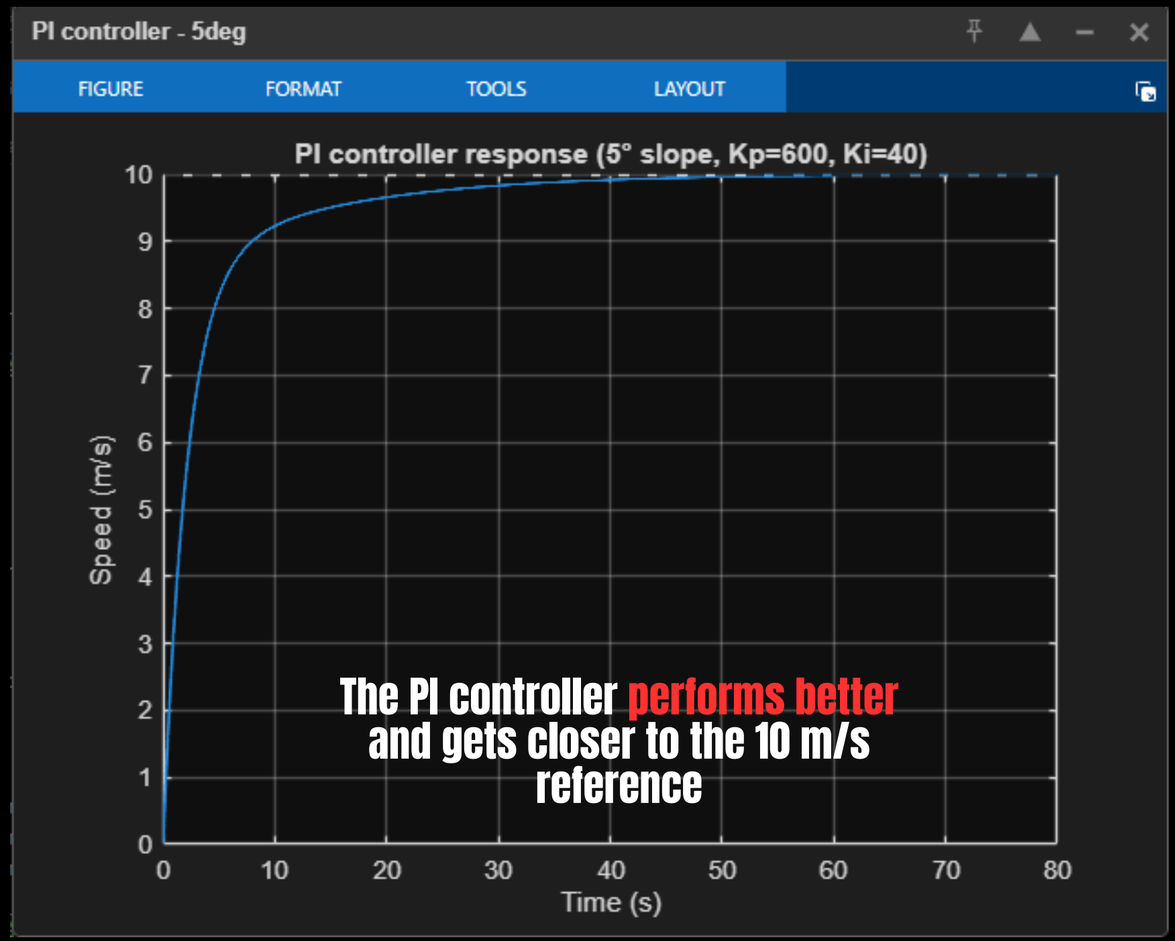
```
% ------------------------------
[vP5, uP5]     = run_controller(Kp_P, 0,    theta_5,  R, dt, m, g, b, u_min, u_max);
[vP10, uP10]   = run_controller(Kp_P, 0,    theta_10, R, dt, m, g, b, u_min, u_max);
[vPvar, uPvar] = run_controller(Kp_P, 0,    theta_var, R, dt, m, g, b, u_min, u_max);

[vPI5, uPI5]     = run_controller(Kp_PI, Ki_PI, theta_5,  R, dt, m, g, b, u_min, u_max);
[vPI10, uPI10]   = run_controller(Kp_PI, Ki_PI, theta_10, R, dt, m, g, b, u_min, u_max);
[vPIvar, uPIvar] = run_controller(Kp_PI, Ki_PI, theta_var, R, dt, m, g, b, u_min, u_max);
```

2.4. Simulation Results (Plots)
Simulation for → 5° For each P, PI

For the P controller , 5 degree Slope Kp=600

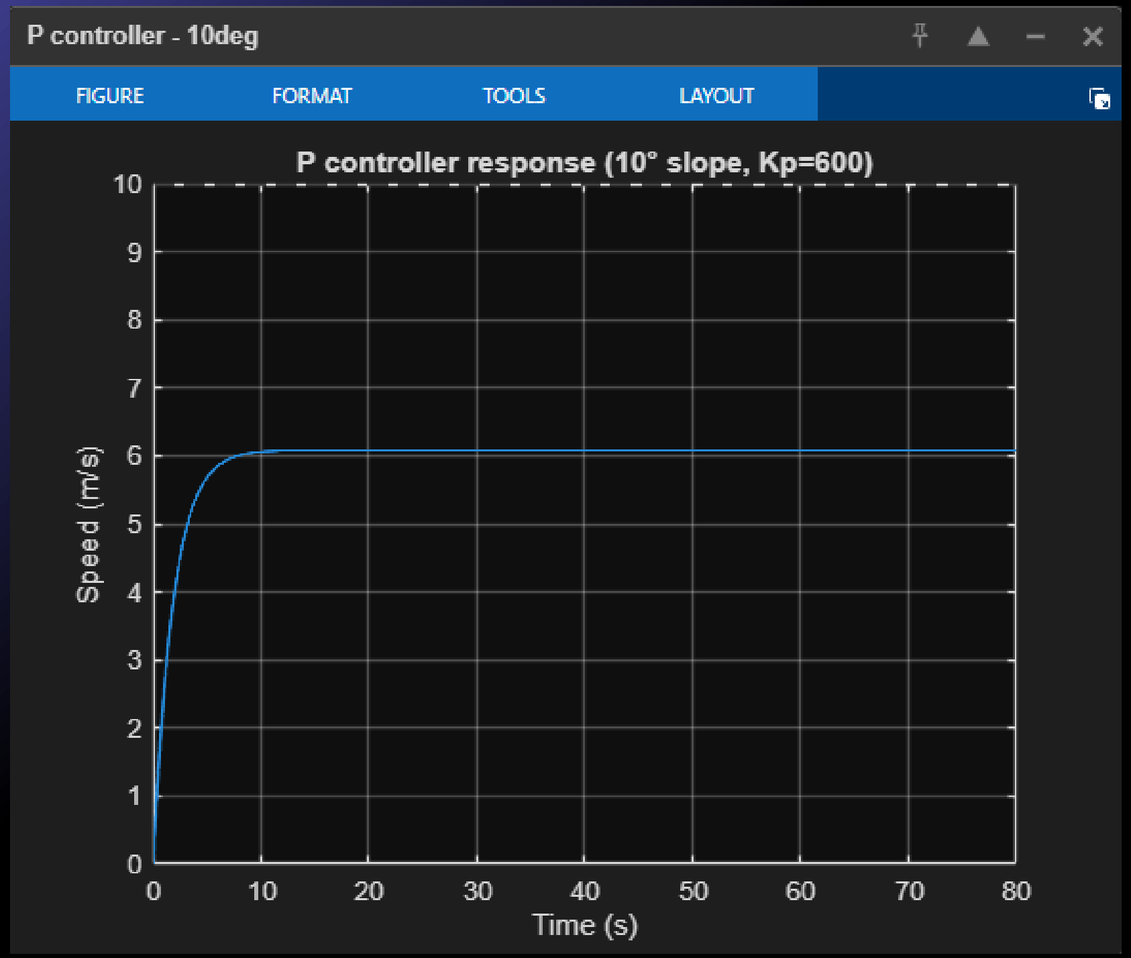


For the Pi controller , 5 degree Slope Kp=600 , Ki=40

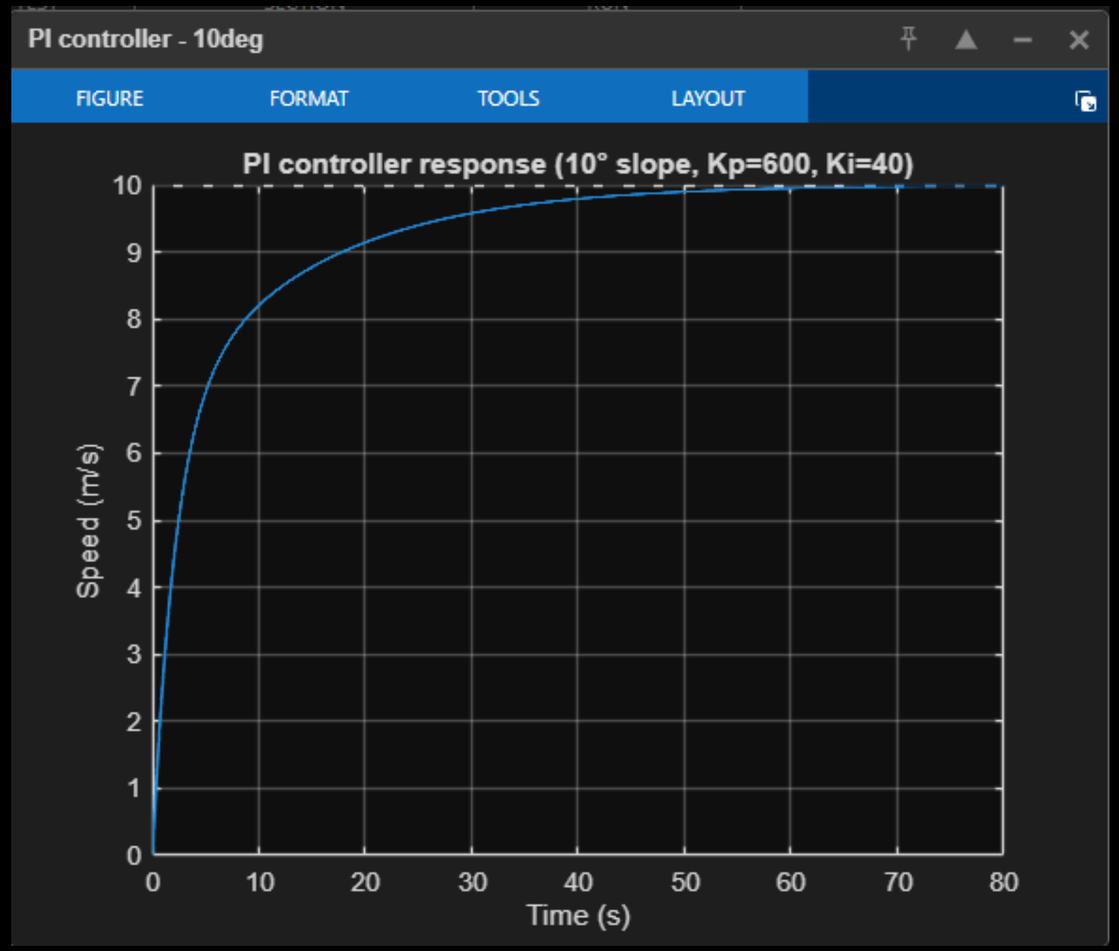


2.4. Simulation Results (Plots)
Simulation for → 10° For each P, PI

For the P controller , 10 degree Slope Kp=600

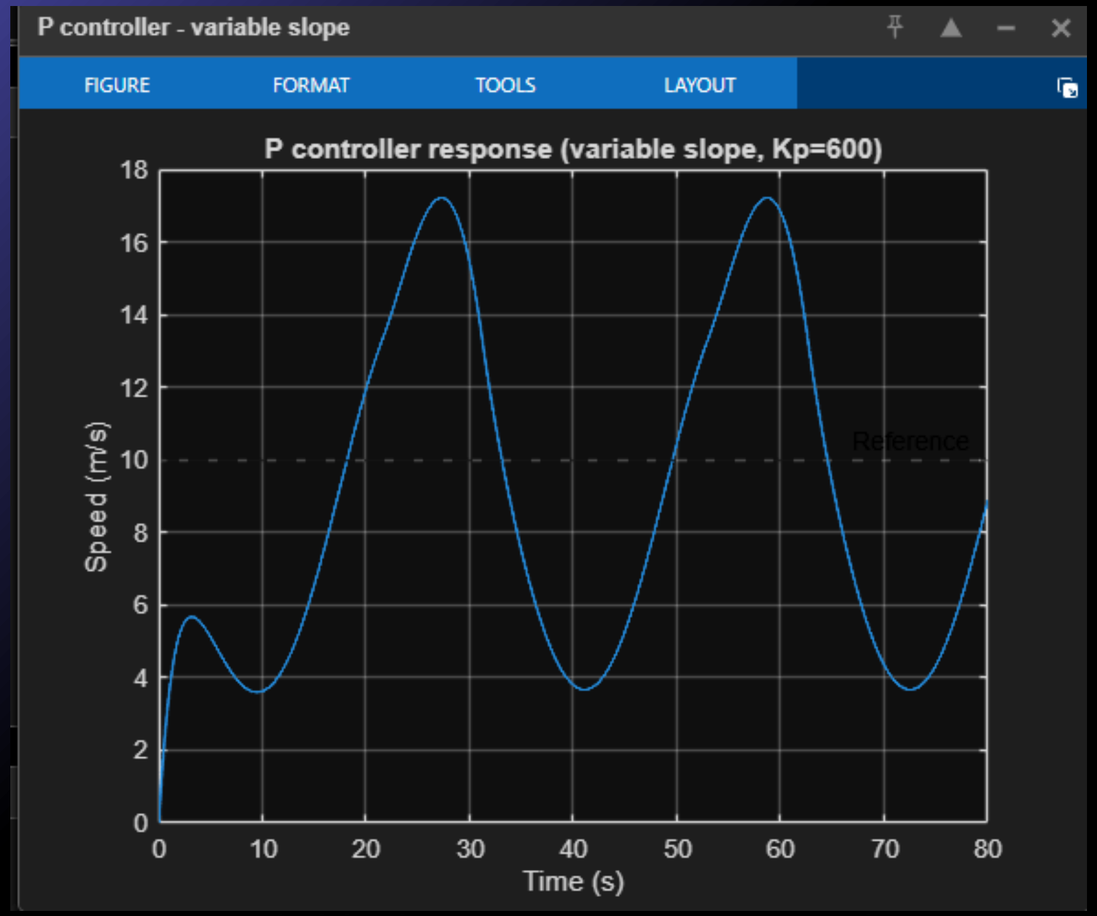


For the PI controller , 10 degree Slope Kp=600 ,Ki=40



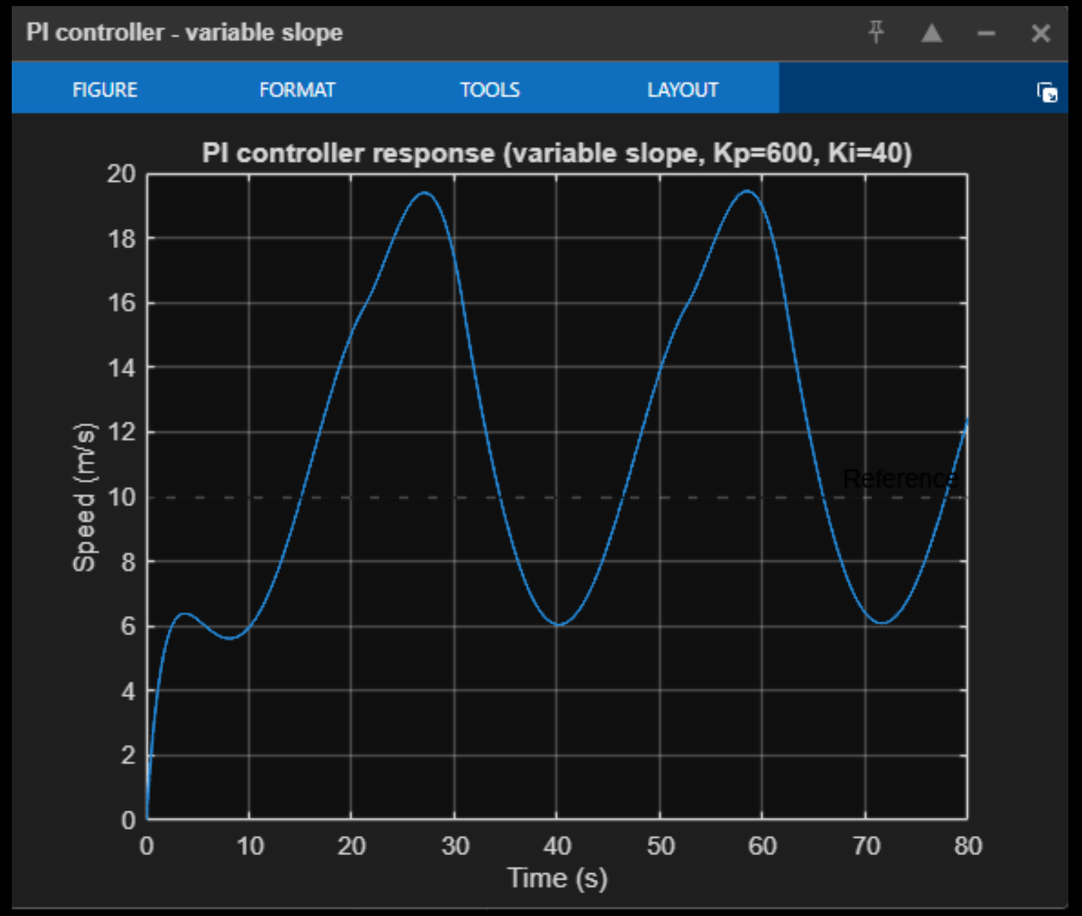
2.4. Simulation Results (Plots)
Variable slope

P Controller Response ,Variable slope , Kp= 600



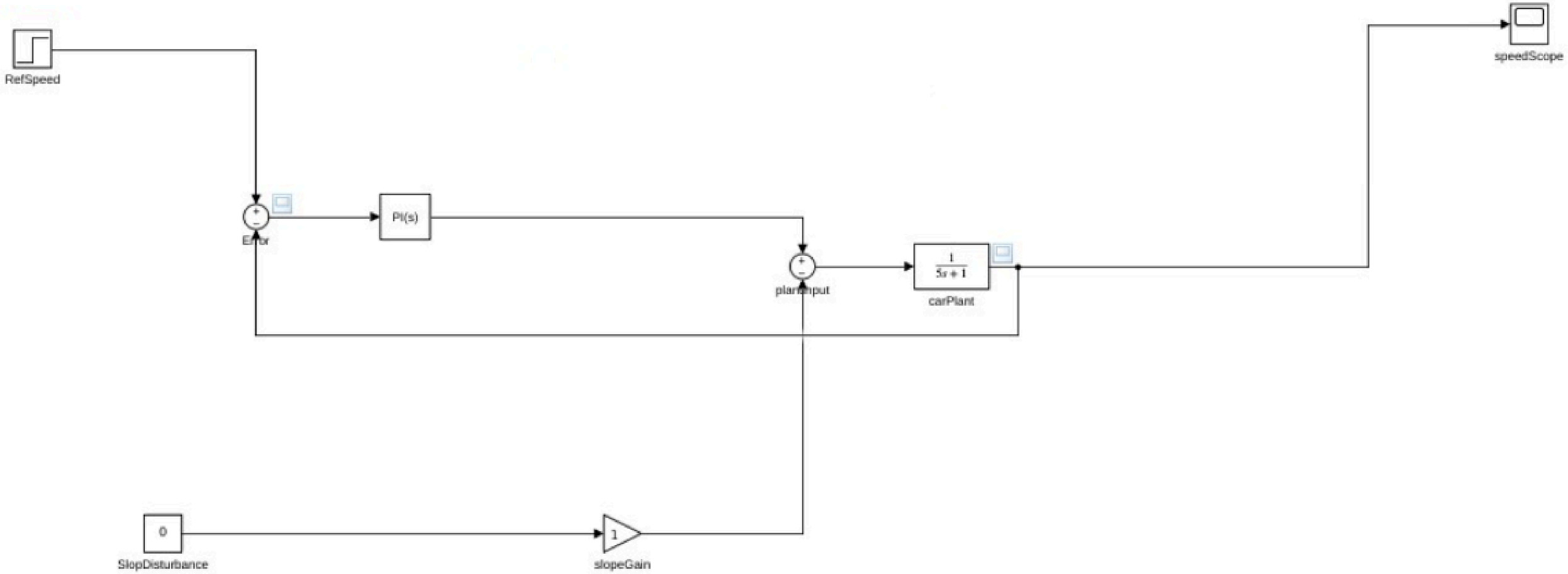
Reacts quickly but cannot correct long-term

PI Controller Response ,Variable slope , Kp= 600, Ki=40



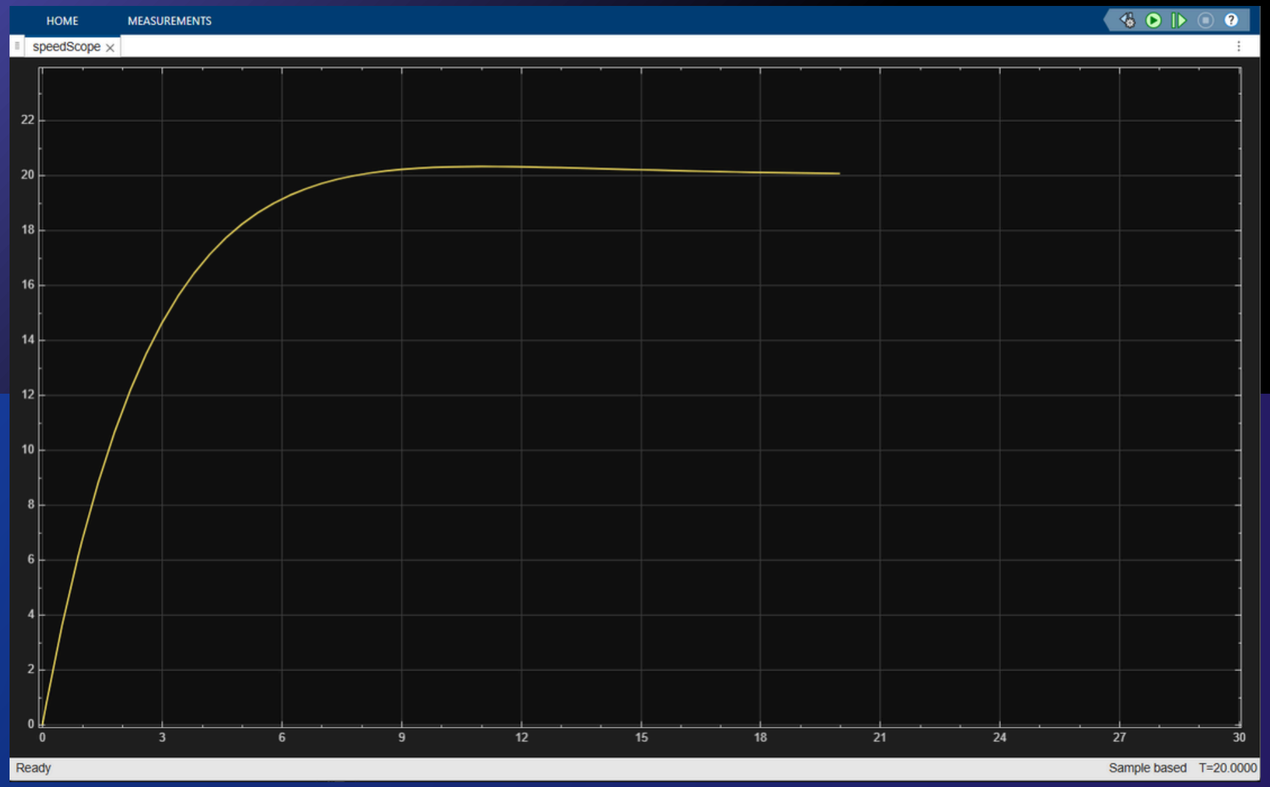
Reduces steady-state error significantly.

TASK 3

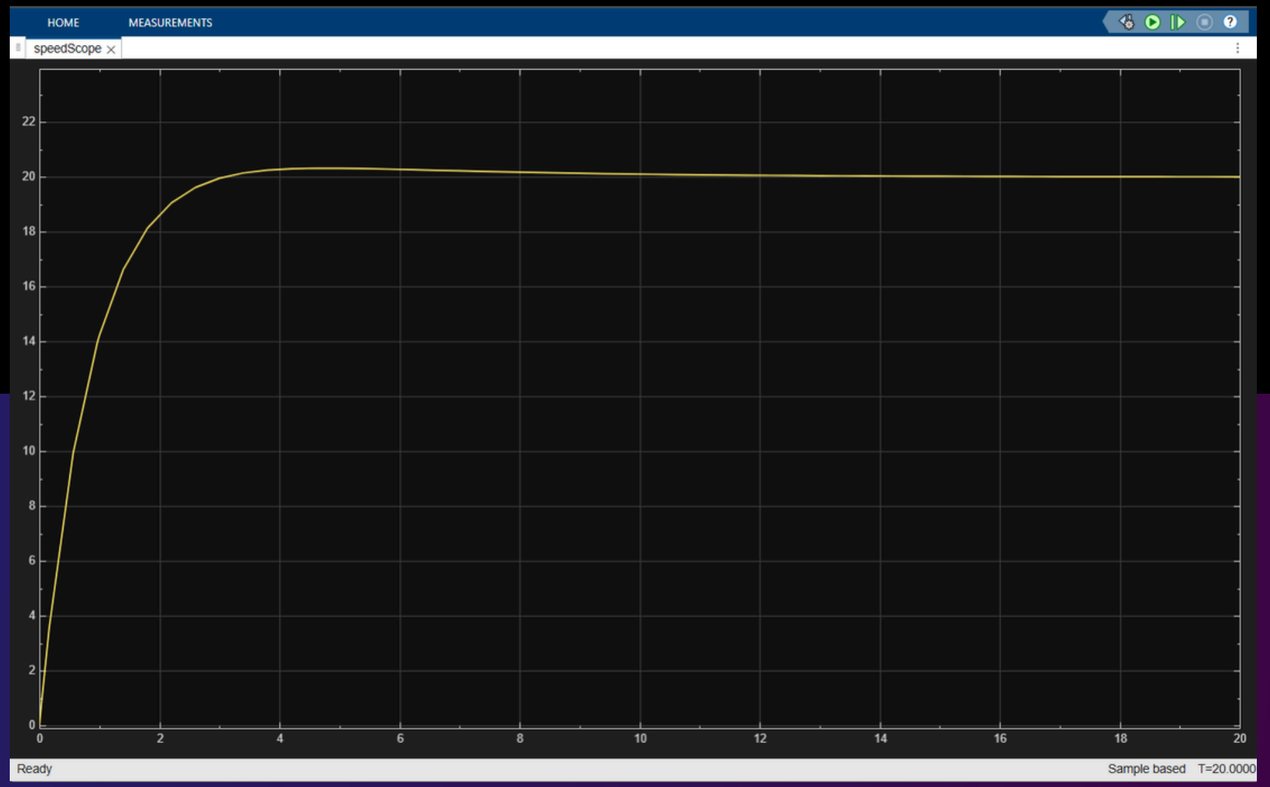

RefSpeed
Error
PI(s)
plantInput
1
5s + 1
carPlant
speedScope
0
SlopDisturbance
1
slopeGain

SIMULATE USING DIFFERENT SLOPES (CONSTANT VS TIME VARIANT SLOPES) WITH DIFFERENT KP , KI VALUES

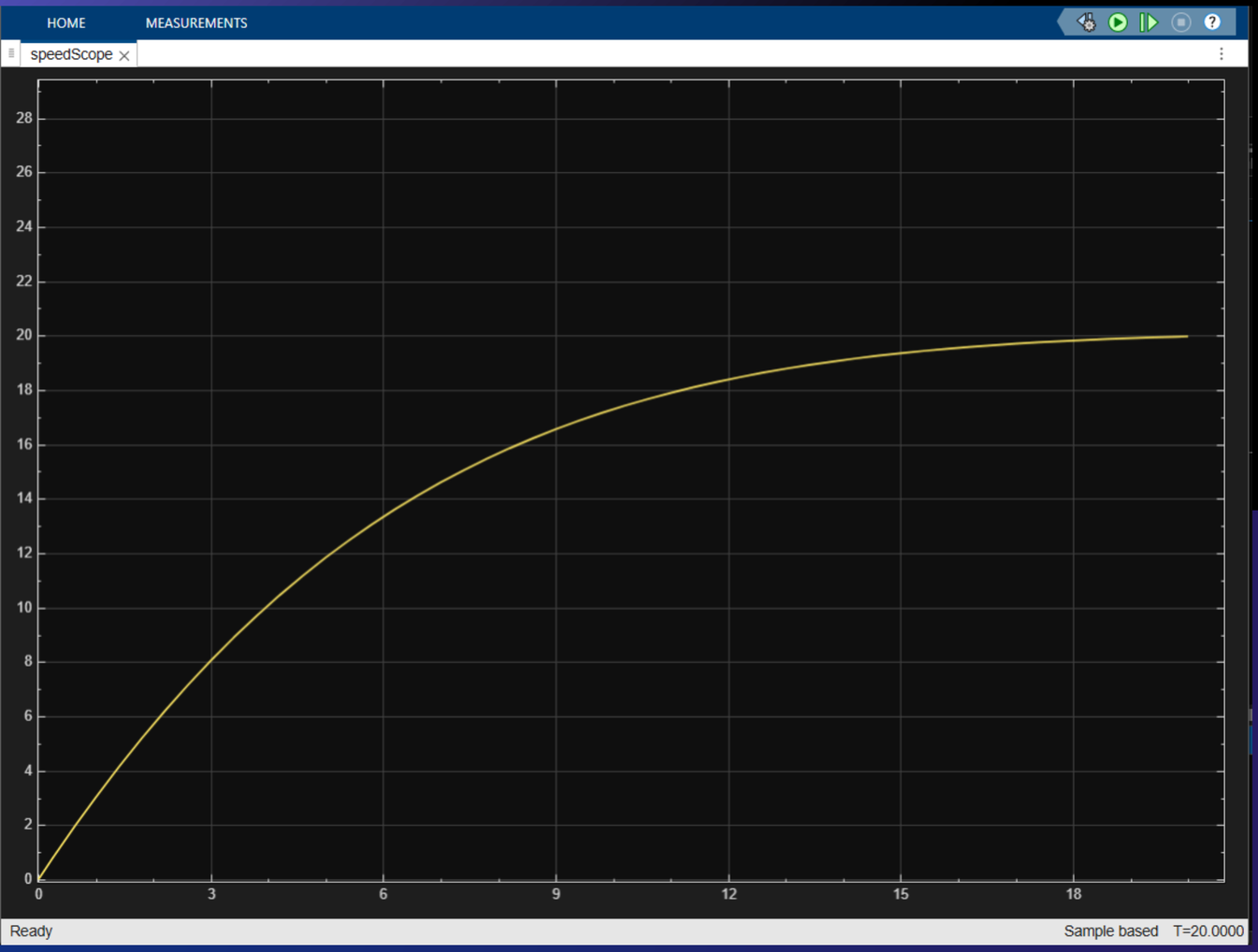
CASE 1 → KP = 2, KI = 0.5



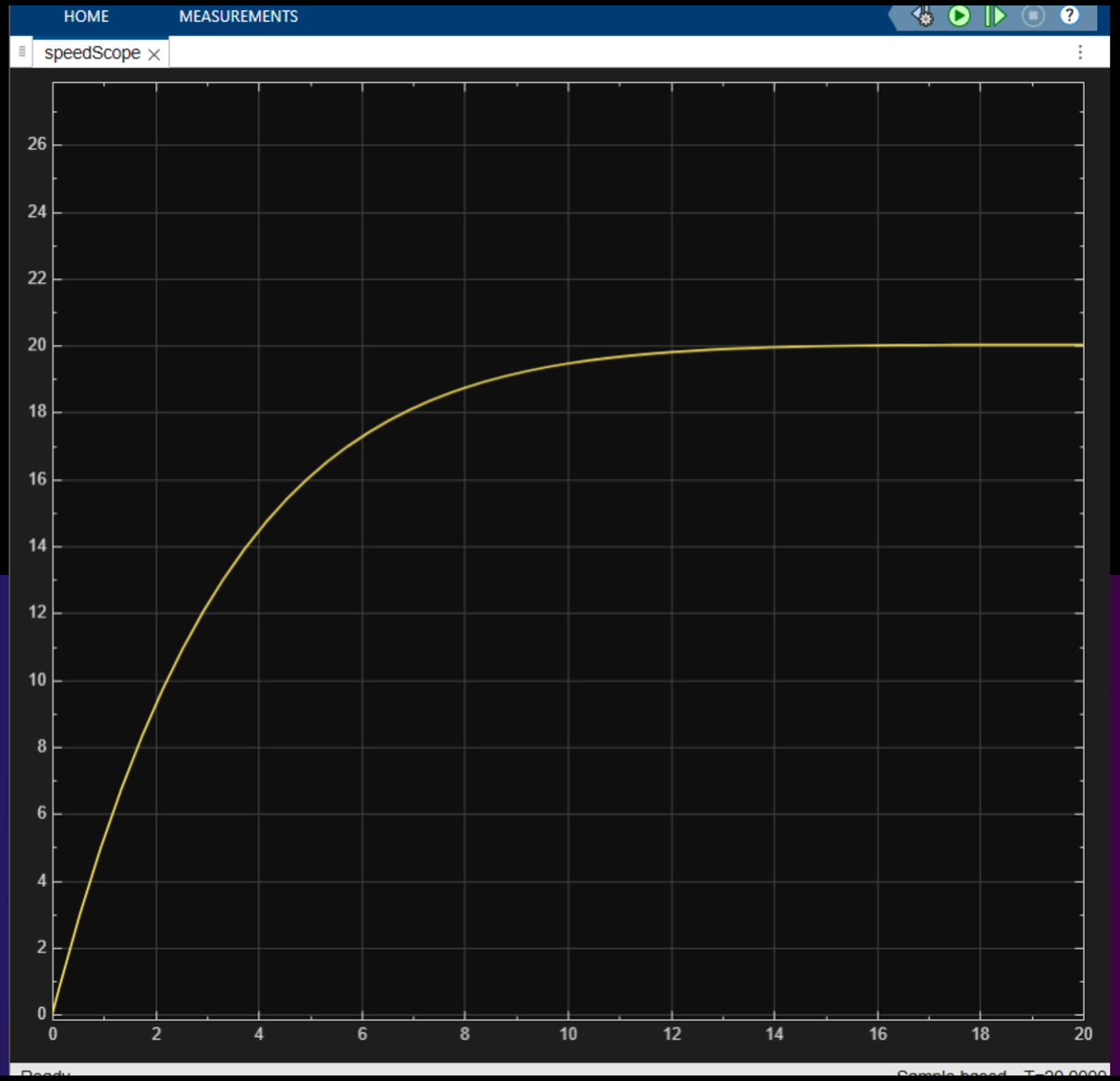
CASE 2 → KP = 10, KI = 4.5



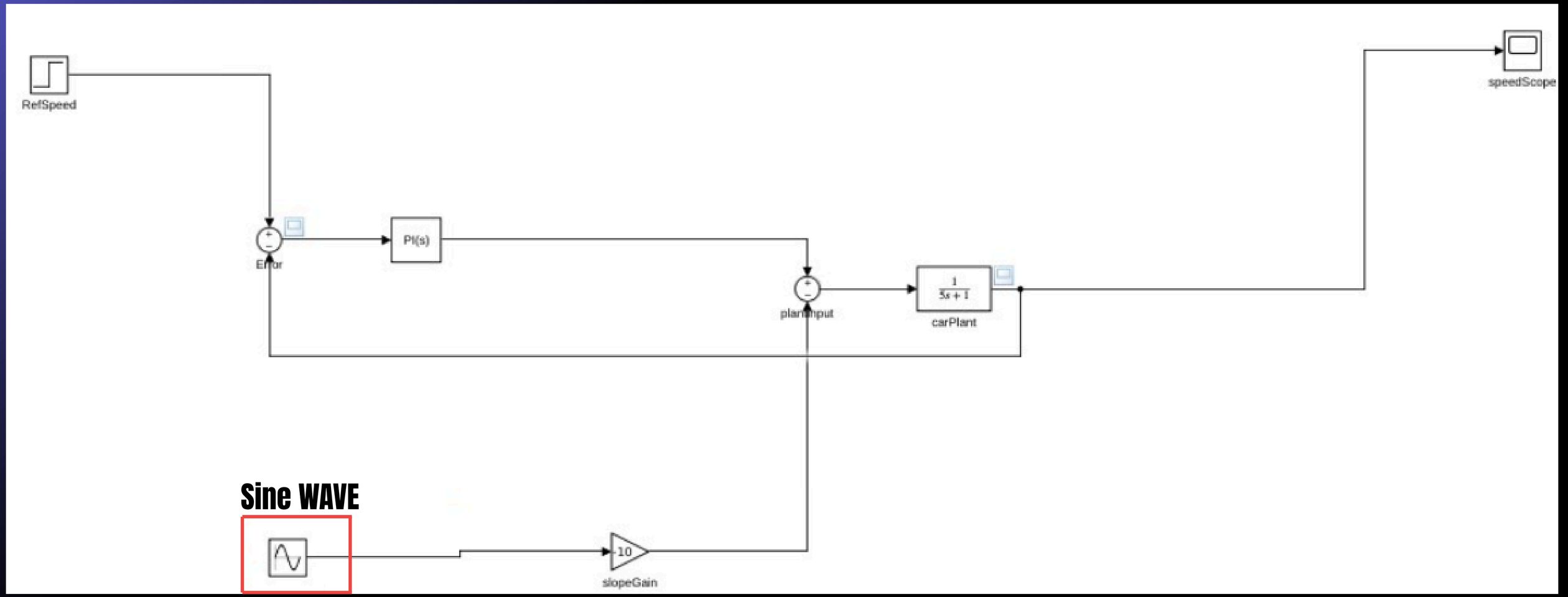
CASE 3 → KP = 0.8, KI = 0.2



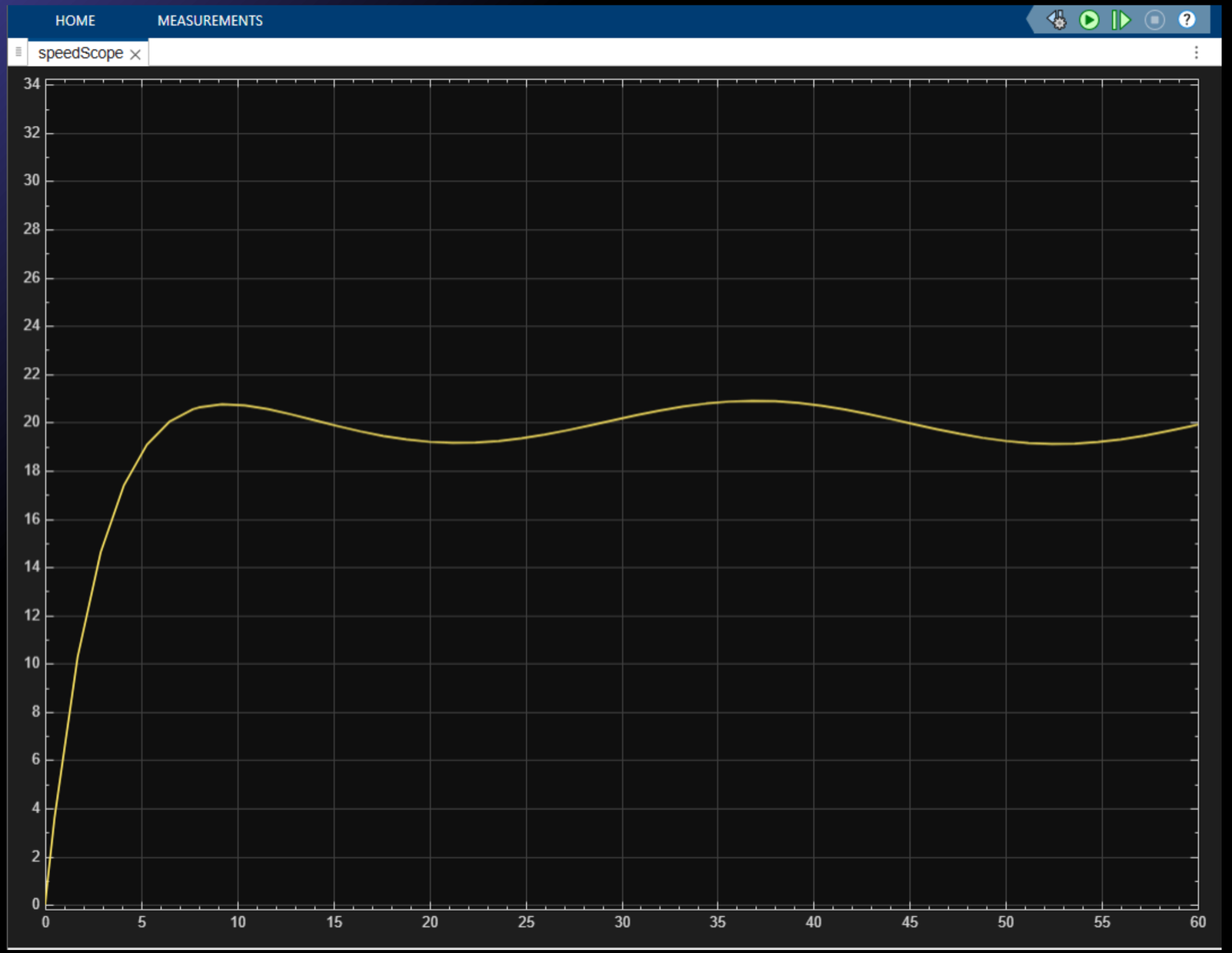
CASE 4 → KP = 2, KI = 0.5 + SLOPE 10



COMPONENTS WITH (SINE WAVE) VARIABLE SLOP



CASE 5 → KP = 2, KI = 0.5 + VARIABLE SLOPE



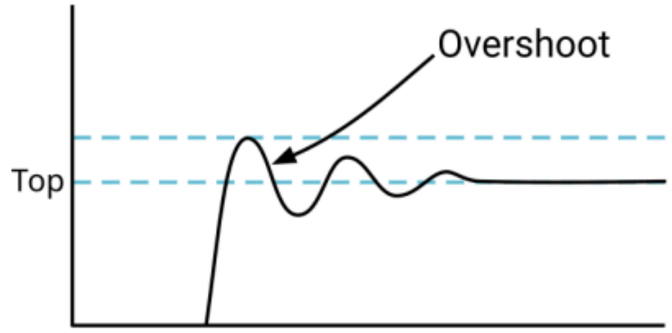
TASK 4

Overshoot

Overshoot --> how much the speed goes above the target speed before settling down

If overshoot is high → the controller reacts too fast or is too aggressive

Overshoot (%) = ((PeakValue - ReferenceValue) / ReferenceValue) * 100



Settling Time

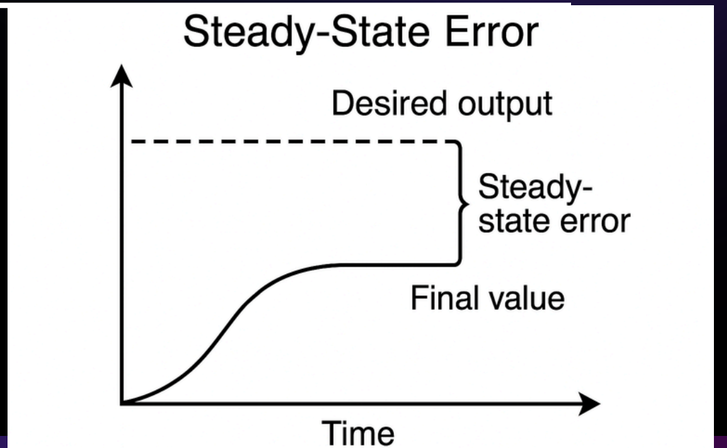
Settling time is how long it takes until the speed becomes stable and stays within a small band around the target

Steady-State Error

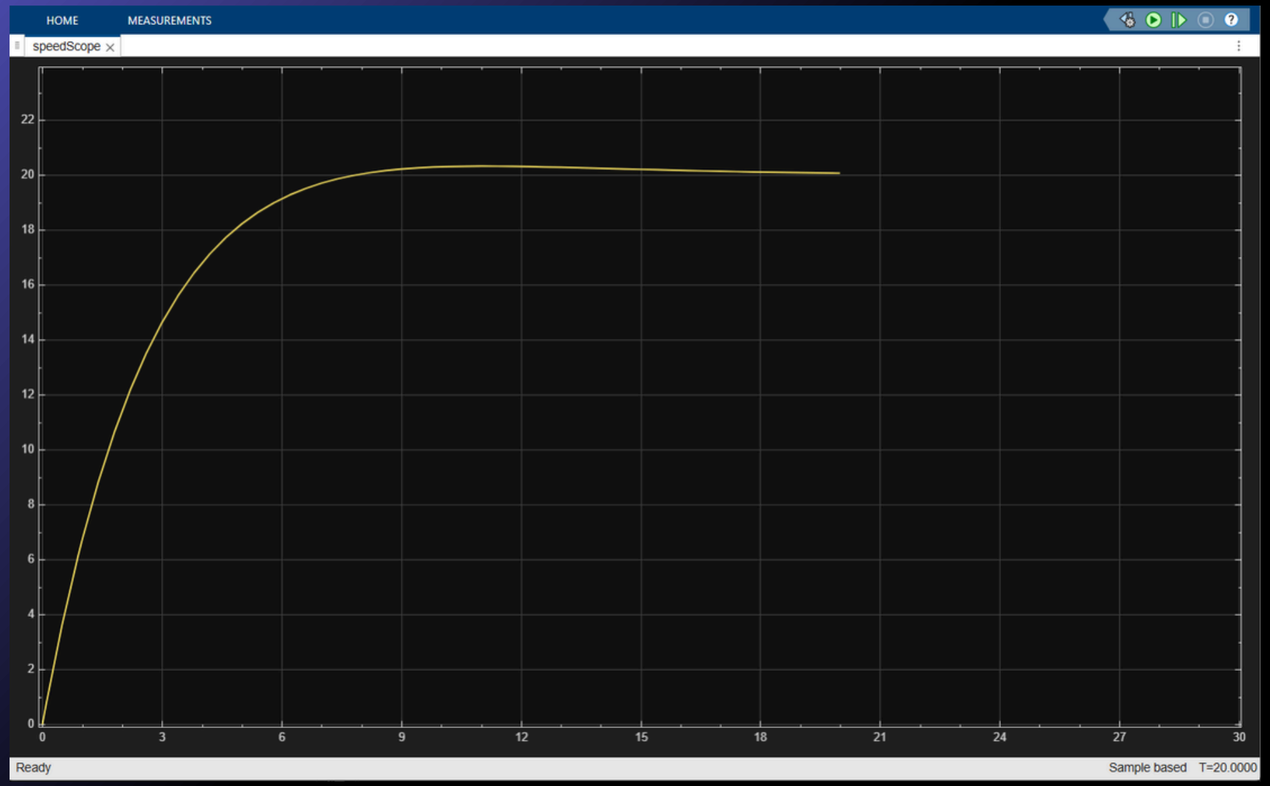
SSE tells you how close the final speed is to the target speed.
If SSE = 0 → perfect tracking.

If SSE is negative → final speed is lower than desired.

SSE = FinalValue - ReferenceValue



Case 1 → Kp = 2, Ki = 0.5

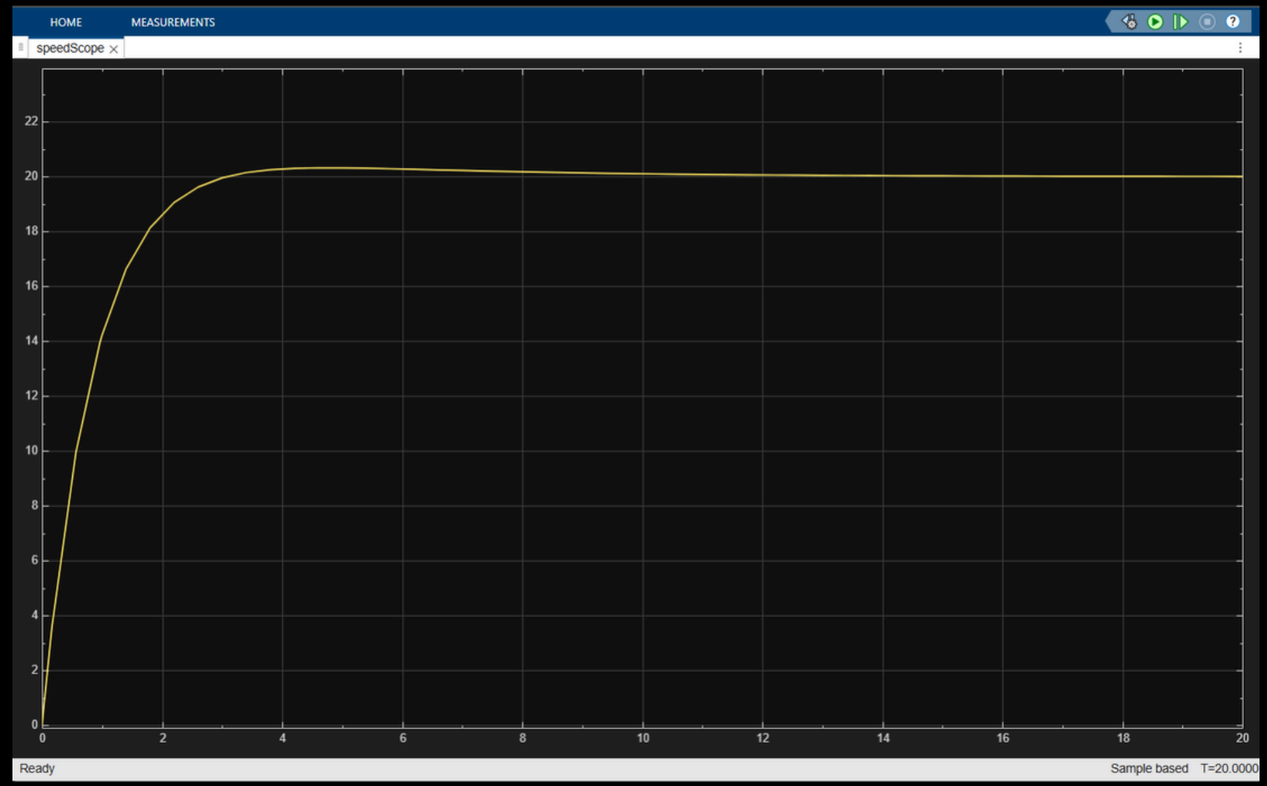


Overshoot: OS=(20.3-20)/ 20×100=1.5%

Settling Time ≈ 7 s

SSE: 20.0657-20=+0.0657 m/s

CASE 2 – Aggressive Gains, Kp = 10, Ki = 4.5

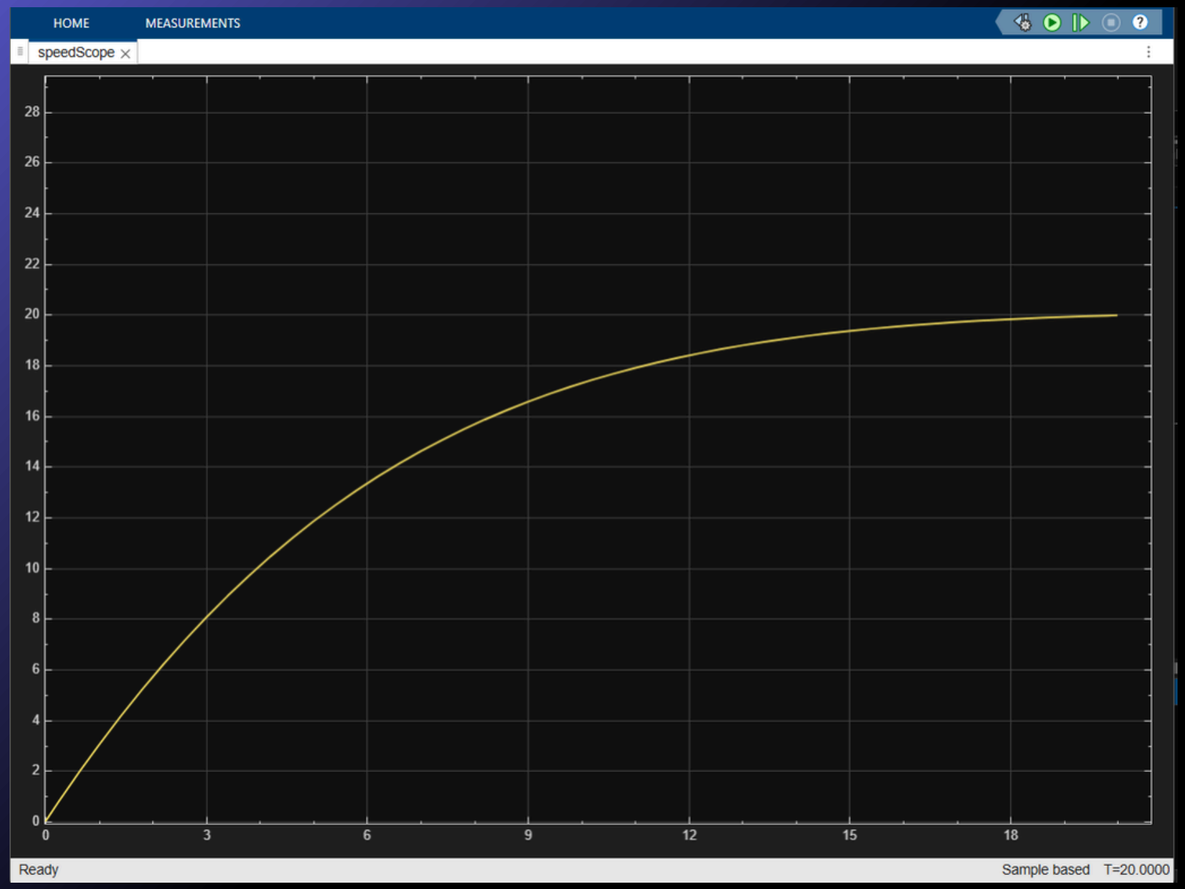


OS = (21.1805 - 20) / 20 × 100 = 5.9%

Settling Time ≈ 4s

SSE: 20.001 - 20 = +0.001 m/s

CASE 3 – Low Gains, Kp = 0.8, Ki = 0.2

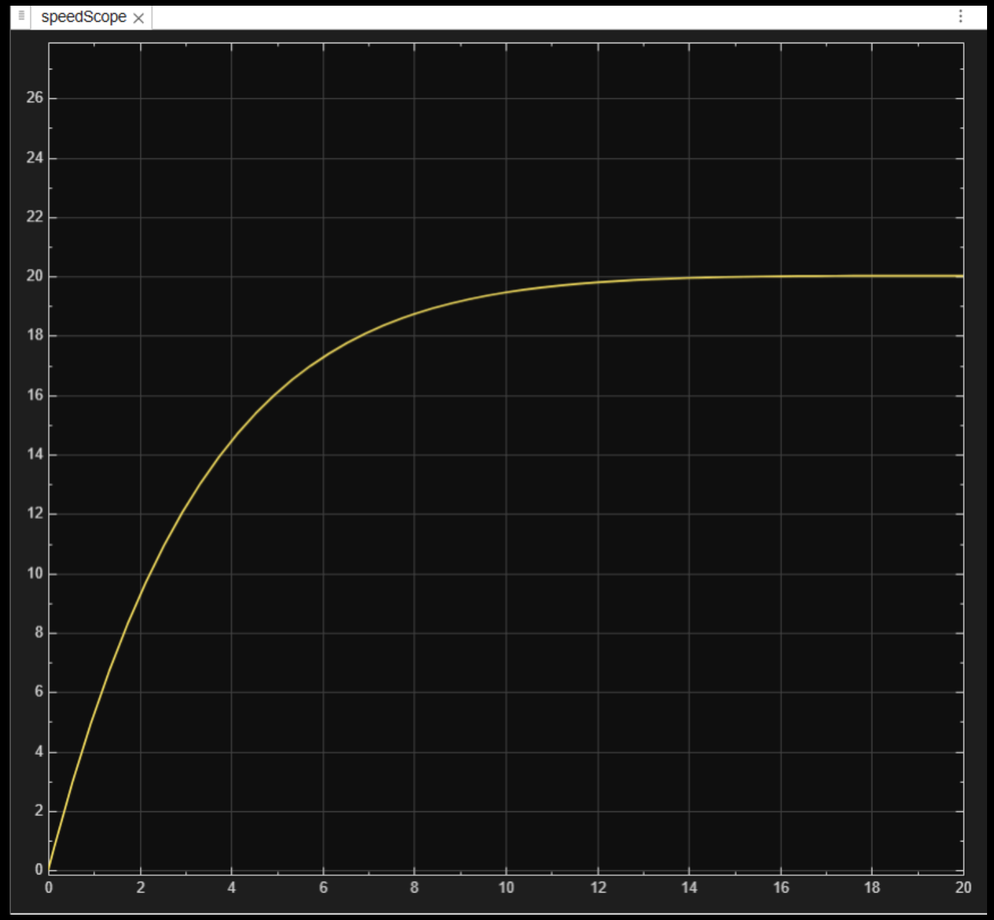


OS = (19.9704 - 20) / 20 × 100 = 0%

Settling Time: ≈ 17 s

SSE: 19.9704 - 20 = -0.03 m/s

CASE 4 – Constant Slope, Kp = 2, Ki = 0.5, Slope = 10°

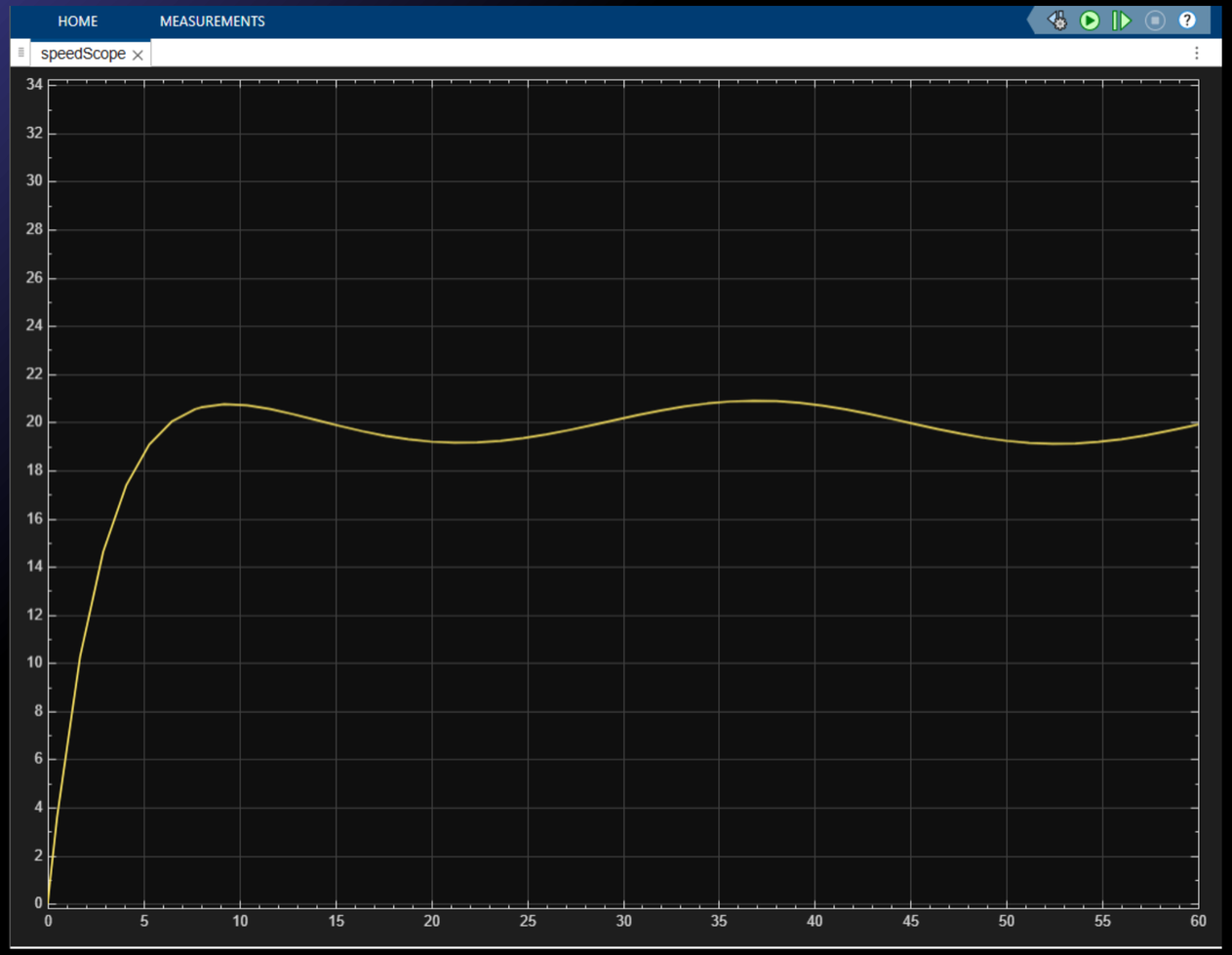


OS = (20.0206 - 20) / 20 × 100 = 0.1%

Settling Time:≈ 10 s

SSE: 20.0206 - 20 = +0.02 m/s

CASE 5 — variabkle Slope, Kp = 2, Ki = 0.5,



OS = (20.7 - 20) / 20 × 100 = 3.5%

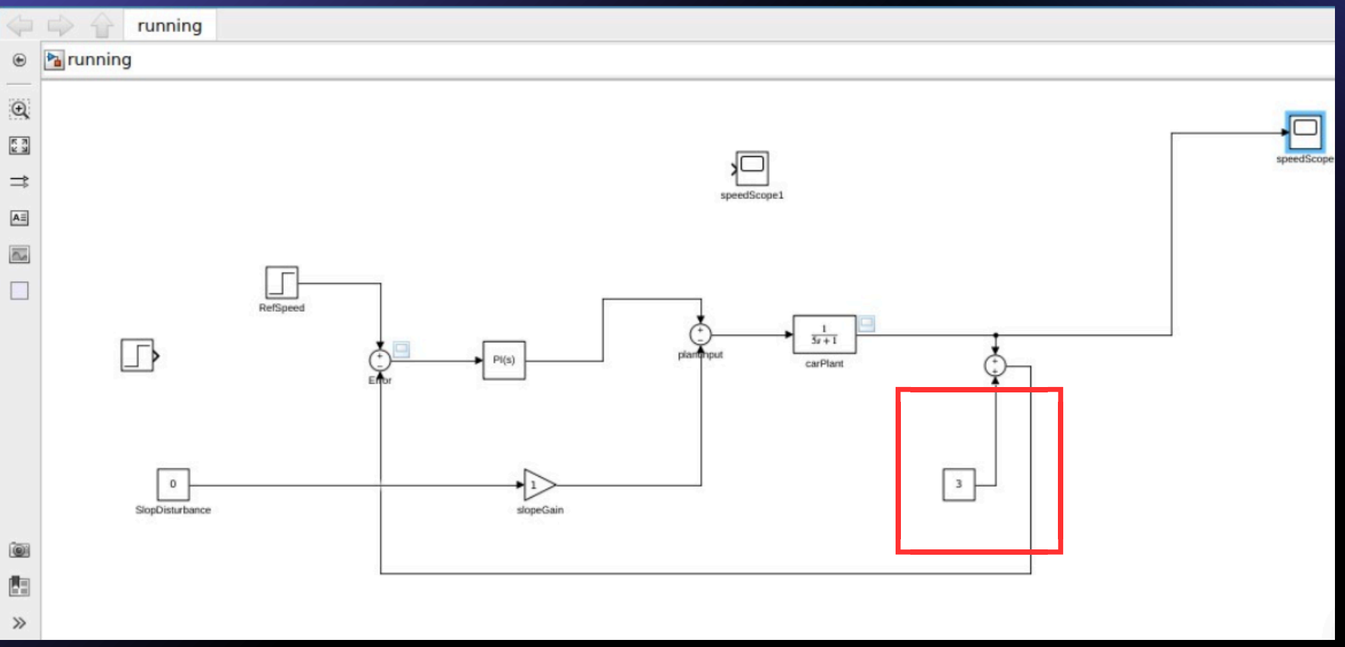
No fixed settling — periodic steady-state after ≈ 15 s

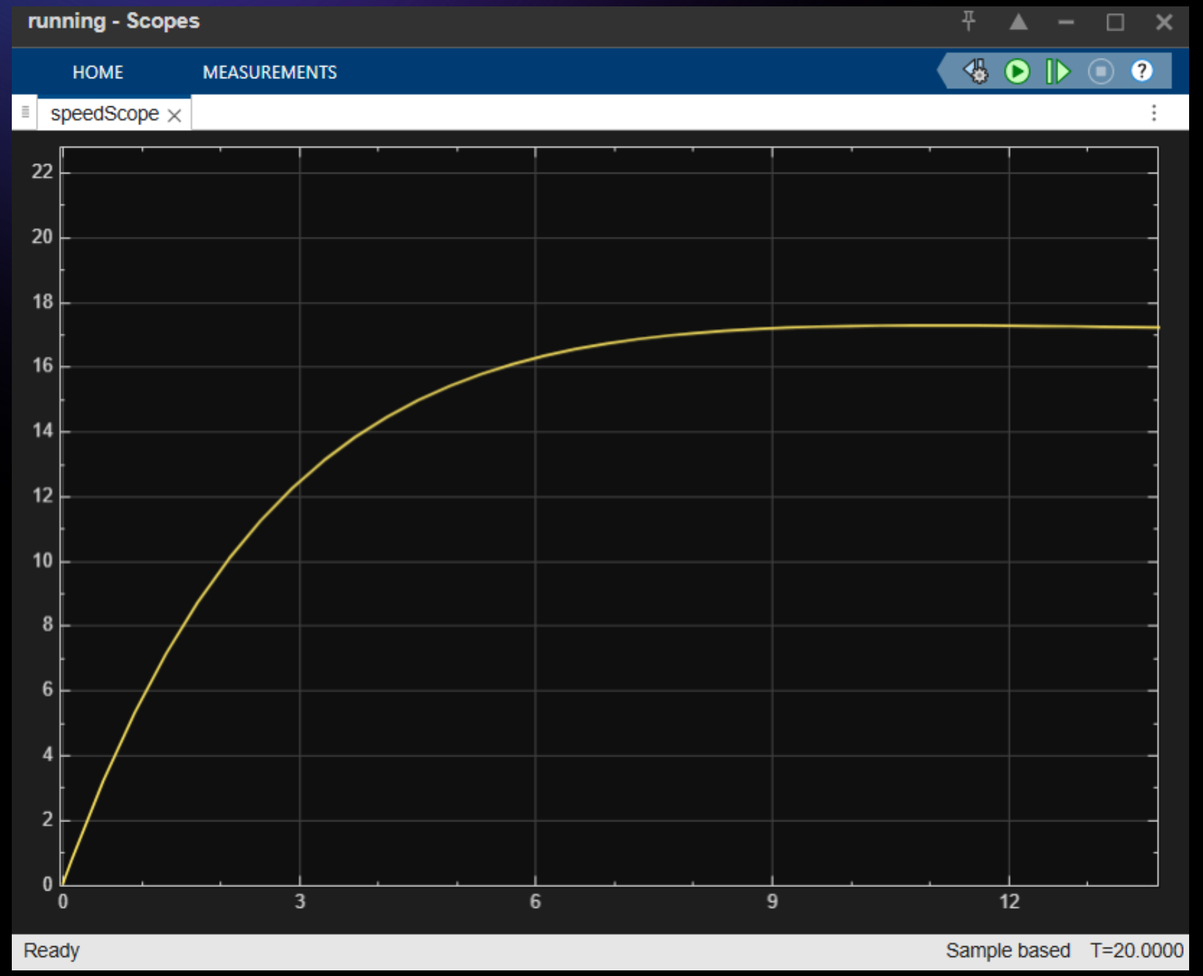
sse = 19.8375 - 20 = -0.1625 m/s

Comparasion table for the cases and overshoot , sittling time , SSE:

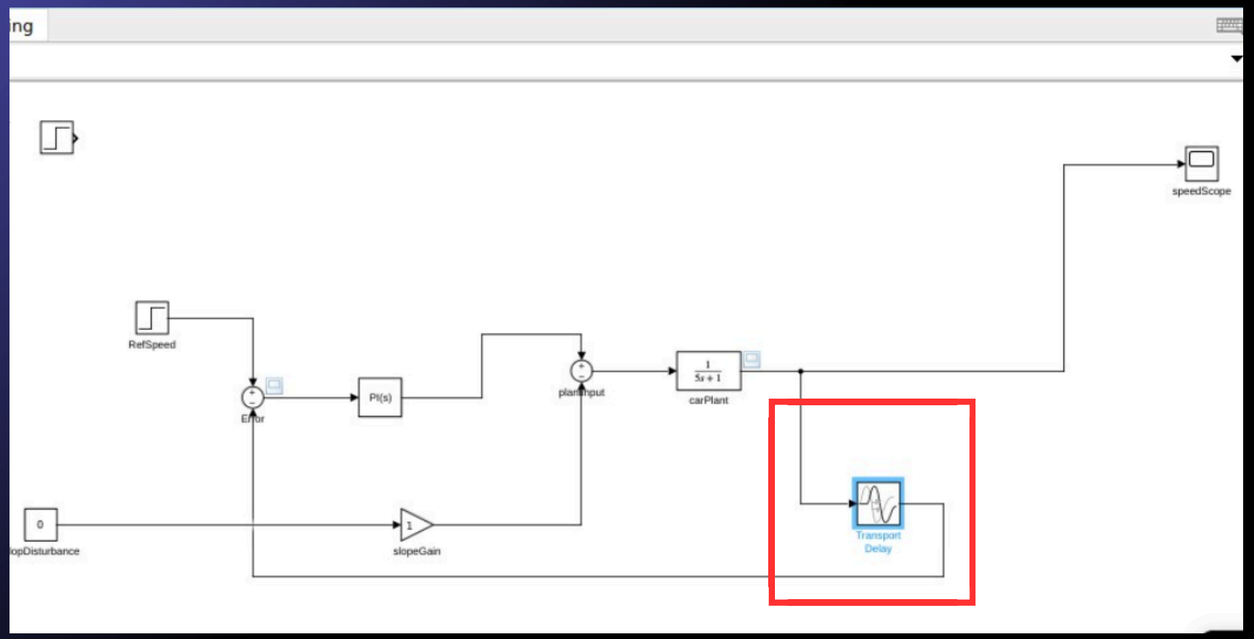
Case	Kp	Ki	Slope Condition	Peak Speed (m/s)	Overshoot (%)	Settling Time (s)	SSE (m/s)
1	2	0.5	Flat (θ = 0)	20.3	≈ 1.5%	≈ 7 s	0.0657
2	10	4.5	Flat (θ = 0), aggressive gains	21.1805	≈ 6%	≈ 7 s	0.001
3	0.8	0.2	Flat (θ = 0), low gains	19.9704	0%	≈ 17 s	−0.03
4	2	0.5	Constant slope (θ = 10°)	20.0206	≈ 0.1%	≈ 10 s	0.02
5	2	0.5	variable slope, , gain −10	20.7	≈ 3.5%	Periodic oscillation (steady after ~15 s)	−0.1625

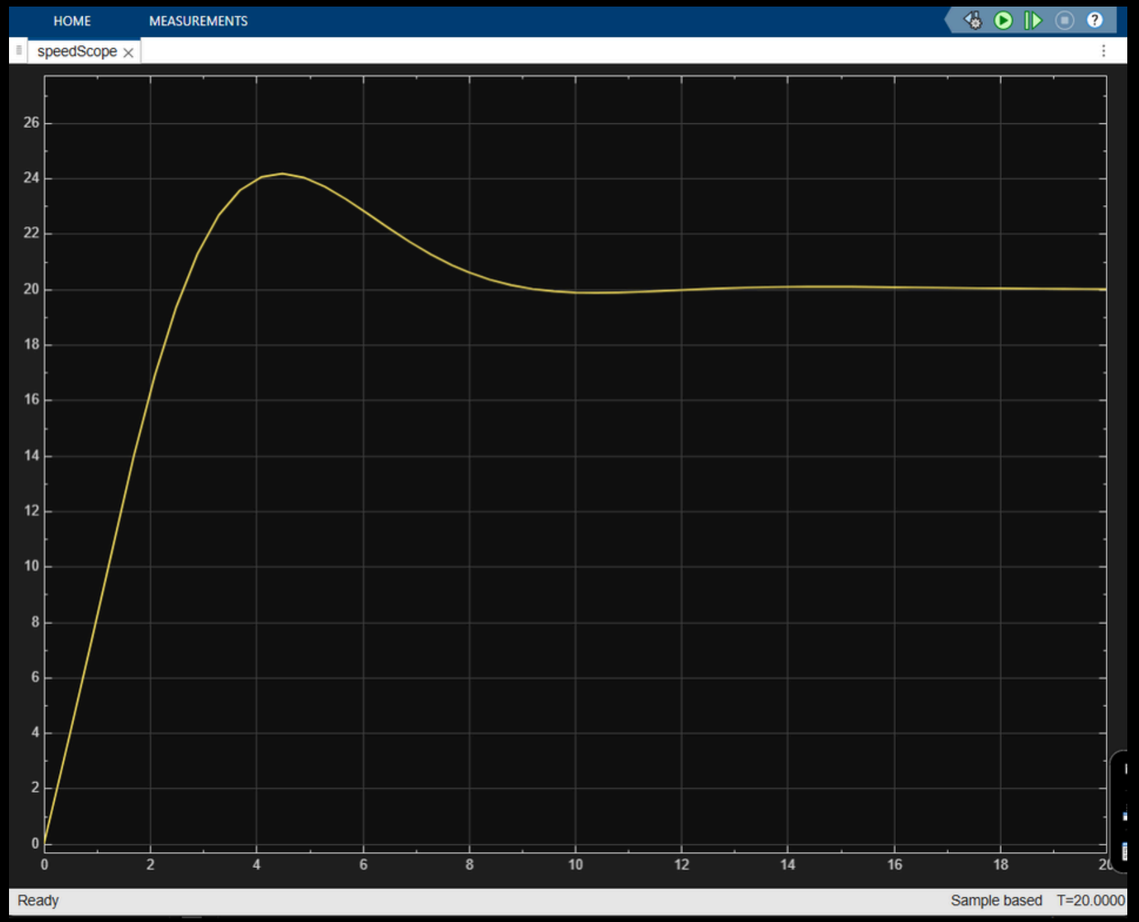
Task 5 : Security Attacks
Attack 1 — Sensor Spoofing
Spoofing Attack (+3 m/s Bias)





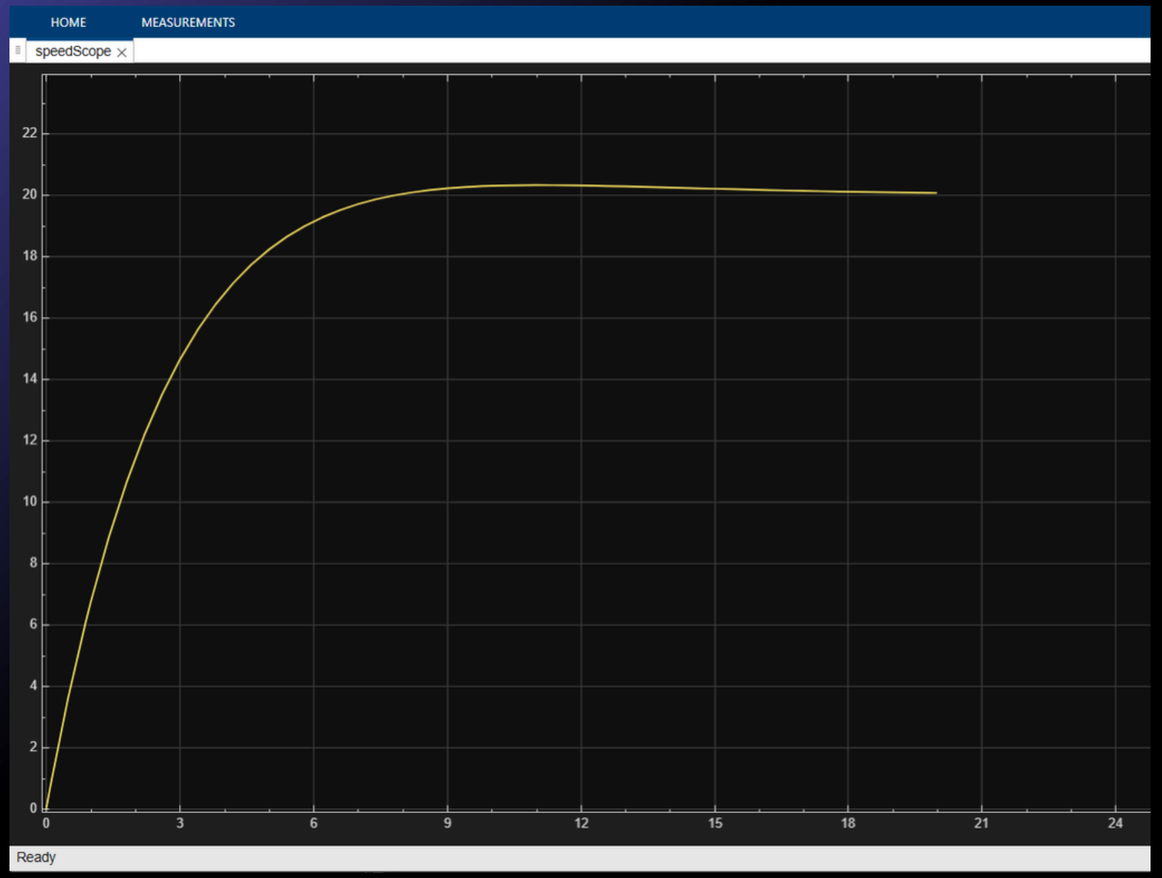
Attack 2 — DoS (Delay) Attack (1.5 s Feedback Delay)



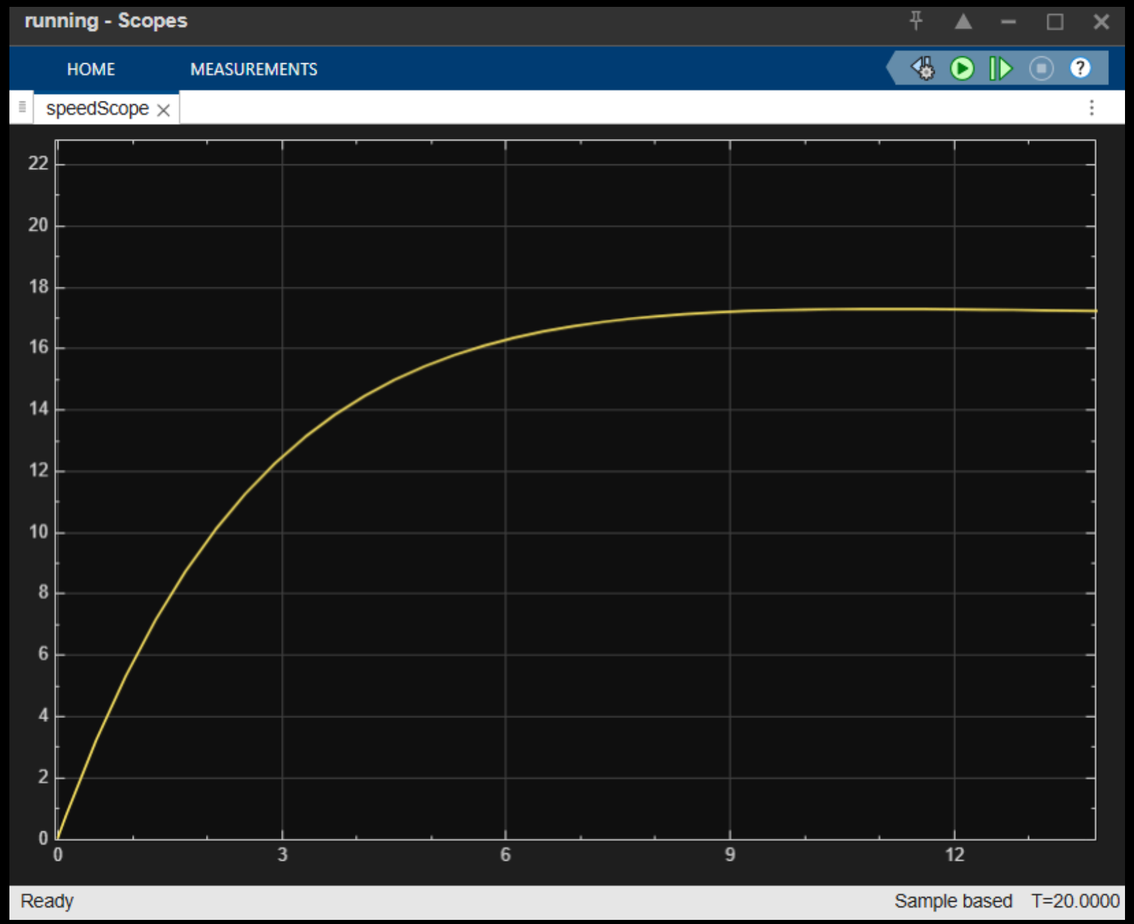


Normal vs attacked PID responses

Normal

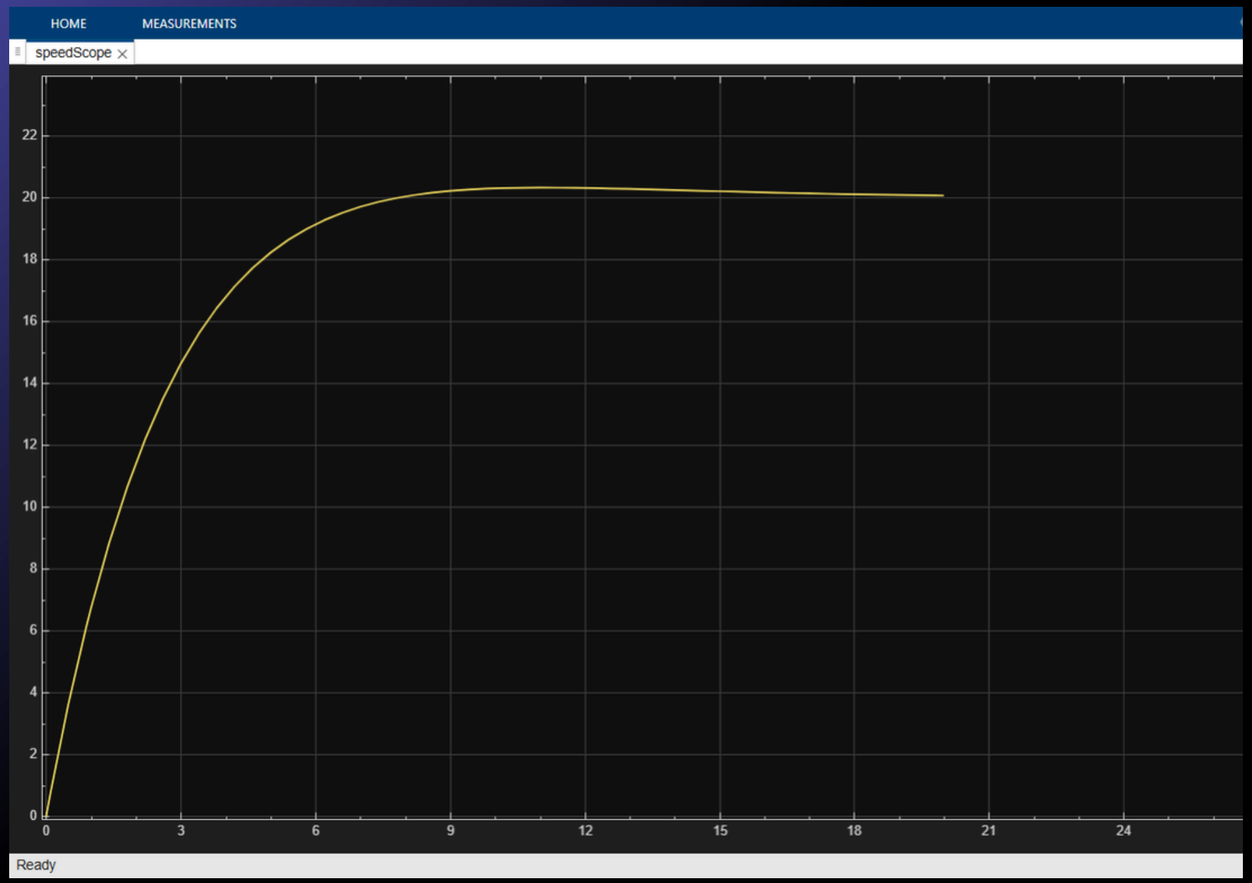


spoofing attack +3

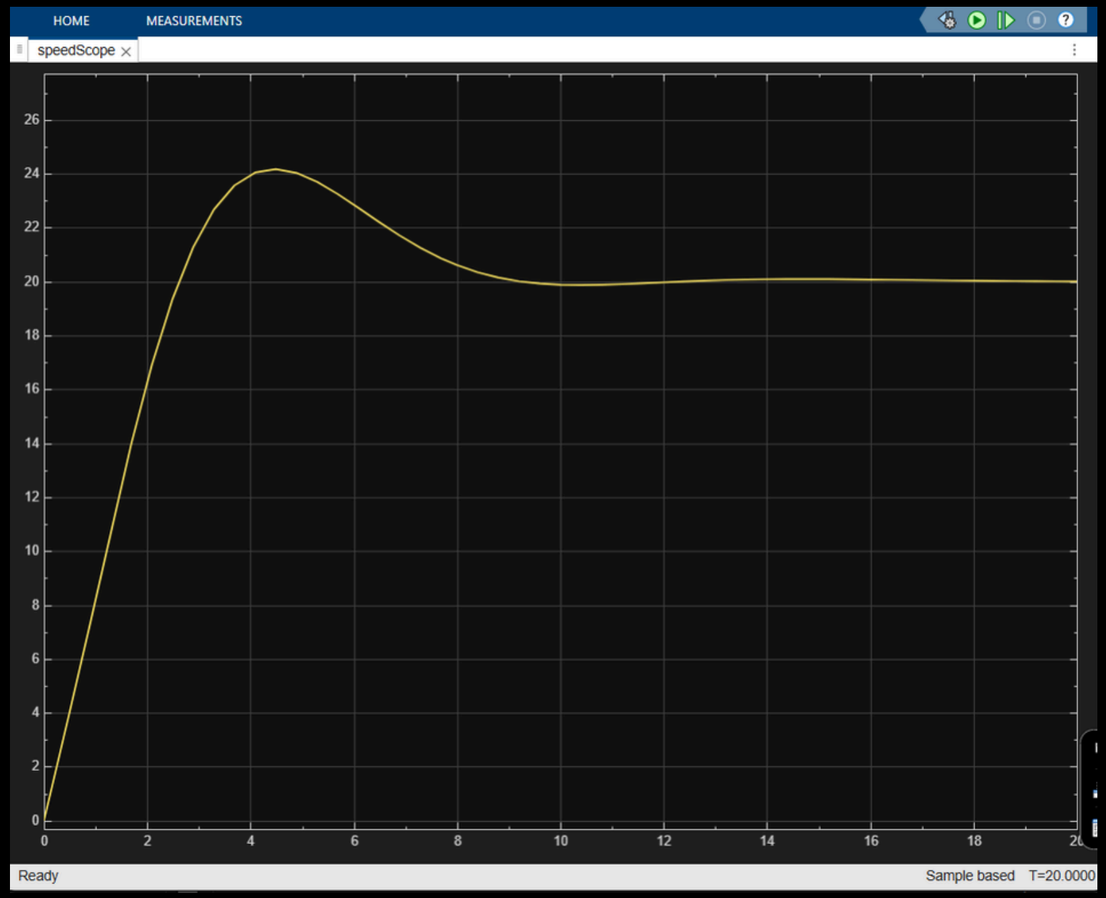


Normal vs attacked PID responses

Normal



Dos attack



DoS Delay (1.5 s):

outdated feedback → ~20% overshoot

longer settling time (~13 s)

System becomes unstable

Overshoot: 1.5% → 20%

Settling time: 7 s → 13 s

Sensor Spoofing (+3 m/s):

controller thinks the car is faster → speed drops to ~17 m/s

data was incorrect

SSE: – 3 m/s large steady-state error

Settling time: ~12 s

Controller thinks car is faster → reduces throttle

Mitigation

Spoofing

- **Communication authentication:**

Make sure the messages inside the car network are real and not changed by an attacker.

- **Redundant sensors :**

Use more than one sensor. If one sensor lies or fails, the other sensors confirm the truth

DoS Delay Attack

- **Timestamp validation :**

checks if incoming data is too old.
If the feedback or sensor data is delayed (DoS attack), the system discards it and uses only fresh data.

- **Redundant communication :**

channels provide backup data paths.
If one network is attacked or delayed, the second network continues delivering fresh data.