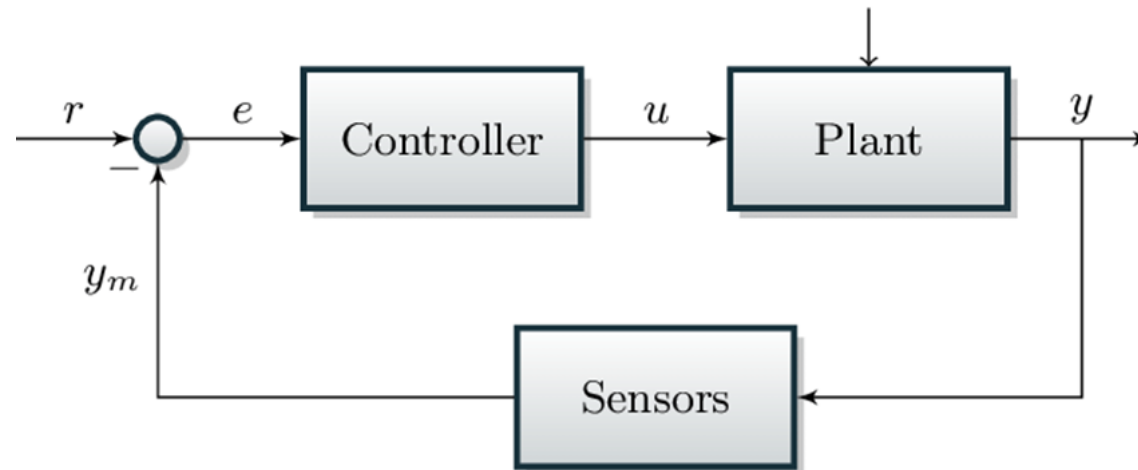


# L6 Control

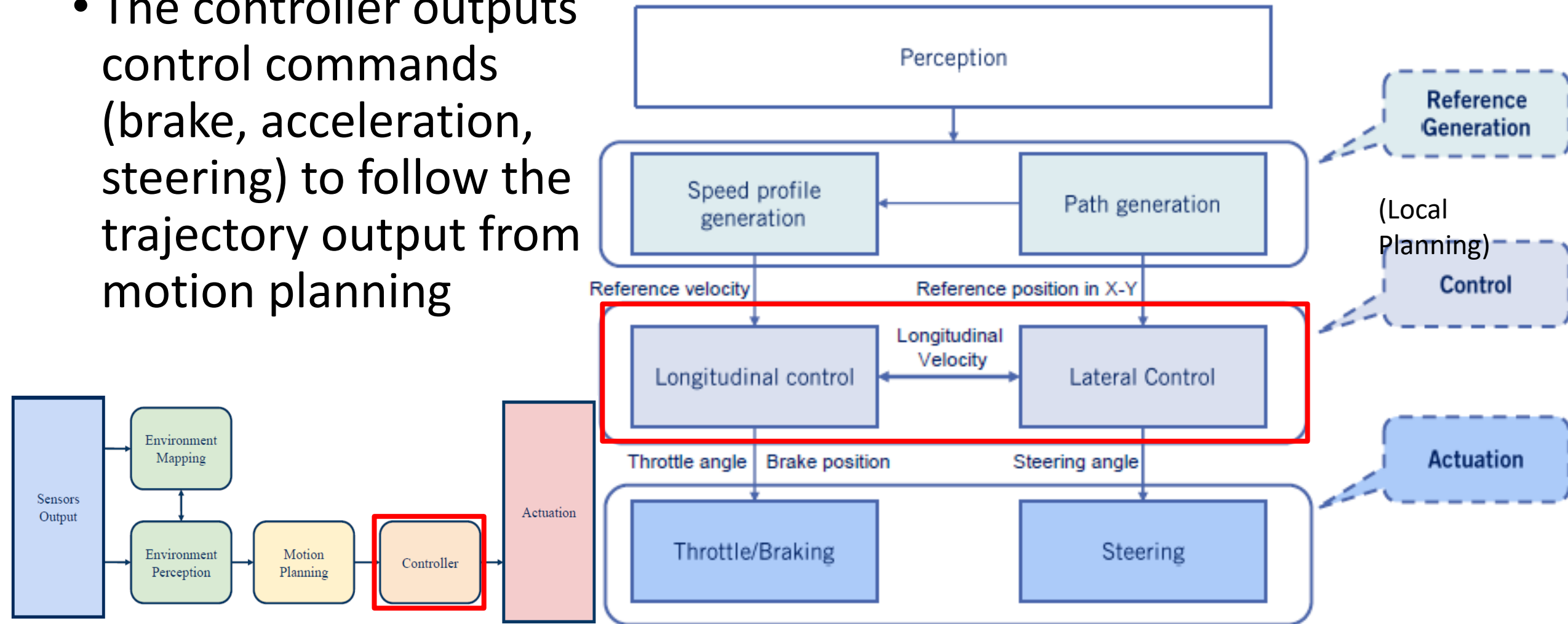


Zonghua Gu, Umeå University

Nov. 2023

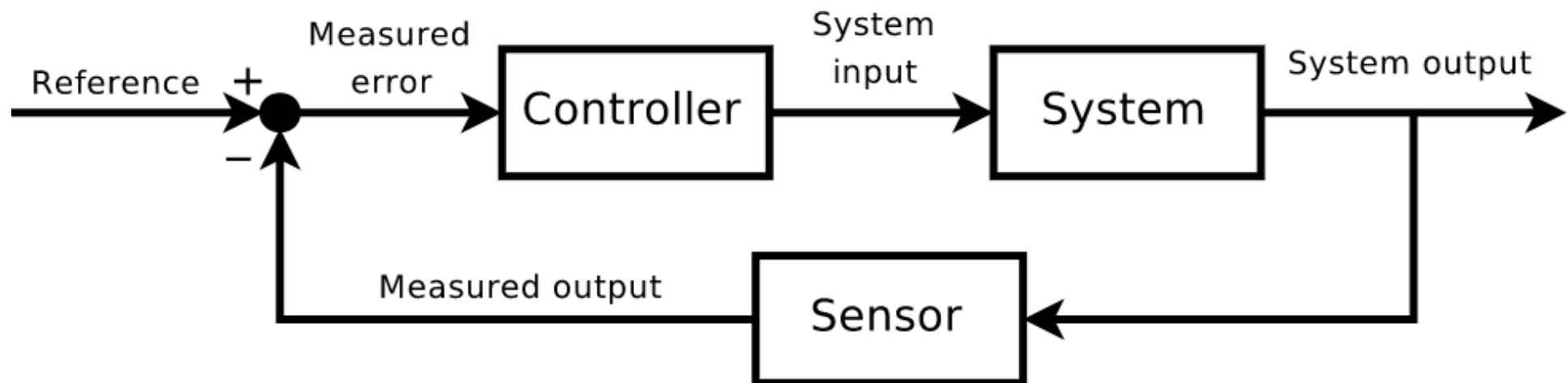
# Control in the AD Pipeline

- The controller outputs control commands (brake, acceleration, steering) to follow the trajectory output from motion planning



# Feedback Control Problem

- Given a system and a reference signal, find a control law such that the closed loop system is stable and follows the reference signal
- The most common control algorithms in automotive systems are Proportional–Integral–Derivative (PID) and Model Predictive Control (MPC)



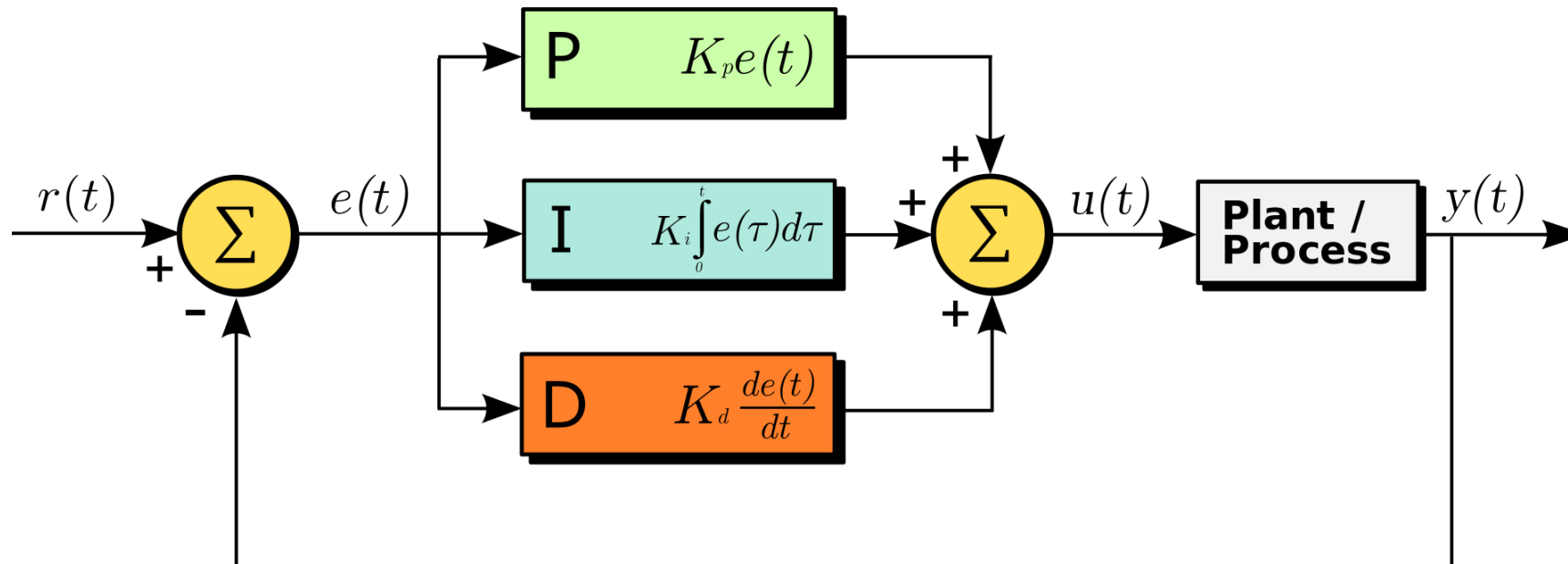
# Outline

---

- PID Control
- MPC Control
- Kinematic bicycle model
- Twiddle() for PID control tuning

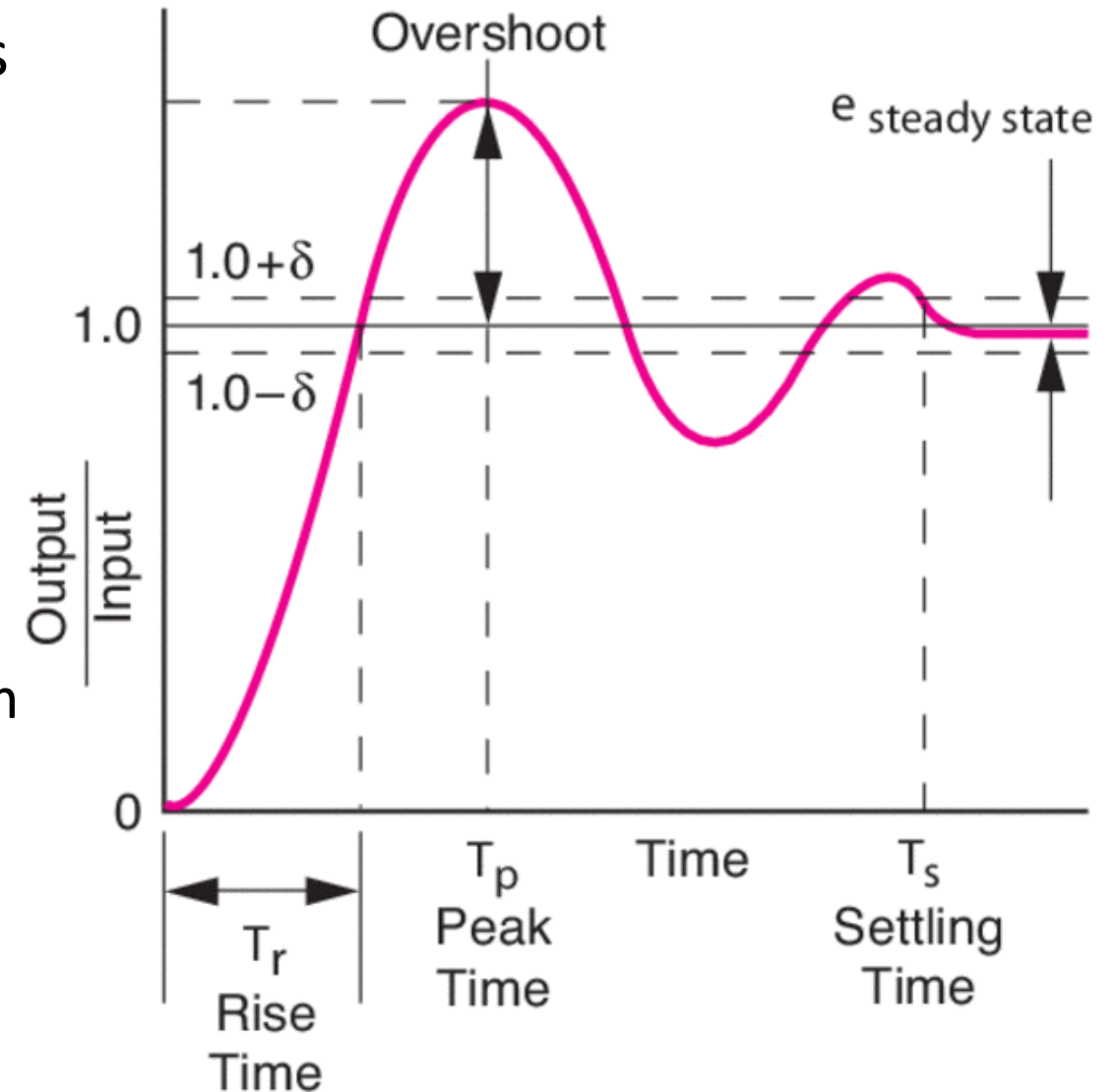
# PID Control

- Tracking Error:  $e(t) = r(t) - y(t)$
- Control input:  $u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t)$
- Ref: Controlling Self Driving Cars
  - <https://www.youtube.com/watch?v=4Y7zG48uHRo>



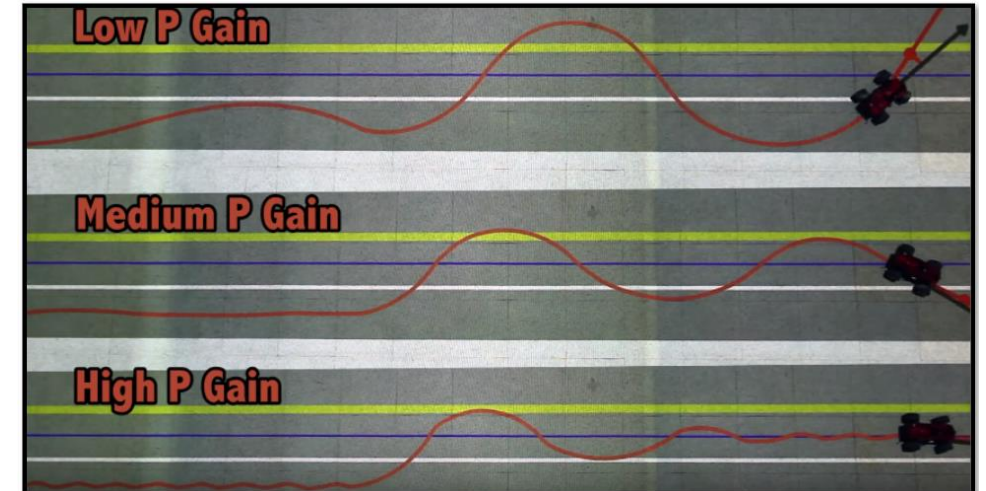
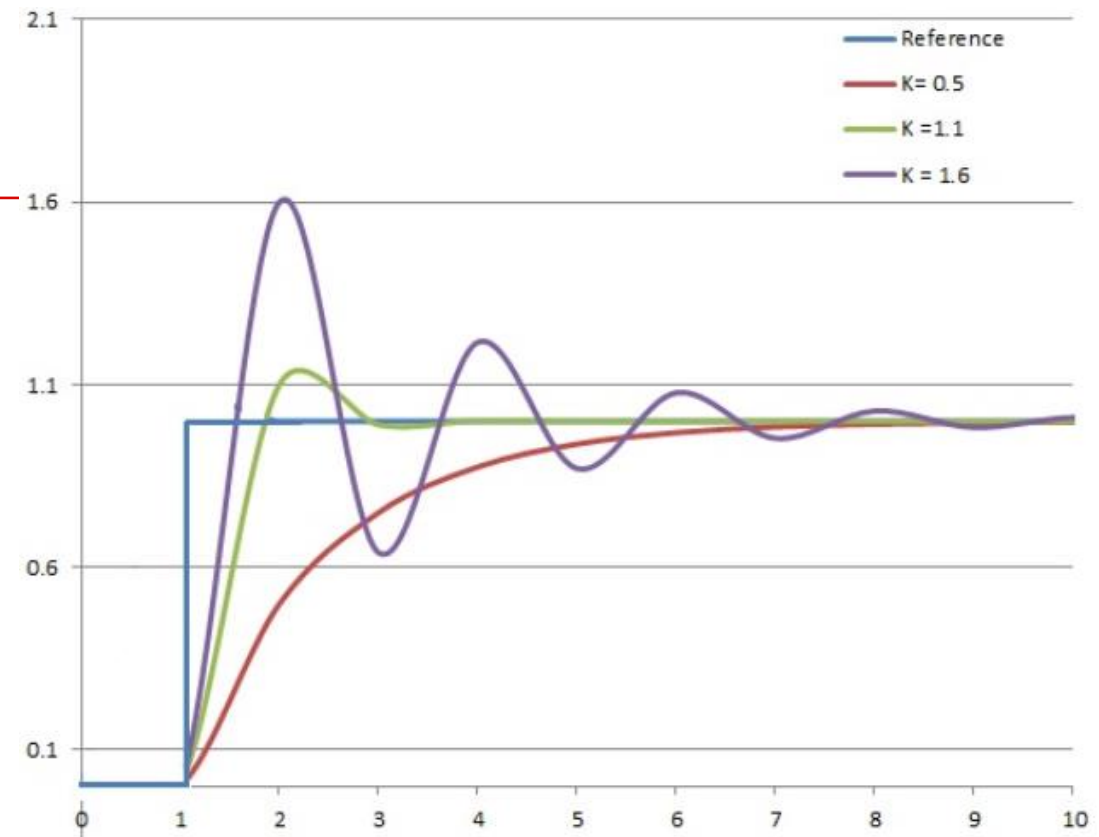
# Step Response Performance Metrics

- The control input (reference input) jumps from 0 to a reference value (e.g., 1.0) at time 0, and the controller aims to track it closely
  - Rise time: the time it takes the transient response to move to  $1.0 - \delta$  of the steady state response
  - Maximum overshoot: the amount (or percentage) by which the maximum value of the transient response exceeds the steady state value
  - Peak time: the time at which the maximum overshoot occurs
  - Settling time: the time after which the output is within a specified band around the steady state value  $[1.0 - \delta, 1.0 + \delta]$



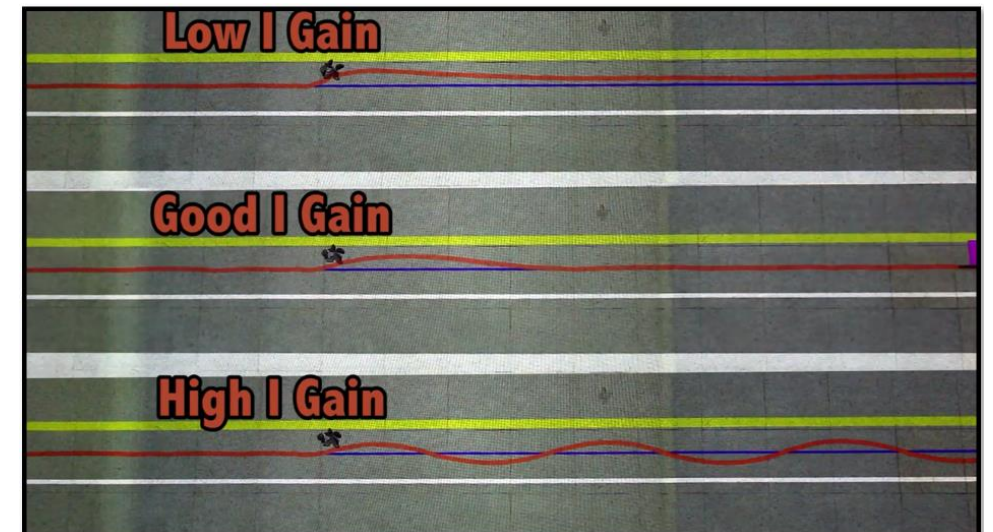
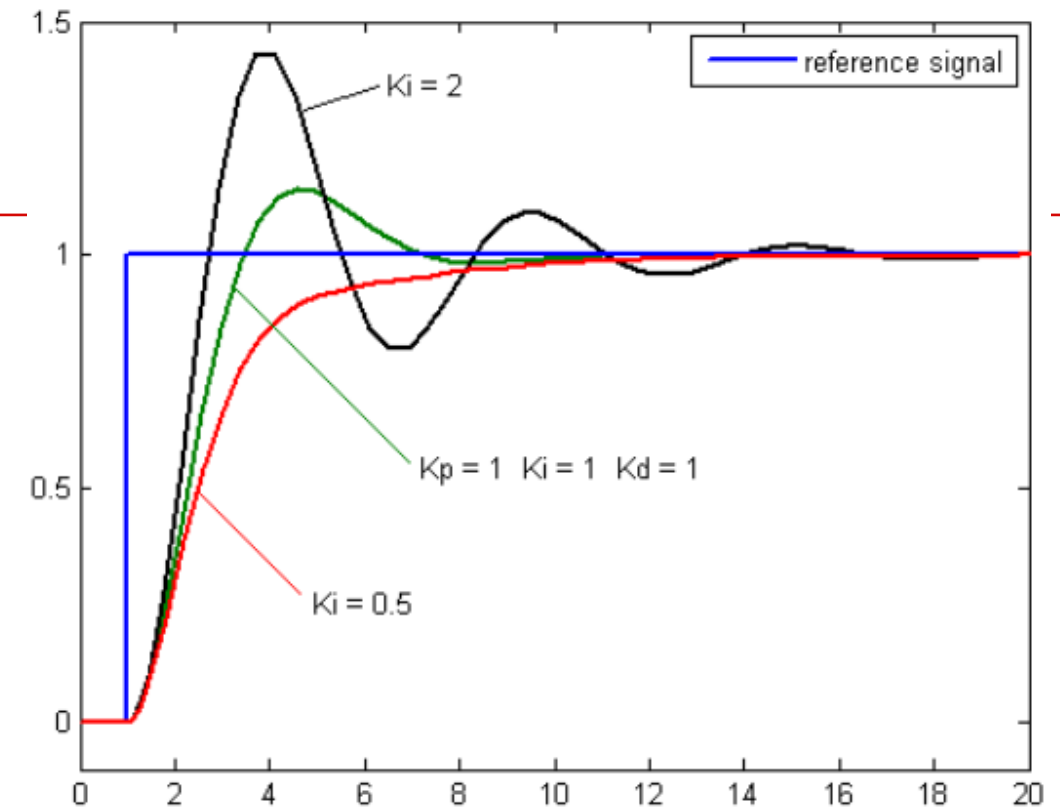
# Proportional Term $K_p$

- $u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t)$
- $u(t)$  proportional to error  $e(t)$  with factor  $K_p$
- Increasing  $K_p$  leads to:
  - Faster response
  - Bigger overshoot, oscillations
    - System may become unstable
  - Smaller but non-zero steady state error



# Integral Term $K_i$

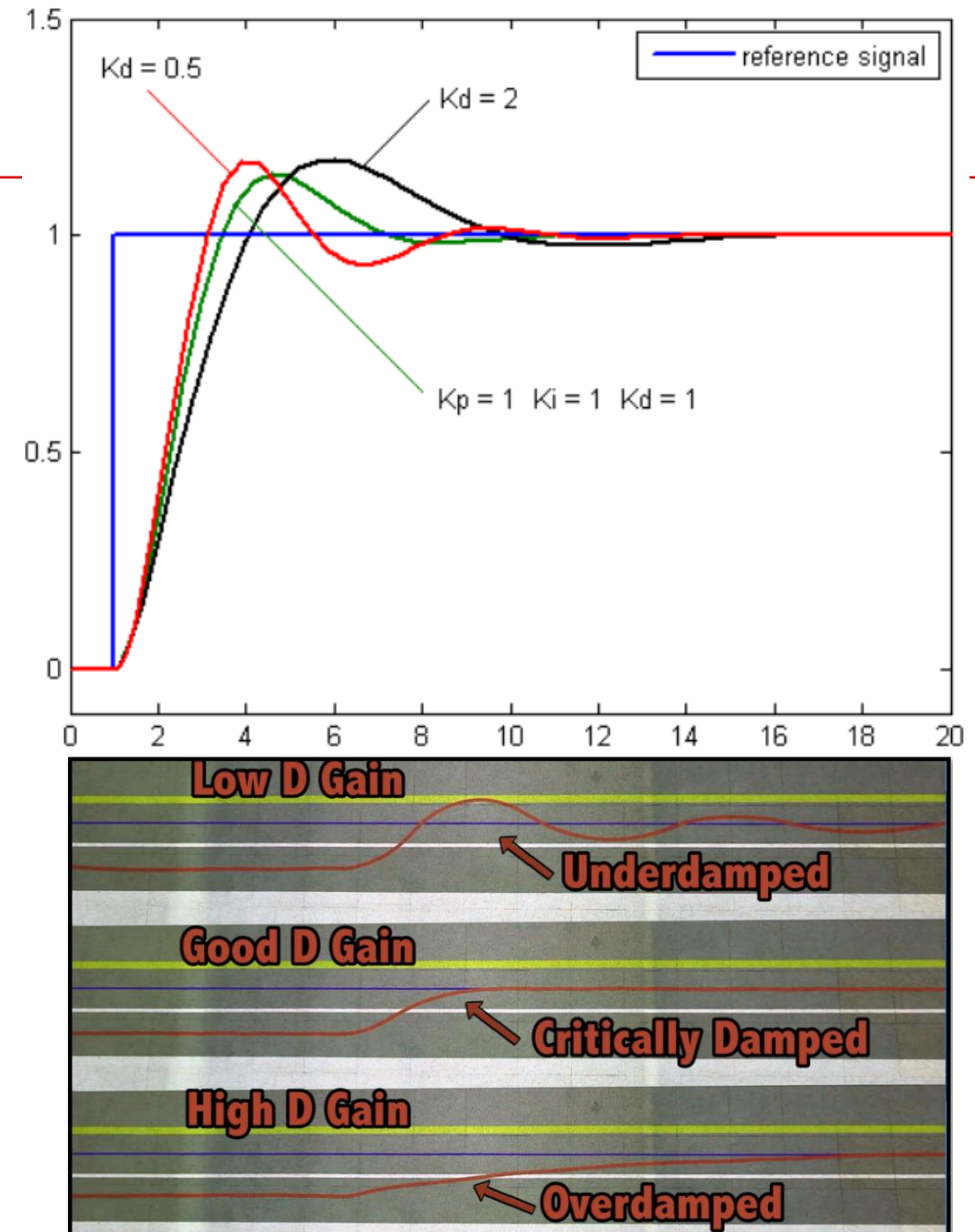
- $u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t)$
- $K_i$  takes into account history of tracking error, and eliminates steady state error
- Increasing  $K_i$  leads to:
  - Increased overshoot
  - More robust to disturbances





# Derivative Term $K_d$

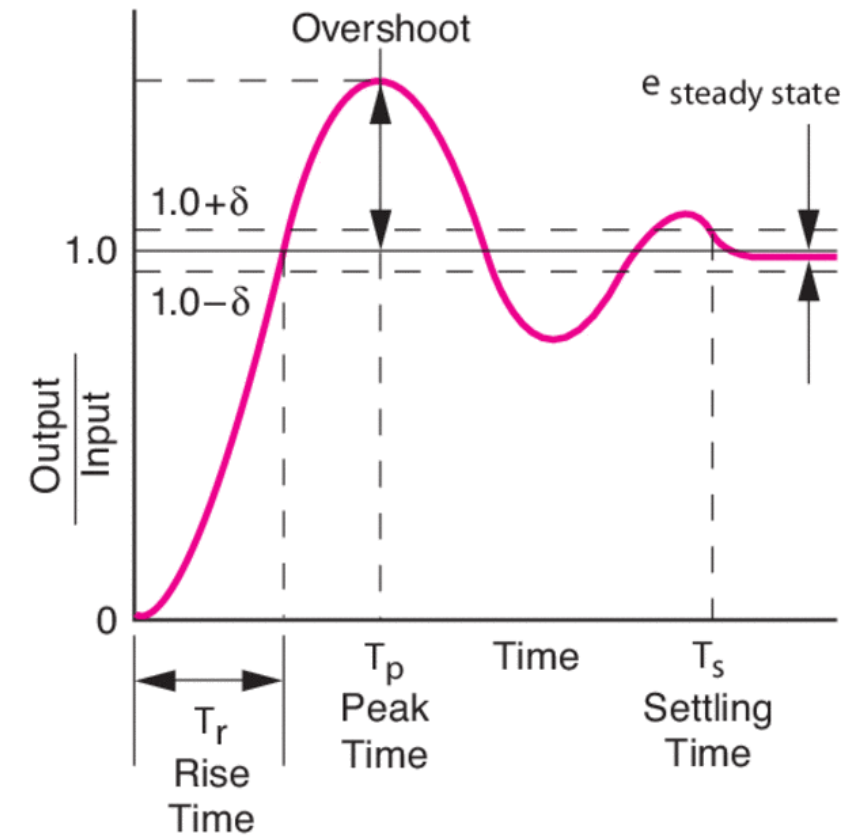
- $u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t)$
- $u(t)$  proportional to error derivative  $\dot{e}(t)$  by factor  $K_d$
- Increasing  $K_d$  leads to:
  - Reduced overshoot
  - Faster response
  - Little effect on steady state
  - More sensitive to measurement noise



# Effects of PID Gains

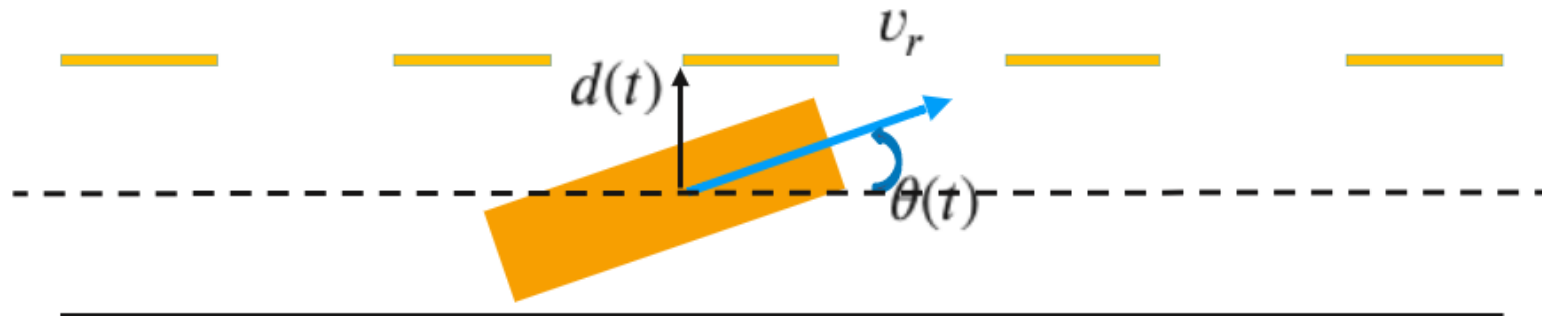
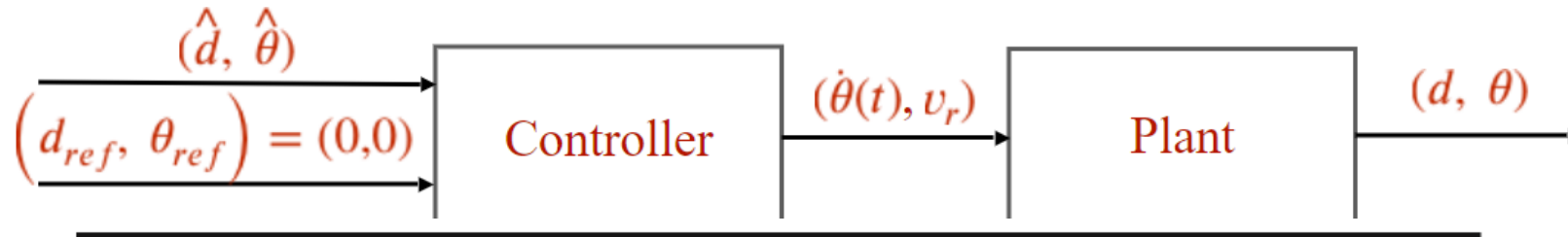
- $u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \dot{e}(t)$

Closed-Loop Response	Rise Time	Overshoot	Settling Time	Steady State Error
Increase $K_p$	Decrease	Increase	Small increase	Decrease
Increase $K_i$	Small decrease	Increase	Increase	Large decrease
Increase $K_d$	Small decrease	Decrease	Decrease	Minor change



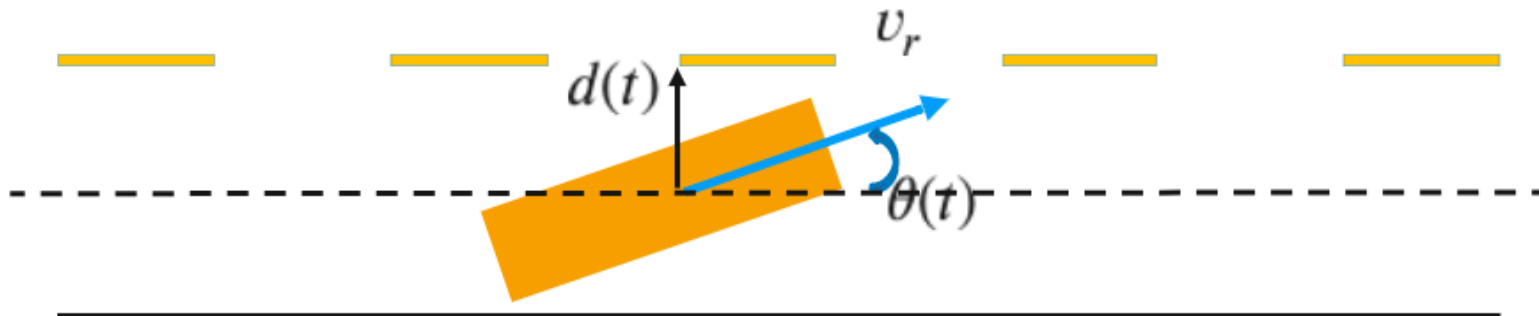
# Example: Vehicle Lateral Control

- State  $(d, \theta)$ 
  - $d$ : distance to center of lane;  $\theta$ : heading angle.
- Reference trajectory:  $(d_{ref}, \theta_{ref}) = (0, 0)$ 
  - The vehicle travels straight ahead at center of lane.
- Vehicle has constant speed  $v_r$ , so the only control input is  $u(t) = \dot{\theta}(t)$ , the angular velocity.
- Assume perfect sensor state estimation:  $(\hat{d}, \hat{\theta}) = (d, \theta)$ 
  - Ref. Control Algorithms for Autonomous Vehicles [https://www.icloud.com/keynote/035Qivaw\\_FXYD70xCbD5R9IDA#control\\_AV](https://www.icloud.com/keynote/035Qivaw_FXYD70xCbD5R9IDA#control_AV)



# System and P Control Modeling

- Linear system dynamics:
  - $\begin{bmatrix} \dot{d}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} 0 & v_r \\ 0 & 0 \end{bmatrix} \begin{bmatrix} d(t) \\ \theta(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t)$
  - $\dot{d}(t) = v_r \sin \theta(t) \approx v_r \theta(t)$  assuming  $\theta(t)$  is small ( linearized kinematic model)
  - $\dot{\theta}(t) = u(t)$
- P Controller:
  - $u(t) = [K_d \quad K_\theta] \begin{bmatrix} d(t) \\ \theta(t) \end{bmatrix}$
- Closed-loop dynamics is obtained by plugging  $u(t)$  into system dynamics:
  - $\begin{bmatrix} \dot{d}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} 0 & v_r \\ K_d & K_\theta \end{bmatrix} \begin{bmatrix} d(t) \\ \theta(t) \end{bmatrix}$
- Ref trajectory:  $\begin{bmatrix} d_{ref} \\ \theta_{ref} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- Error signal:
  - $\begin{bmatrix} e_d(t) \\ e_\theta(t) \end{bmatrix} = \begin{bmatrix} d_{ref} \\ \theta_{ref} \end{bmatrix} - \begin{bmatrix} d(t) \\ \theta(t) \end{bmatrix} = - \begin{bmatrix} d(t) \\ \theta(t) \end{bmatrix}$
- Error dynamics:
  - $\begin{bmatrix} \dot{e}_d(t) \\ \dot{e}_\theta(t) \end{bmatrix} = - \begin{bmatrix} \dot{d}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} 0 & -v_r \\ -K_d & -K_\theta \end{bmatrix} \begin{bmatrix} d(t) \\ \theta(t) \end{bmatrix}$
- Control objective: design  $K_d, K_\theta$  to drive error  $\begin{bmatrix} e_d(t) \\ e_\theta(t) \end{bmatrix}$  to  $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$



# P Control Design with Pole Placement (not covered in this course)

- Closed-loop poles:

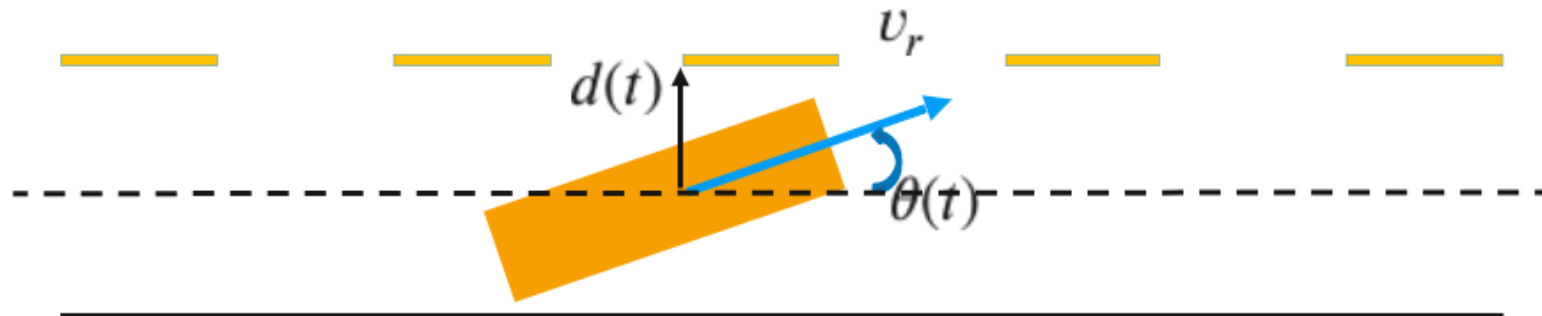
- $0 = \det(\lambda I - \begin{bmatrix} 0 & -v_r \\ -K_d & -K_\theta \end{bmatrix}) = \lambda^2 + K_\theta \lambda - v_r K_d$

- Solution  $\lambda_{1,2} = -\frac{K_\theta}{2} \pm \frac{1}{2} \sqrt{K_\theta^2 + 4v_r K_d}$

- Critically-damped dynamics (repeated real roots):

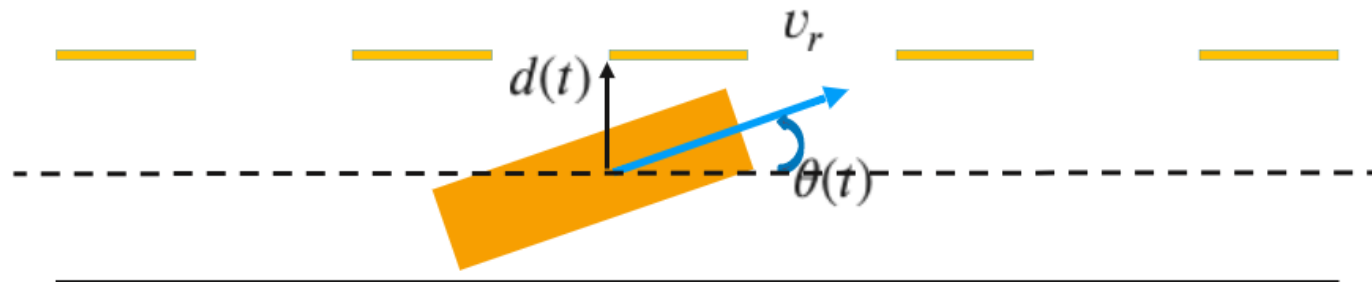
- $\sqrt{K_\theta^2 + 4v_r K_d} = 0 \Rightarrow K_d = -\frac{K_\theta^2}{4v_r}$

- $K_\theta$  chosen empirically



# P Control Design Cont'd

- Linear systems dynamics relies on the small angle approximation  $\sin \theta(t) \approx \theta(t)$ , assuming  $|\theta(t)| < \theta_{th}, \forall d(0)$
- Closed-loop dynamics  $\dot{\theta}(t) = K_d d(t) + K_\theta \theta(t)$ 
  - Set  $\dot{\theta}(t) = -\frac{K_\theta^2}{4v_r} \text{sat}(d(t), d_{th}) + K_\theta \theta(t)$ ,
  - where:  $\text{sat}(d(t), d_{th}) = \begin{cases} -d_{th} & \text{if } d(t) < -d_{th} \\ d(t) & \text{if } d(t) \in [-d_{th}, d_{th}] \\ d_{th} & \text{if } d(t) > d_{th} \end{cases}$
- Empirical param settings:  $\theta_{th} = \frac{\pi}{6}, d_{th} = \left| \frac{K_\theta \theta_{th}}{K_d} \right|$



# Simplifying Assumptions

---

- Did not consider:
  - Estimation uncertainty due to measurement noise
    - We assume perfect state estimation  $(\hat{d}, \hat{\theta}) = (d, \theta)$
  - Estimation latency:
    - Time from measurements  $(d, \theta)$  to state estimation  $(\hat{d}, \hat{\theta})$  availability to the controller
  - Constraints (e.g., actuator limits)
    - Need to impose a maximum curvature radius to simulate a real car.
  - Discrete time (multi-rate), non-uniform sampling:
    - Our controller is continuous time, but the actual implementation runs in discrete time.
    - Sampling rate of the estimator (slower) may be different than that of actuation (faster), or may be variable

# Outline

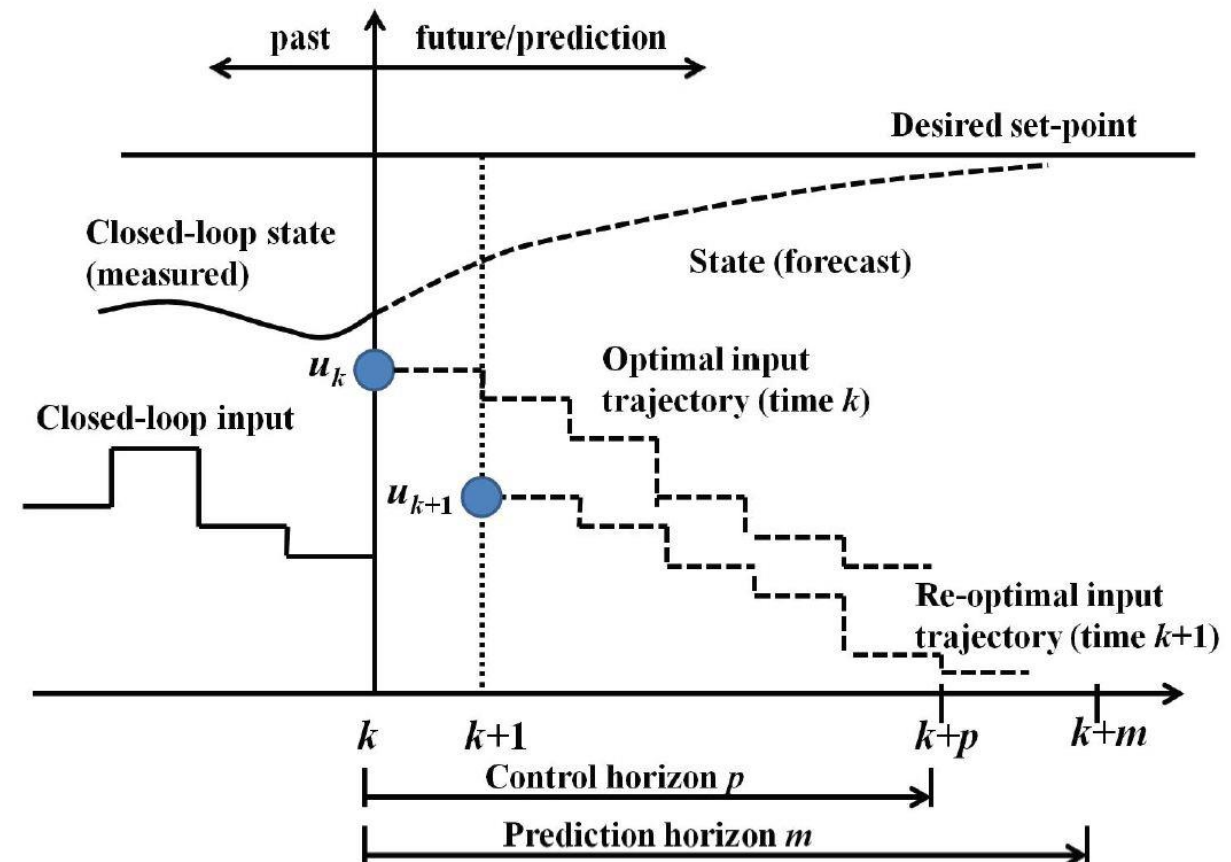
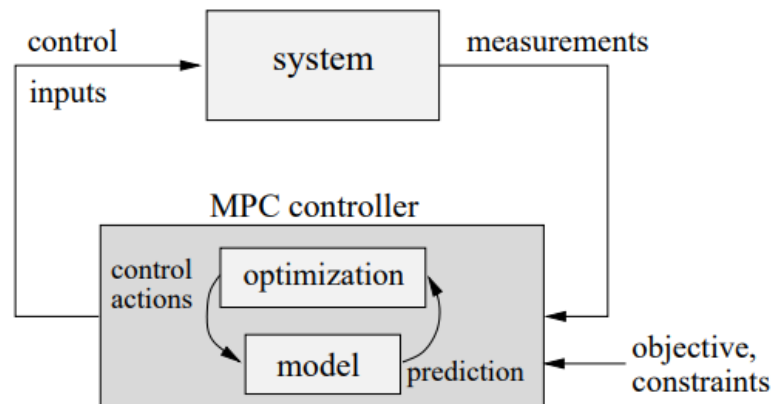
---

- PID Control
- MPC Control
- Kinematic bicycle model
- Twiddle() for PID control tuning

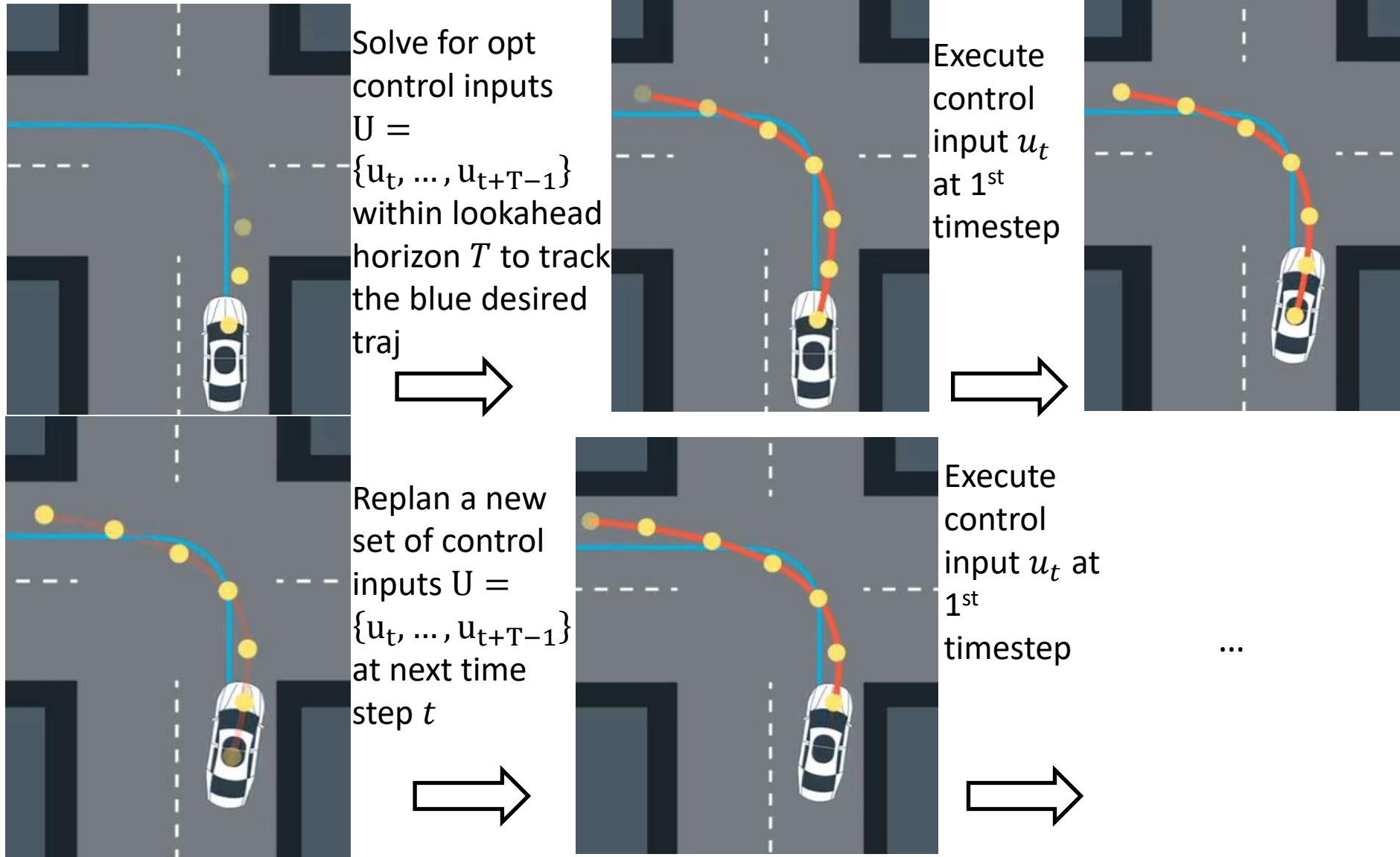


# MPC (Model-Predictive Control)

- Also called Receding Horizon Control
- Choose prediction horizon  $m$  and control horizon  $p$
- At each time step  $k$ :
  - Set initial state to predicted state  $x[k]$
  - Solve a constrained optimization problem over lookahead window  $[k, k + m]$ , to get a sequence of control inputs  $u$ , while in the time interval  $[k - 1, k]$
  - Apply 1<sup>st</sup> control command  $u[k]$  at time step  $k$
- Control horizon  $p$  and prediction horizon  $m$  may be different, but often the same
  - denoted as lookahead horizon  $T$  in the next slides (current time is denoted as  $t$ )



# MPC Example



# MPC Illustration Con't

- **Yellow**: reference trajectory from planner
- **Green**: trajectory from running control inputs  $u[0, \dots, T - 1]$  computed by MPC based on system model for lookahead horizon  $T$



<https://medium.com/@david010/vehicle-mpc-controller-33ae813cf3be>

# Linear vs. Nonlinear MPC

- Linear MPC (no constraints)

- $\min_{U=\{u_t, \dots, u_{t+T-1}\}} J(x(t), U) = x_{t+T}^T Q_f x_{t+T} + \sum_{j=t}^{t+T-1} (x_j^T Q x_j + u_j^T R u_j)$
- s.t. for  $t \leq j \leq t + T - 1$
- $x_{j+1} = Ax_j + Bu_j$
- $x_j$  is the difference between actual state and ref state (assumed to be 0 here), which should be minimized with the term  $x_{t+T}^T Q_f x_{t+T}$
- $u_j$  is control input, which should be minimized with the term  $u_j^T R u_j$  (to reduce control effort)
- Relative magnitudes of  $Q$  and  $R$  encode relative importance of the two objectives
- Can be solved analytically at each time step  $u_t = -K_t x_t$

- Nonlinear MPC (with constraints)

- $\min_{U=\{u_t, \dots, u_{t+T-1}\}} J(x(t), U) = \sum_{j=t}^{t+T} C(x_j, u_j)$
- s.t. for  $t \leq j \leq t + T - 1$
- $x_{j+1} = f(x_j, u_j)$
- $x_{min} \leq x_{j+1} \leq x_{max}$
- $u_{min} \leq u_j \leq u_{max}$
- $g(x_j, u_j) \leq 0$
- $h(x_j, u_j) = 0$
- Both objective  $J(x(t), U)$  and system dynamics  $f(x_j, u_j)$  may be nonlinear
- Cannot be solved analytically; must solve numerically with optimizers, e.g., CasADi

- (Note: in  $x_{t+T}^T$ , superscript  $T$  denotes “vector transpose”; subscript  $T$  denotes “lookahead horizon”)

# MPC Pros and Cons

---

- Pros

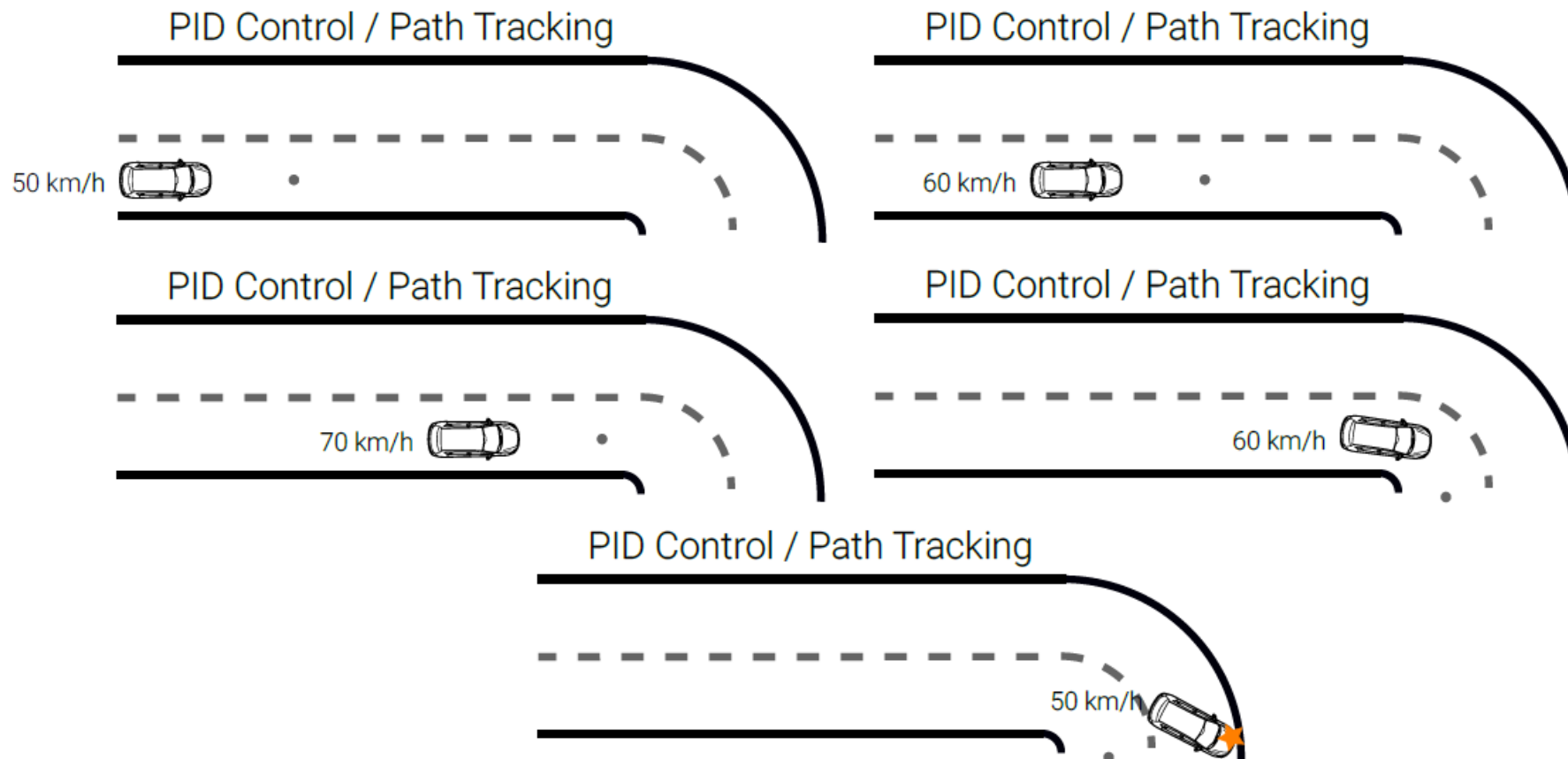
- Predictive control with lookahead
  - PID is reactive control, like driving your car by looking in the rearview mirror
- Handles constraints explicitly
  - PID control cannot handle constraints
- Applicable to both linear and nonlinear systems
  - Pole placement for PID control design is applicable to linear systems only
  - Empirical PID param tuning is model-free and applicable to any system (blackbox)

- Cons

- Requires accurate yet efficient system model (whitebox)
- Optimizer computation may be expensive, esp. for non-linear MPC
  - Select lookahead horizon  $T$  to tradeoff between control performance and computation overhead
  - Small  $T \rightarrow$  myopic behavior, lower overhead; Larger  $T \rightarrow$  better control performance but higher overhead

# PID Control Example (No Look-Ahead)

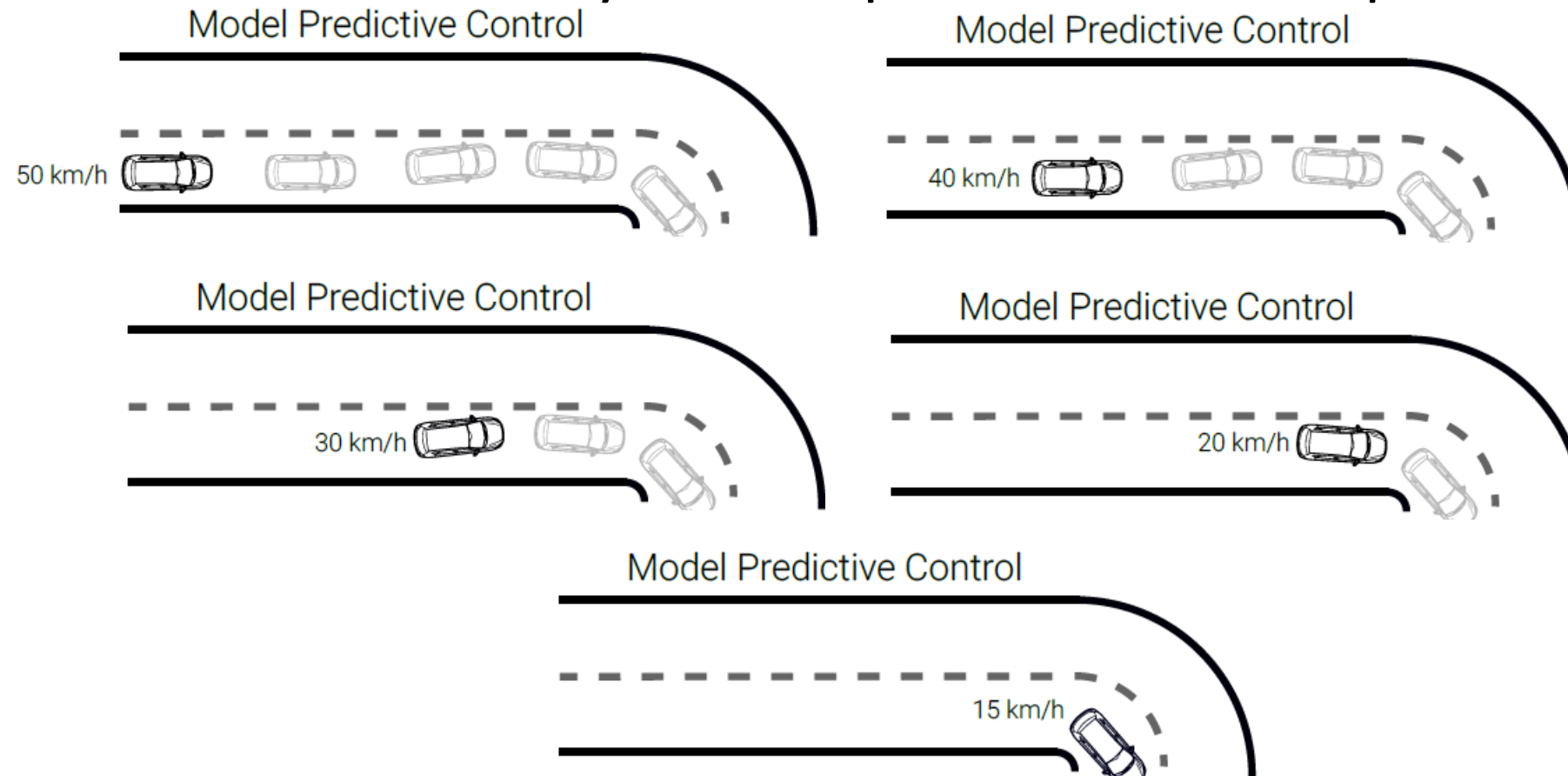
- Vehicle accelerates to track setpoint speed of 70 km/h
- Then decelerates upon encountering a sharp turn, but it is too late and cannot track the lane





# MPC Control Example (Look-Ahead)

- MPC computes control actions based on a lookahead window into the future
- Vehicle decelerates early in anticipation of the sharp turn



# MPC Quiz

---

- Which from the below statement about MPC are true?
  - 1) Horizon is a finite window of time
  - 2) Prediction horizon keeps being shifted at each time step
  - 3) Full optimization over the time horizon is performed at each iteration
  - 4) Only the first control action from the optimization is applied at time  $t$
  - 5) All of the above
- ANS: 5



# Outline

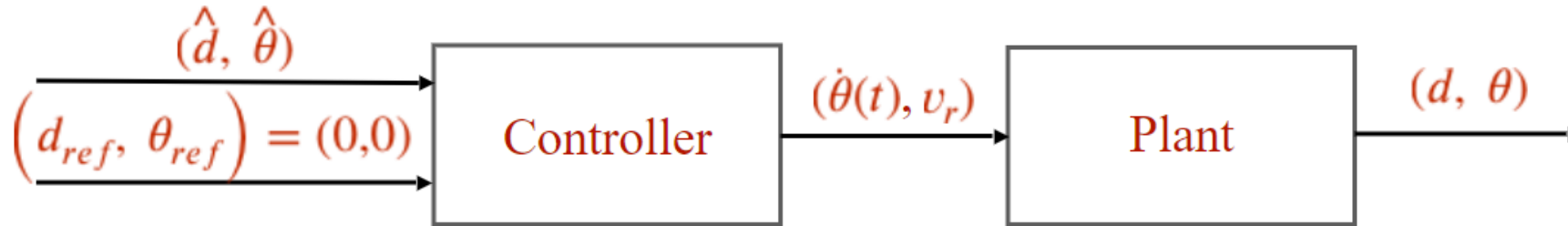
---

- PID Control
- MPC Control
- Kinematic bicycle model
- Twiddle() for PID control tuning

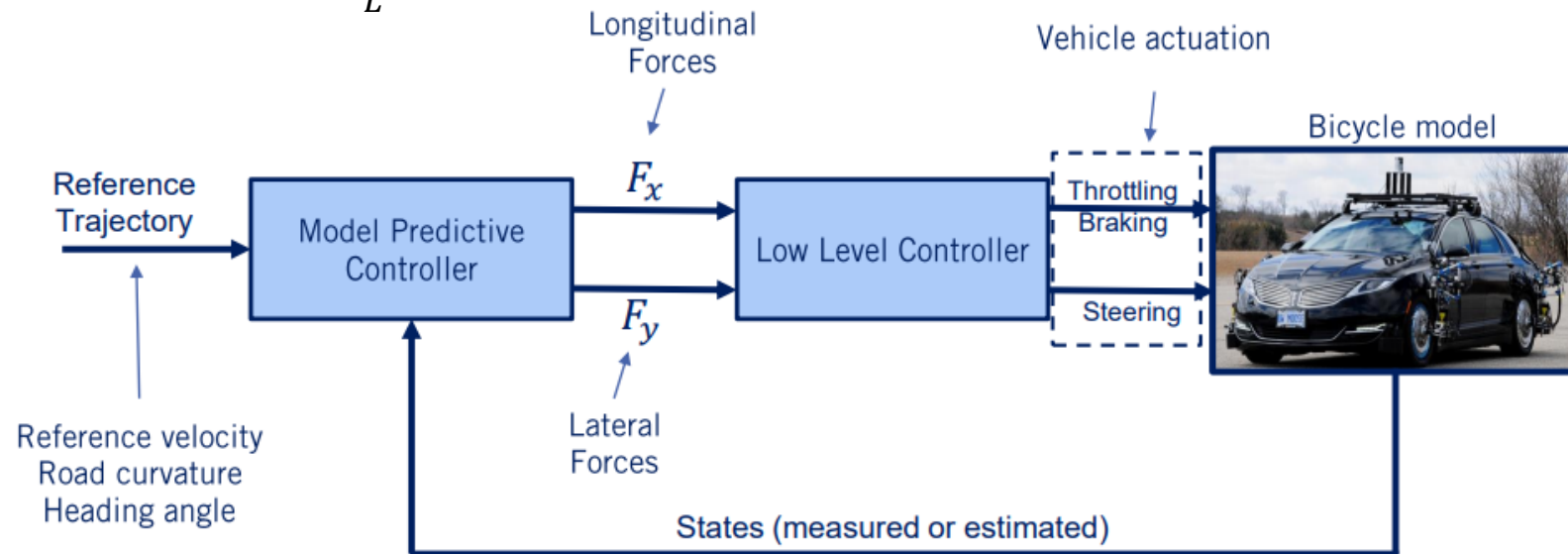
# Kinematics vs. Dynamics

- Kinematics is study of motion without considering the forces that affect the motion. It deals with the geometric relationships that govern the system
  - A kinematic model:  $\dot{x} = v, \dot{v} = \ddot{x} = a$
  - Uses position, velocity, acceleration (and/or further derivatives) as control input (e.g. the kinematic bicycle model)
- Dynamics is the study of motion taking into account the forces that affect it. It is described by the equations of motion.
  - A dynamic model:  $F = M\ddot{x} + B\dot{x} = Ma + Bv$  (Newton's law with friction)
  - Uses force and torque as control input, and takes mass and inertia into consideration.
  - e.g., vehicle dynamics model (longitudinal and lateral)
- Consider two vehicles with the same geometry but different mass/weight turning a tight corner
  - They may have the same kinematic model, but different dynamic model due to different mass  $M$ .
  - Both may be controllable kinematically by control input  $a$ 
    - The light vehicle may be controllable dynamically by control input  $F$ .
    - The heavy vehicle may not be controllable dynamically by control input  $F$ . Due to large  $M$ ,  $F$  may exceed the max actuator limit.
- Orthogonal issue from control algorithm
  - Control algorithms, e.g., PID or MPC, may be either kinematic or dynamic control

# Kinematic vs. Dynamic Control



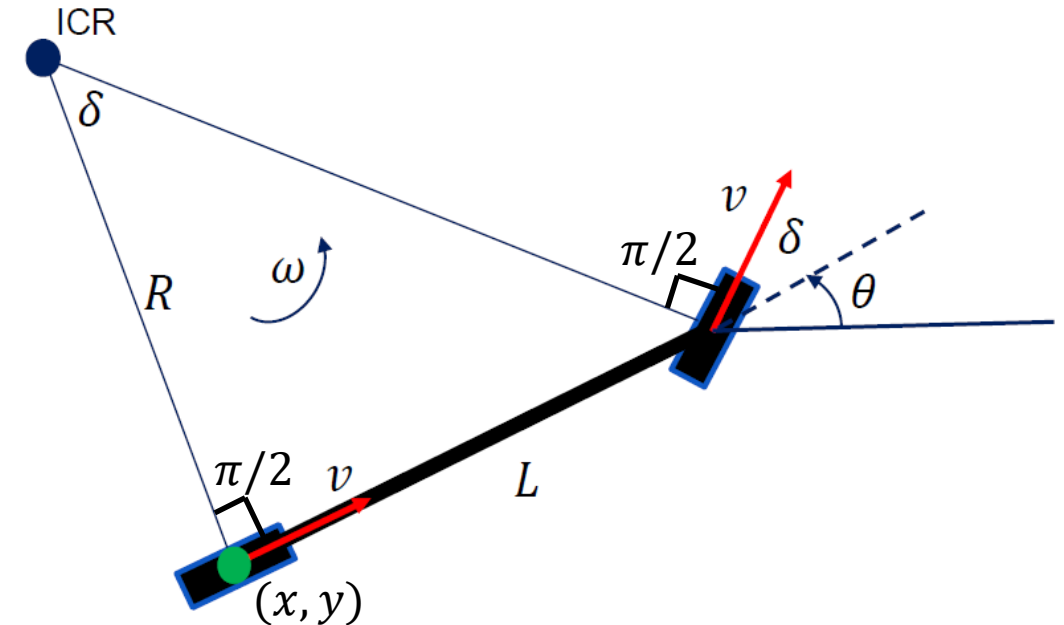
PID controller for kinematic bicycle model with heading angle rate ( $\dot{\theta}$ ) and velocity ( $v_r$ , constant here) as control input. ( $\dot{\theta}$  should be controlled indirectly by controlling steering angle  $\delta$ , with  $\dot{\theta} = \frac{v \tan \delta}{L}$ .)



MPC for dynamic vehicle model (high-level controller) with forces (longitudinal and lateral) as control input.

# Kinematic Bicycle Model

- Front wheel steering. Assuming here rear wheel as reference point (may also use front wheel or center of gravity).
- State vector:  $[x \ y \ \theta]^T$ : vehicle pose includes its position  $(x, y)$  and heading angle  $\theta$ .
- Control inputs:  $[\delta \ v]^T$ : steering angle  $\delta$  and vehicle speed  $v$  (assumed to be constant).
- $\tan \delta = \frac{L}{R}$ 
  - $L$ : vehicle length (distance between 2 wheels);  $R$ : rotation radius of Instantaneous Center of Rotation (ICR), equal to distance between ICR and rear wheel. Curvature  $\kappa = \frac{1}{R}$ .
  - Line from ICR to each wheel is perpendicular to it.
- $\dot{\theta} = \omega = \frac{v}{R} = \frac{v \tan \delta}{L}$ 
  - Angular velocity is speed  $v$  divided by rotation radius  $R$
- Typically, angles  $\delta, \theta$  are based on counter-clockwise convention w.r.t reference direction.
  - In the fig,  $\delta \approx \frac{\pi}{6}$  w.r.t ref direction  $v$ ;  $\theta \approx \frac{\pi}{6}$  w.r.t ref direction east (horizontal);  $\dot{\theta} > 0$  for counter-clockwise rotation.



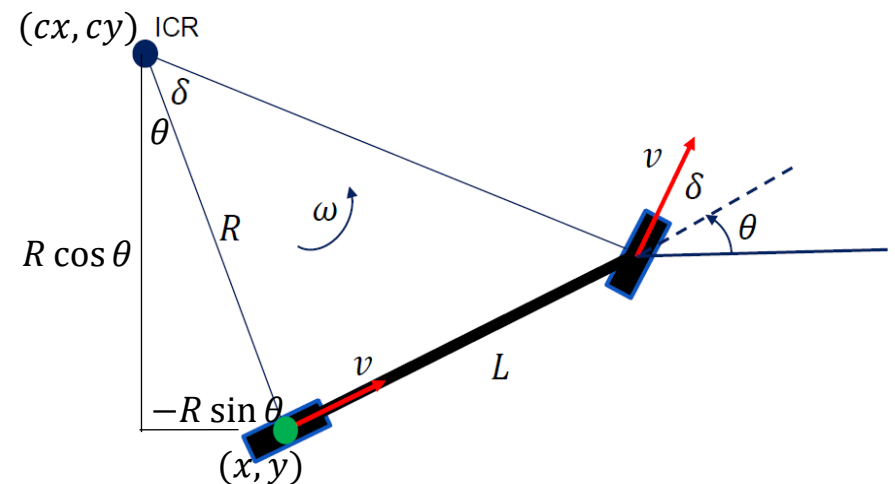
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = v \begin{bmatrix} \cos \theta \\ \sin \theta \\ \frac{\tan \delta}{L} \end{bmatrix}$$

(Non-linear in  $\delta$ ;  
Linear in  $v$ )

# State Update in Python Code

- # apply noise
- steering2 = random.gauss(steering, self.steering\_noise)
- distance2 = random.gauss(distance, self.distance\_noise)
- # apply steering drift
- steering2 += self.steering\_drift
- # noise and drift are all set to 0, so steering2 is steering angle  $\delta$ , distance2 is distance traveled per time step  $vdt$
- # Execute motion
- turn = np.tan(steering2) \* distance2 / self.length ( $\dot{\theta} dt = \frac{vdt \tan \delta}{L}$ )
- **if abs(turn) < tolerance:** (with small  $\dot{\theta} dt$ )
- # approximate by straight line motion
- self.x += distance2 \* np.cos(self.orientation) ( $x += vdt \cos \theta$ )
- self.y += distance2 \* np.sin(self.orientation) ( $y += vdt \sin \theta$ )
- self.orientation = (self.orientation + turn) % (2.0 \* np.pi) ( $\theta = (\theta + \dot{\theta} dt) \% (2\pi)$ )
- **else:** (with large  $\dot{\theta} dt$ )
- # Circular motion around ICR
- radius = distance2 / turn ( $R = \frac{v}{\dot{\theta}}$ ; can also use  $R = \frac{L}{\tan \delta}$ )
- # compute ICR's coordinates (cx, cy)
- cx = self.x - (np.sin(self.orientation) \* radius) ( $cx = x - R \sin \theta$ )
- cy = self.y + (np.cos(self.orientation) \* radius) ( $cy = y + R \cos \theta$ )
- self.orientation = (self.orientation + turn) % (2.0 \* np.pi) ( $\theta = (\theta + \dot{\theta} dt) \% (2\pi)$ )
- # compute vehicle's x, y coordinate after rotation around ICR
- self.x = cx + (np.sin(self.orientation) \* radius) ( $x = cx + R \sin \theta$ )
- self.y = cy - (np.cos(self.orientation) \* radius) ( $y = cy - R \cos \theta$ )

$\%(2\pi)$  (modulo  $2\pi$ ) keeps angles to be within  $2\pi$ ; It can be omitted since it does not affect results of cos, sin functions (assuming no numeric overflow).



# State Update Equation

- Circular motion around ICR (accurate):

$$\bullet \begin{bmatrix} x(t + dt) \\ y(t + dt) \\ \theta(t + dt) \end{bmatrix} = \begin{bmatrix} x(t) - R \sin \theta + R \sin(\theta + \dot{\theta} dt) \\ y(t) + R \cos \theta - R \cos(\theta + \dot{\theta} dt) \\ \theta + \dot{\theta} dt \end{bmatrix}$$

$$\bullet R = \frac{L}{\tan \delta} = \frac{v}{\dot{\theta}}$$

- Straight-line motion for small  $\dot{\theta} dt$  (approximate):

$$\bullet \begin{bmatrix} x(t + dt) \\ y(t + dt) \\ \theta(t + dt) \end{bmatrix} = \begin{bmatrix} x(t) + v dt \cos \theta \\ y(t) + v dt \sin \theta \\ \theta + \dot{\theta} dt \end{bmatrix}$$

$$\bullet \delta \rightarrow 0, \dot{\theta} dt \rightarrow 0$$

# Straight Line vs. Circular Motion

- Straight line motion (with steering angle  $\delta = 0$ ) is a special case of circular motion with radius  $\infty$ .
  - $\delta \rightarrow 0 \Rightarrow R = \frac{L}{\tan \delta} \rightarrow \infty, \dot{\theta} = \frac{v}{R} \rightarrow 0$  (small steering angle  $\delta$  leads to slow angular velocity  $\dot{\theta}$ .)
  - We use this special case to improve computational efficiency for small  $\delta, \dot{\theta}$ , and avoid division by 0.
- $x(t + dt) = x(t) - R \sin \theta + R \sin(\theta + \dot{\theta} dt) = x + R(\sin(\theta + \dot{\theta} dt) - \sin \theta) = x + \frac{v}{\dot{\theta}} * \cos \theta * \dot{\theta} dt = x + v dt \cos \theta$ 
  - $\sin(\theta + \dot{\theta} dt) - \sin \theta \approx \frac{d}{dt} \sin \theta * \dot{\theta} dt = \cos \theta * \dot{\theta} dt$ , for small  $\dot{\theta} dt$ .
- $y(t + dt) = y(t) + R \cos \theta - R \cos(\theta + \dot{\theta} dt) = y - R(\cos(\theta + \dot{\theta} dt) - \cos \theta) = y + \frac{v}{\dot{\theta}} * \sin \theta * \dot{\theta} dt = y + v dt \sin \theta$ 
  - $\cos(\theta + \dot{\theta} dt) - \cos \theta \approx \frac{d}{dt} \cos \theta * \dot{\theta} dt = -\sin \theta * \dot{\theta} dt$ , for small  $\dot{\theta} dt$ .
- Four special cases of straight line motion:
  - $\theta = 0 \Rightarrow x' = x + v dt, y' = y$  (east)
  - $\theta = \pi \Rightarrow x' = x - v dt, y' = y$  (west)
  - $\theta = \frac{\pi}{2} \Rightarrow x' = x, y' = y + v dt$  (north)
  - $\theta = \frac{3\pi}{2} \Rightarrow x' = x, y' = y - v dt$  (south)

# PID Control Lab Setup

---

- Based on Udacity course Lesson AI for Robotics, Lesson 15: PID Control
  - <https://classroom.udacity.com/courses/cs373/>
- Lateral control of a car to run along a race track (either straight line or circular) with fixed speed.



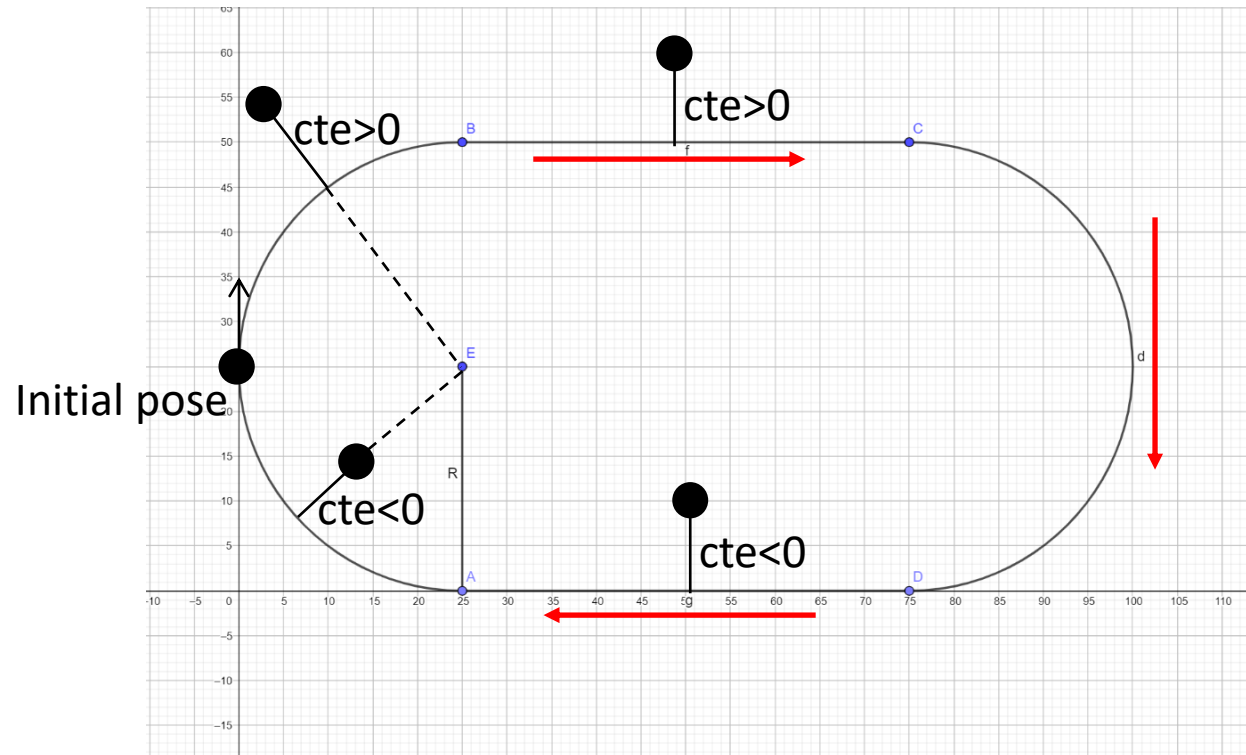
# Outline

---

- PID Control
- MPC Control
- Kinematic bicycle model
- Twiddle() for PID control tuning

# Udacity: Racetrack Control

- Lesson 16: Problem Set 5, 4.  
Quiz: Racetrack control.
- Cross-track error (cte): lateral distance (of rear wheel) to desired trajectory, defined as:
  - if  $x \in [radius, 3 * radius]$ : deviation from the straight horizontal track (with the implicit assumption that the car will not deviate from the track for more than *radius*.)
  - Otherwise: deviation from each semi-circle track.
- For clockwise traversal,  $cte > 0$  if car is outside of the track region;  $cte < 0$  if inside.



# run()

- Implements the PID controller for with PID gains
  - params[0] as  $K_p$ ; params[1] as  $K_d$ ; params[2] as  $K_i$ .
  - Returns actual trajectory as arrays of x\_trajectory[], y\_trajectory[]; and average error, defined as sum of squared cte
- For clockwise travel direction:
  - If cte>0, car is outside of track region; steering angle  $\delta$  should decrease (turn right according to the counter-clockwise convention)
  - If cte<0, car is inside of track region; steering angle  $\delta$  should increase (turn left)
  - Based on the above,  $K_p$  should be positive (this can be a sanity check for your final solution)

```
def run(robot, params, n=100, speed=1.0):  
    x_trajectory = []  
    y_trajectory = []  
    err = 0  
    # TODO: your code here  
    prev_cte = robot.y  
    int_cte = 0  
    for i in range(2 * n):  
        cte = robot.y  
        diff_cte = cte - prev_cte  
        int_cte += cte  
        prev_cte = cte  
        steer = -params[0] * cte - params[1] * diff_cte - params[2] * int_cte  
        robot.move(steer, speed)  
        x_trajectory.append(robot.x)  
        y_trajectory.append(robot.y)  
        if i >= n:  
            err += cte ** 2  
    return x_trajectory, y_trajectory, err / n
```

This is for straight line track; need to call myrobot.cte() for circular track.

PID control law

# For General $dt$

- The Python code assumes controller time step size  $dt = 1$ .
  - Also,  $v$  is constant.  $\delta$ 's range is unconstrained (steering wheel can be turned to arbitrary angle).
- For general  $dt$ , the following needs to be modified:
  - Call `myrobot.move(steer, speed*dt)`, to match its definition `move(self, steering, distance,...)`
  - In the PID control law:
    - Differential of error  $diffcte = \frac{cte - prevcte}{dt}$
    - Integral of error  $intcte += cte * dt$

# twiddle()

- twiddle: “twist, move, or fiddle with (something), typically in a purposeless or nervous way” –Encyclopedia.

Adjust each  $p[i]$  in turn.

Update  $p[i]$  to  $p[i] + dp[i]$  and `run()`. If error decreases, then  $p[i]$  was updated in the right direction. Keep the update  $+dp[i]$  to get  $p[i] + dp[i]$ , and increase step size to  $1.1 * dp[i]$ .

If error increases, then  $p[i]$  was updated in the wrong direction. Update in the opposite direction by  $-2 * dp[i]$  to get  $p[i] - dp[i]$ . Run again.

If error decreases, then  $p[i]$  was updated in the right direction. Keep the current  $p[i] - dp[i]$ , and increase step size to  $1.1 * dp[i]$ .

If error increases, then  $p[i]$  was updated in the wrong direction. Update in the opposite direction by  $+dp[i]$  to get back the original  $p[i]$ . Reduce step size to  $.9 * dp[i]$ .

```
def twiddle(tol=0.2):  
    # TODO: Add code here  
    # Don't forget to call 'make_robot' before you call 'run'!  
    p = [0.0, 0.0, 0.0]  
    dp = [1.0, 1.0, 1.0]  
    robot = make_robot()  
    x_trajectory, y_trajectory, best_err = run(robot, p)  
  
    it = 0  
    while sum(dp) > tol:  
        # print("Iteration {}, best error = {}".format(it, best_err))  
        for i in range(len(p)):  
            p[i] += dp[i]  
            robot = make_robot()  
            x_trajectory, y_trajectory, err = run(robot, p)  
  
            if err < best_err:  
                best_err = err  
                dp[i] *= 1.1  
            else:  
                p[i] -= 2 * dp[i]  
                robot = make_robot()  
                x_trajectory, y_trajectory, err = run(robot, p)  
  
            if err < best_err:  
                best_err = err  
                dp[i] *= 1.1  
            else:  
                p[i] += dp[i]  
                dp[i] *= 0.9  
  
        it += 1  
    return p, best_err
```

# Changing to P Control

- Setting  $dparams[1] = dparams[2] = 0$  turns it into a P controller, since  $K_d, K_i$  are initialized to 0 and never changed.  $K_p$  is adjusted in `twiddle()`
- `twiddle()` is a local optimization algorithm, so perf is dependent on parameter initialization. Here are initialized to 0 for simplicity.
  - <https://classroom.udacity.com/courses/cs373/lessons/91f48b5b-a063-41f9-ace6-5fb9e7508941/concepts/b740218e-b0eb-40dc-80af-343912305293>

#### ENTER CODE BELOW THIS LINE

```
n_params = 3
dparams   = [1.0 for row in range(n_params)]
params    = [0.0 for row in range(n_params)]
dparams[2] = 0.0
dparams[1] = 0.0
```

```
best_error = run(params)
n = 0
while sum(dparams) > tol:
    for i in range(len(params)):
```

# twiddle() vs. Pole Placement

- twiddle() is a model-free approach to tuning PID controller by Gradient descent. It adjusts each PID gain parameter systematically, gradually decreasing or increasing step of adjustment  $dp[i]$  (the gradient) until convergence to minimum best\_err.
- Pole placement for PID controller design is model-based, but requires a linear model
  - $$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = v \begin{bmatrix} \cos \theta \\ \sin \theta \\ \frac{\tan \delta}{L} \end{bmatrix} \approx \begin{bmatrix} v \\ v\theta \\ u \end{bmatrix}$$
  - With simplifying assumptions:
    - $\theta$  is small
    - Simplified kinematic model:  $u = \dot{\theta}$  as control input instead of  $\delta$