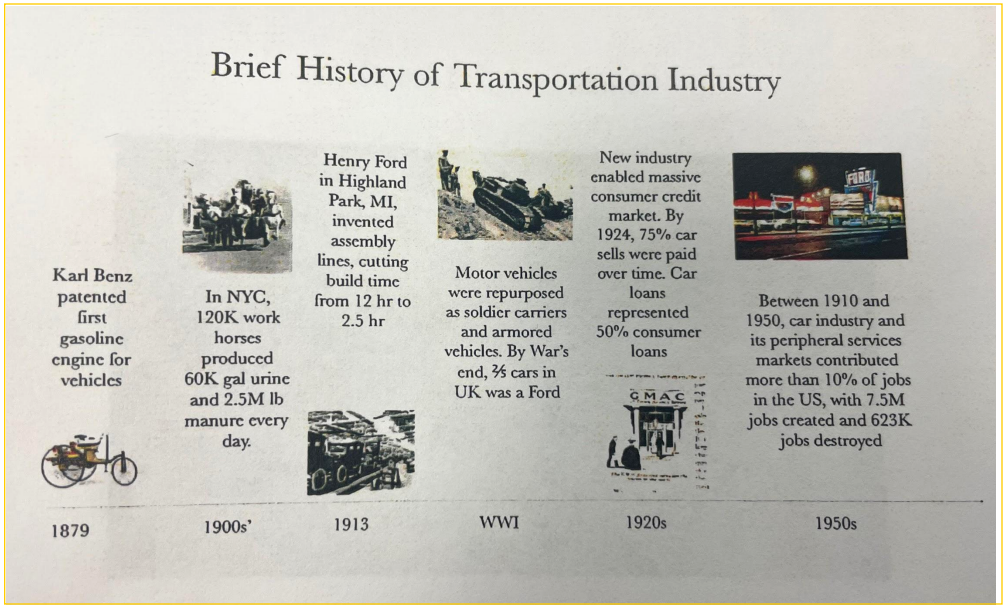
Autonomous Driving

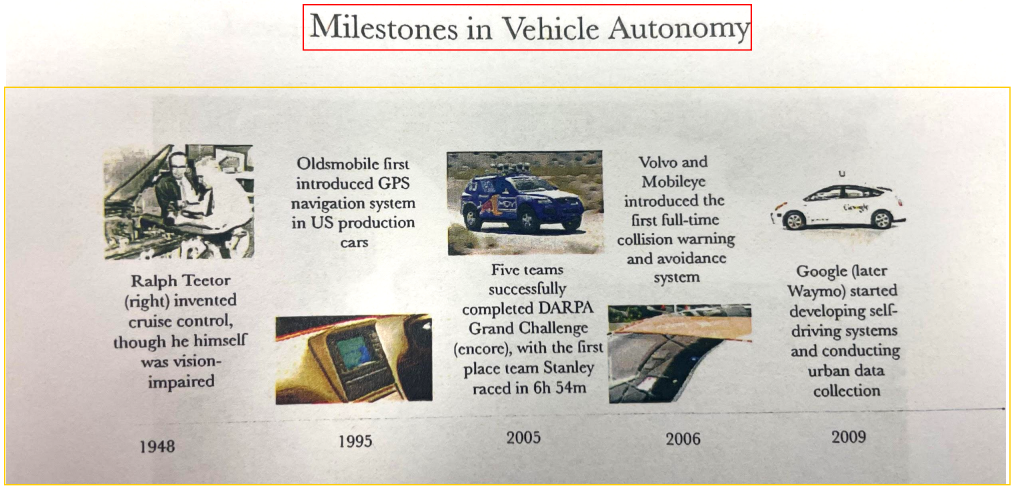
Intro to Autonomous Driving

(See slides for History of the Transportation Industry + Milestones in Vehicle Autonomy)

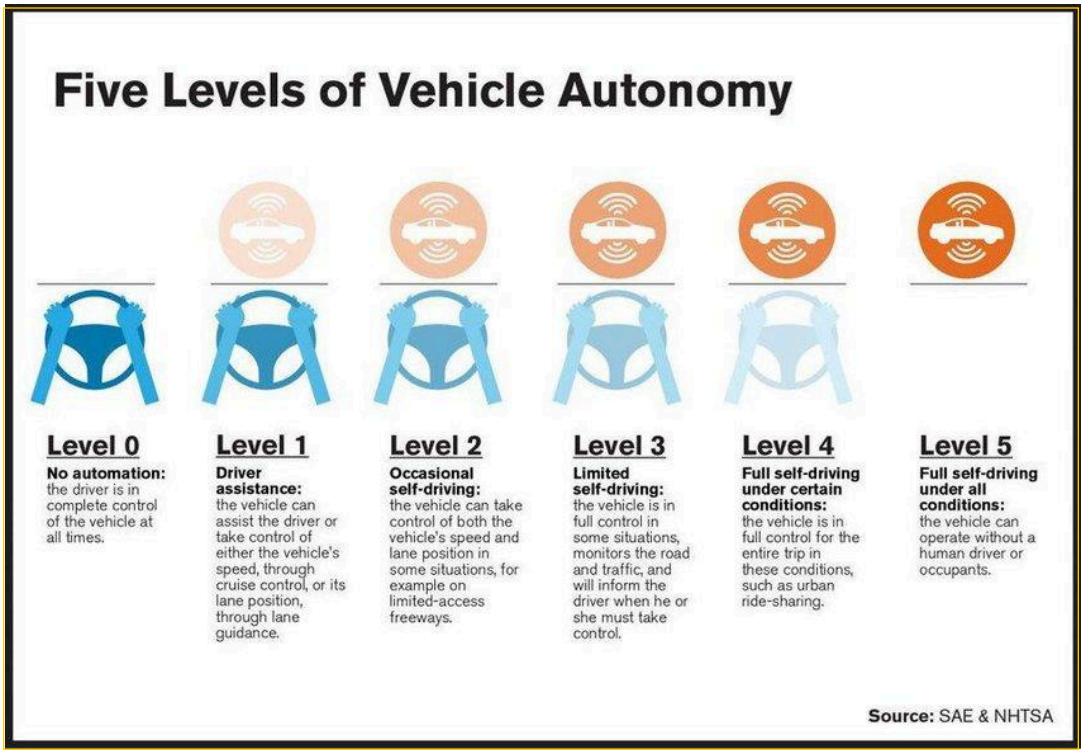
Brief History of the Transportation Industry



Milestones in Vehicle Autonomy



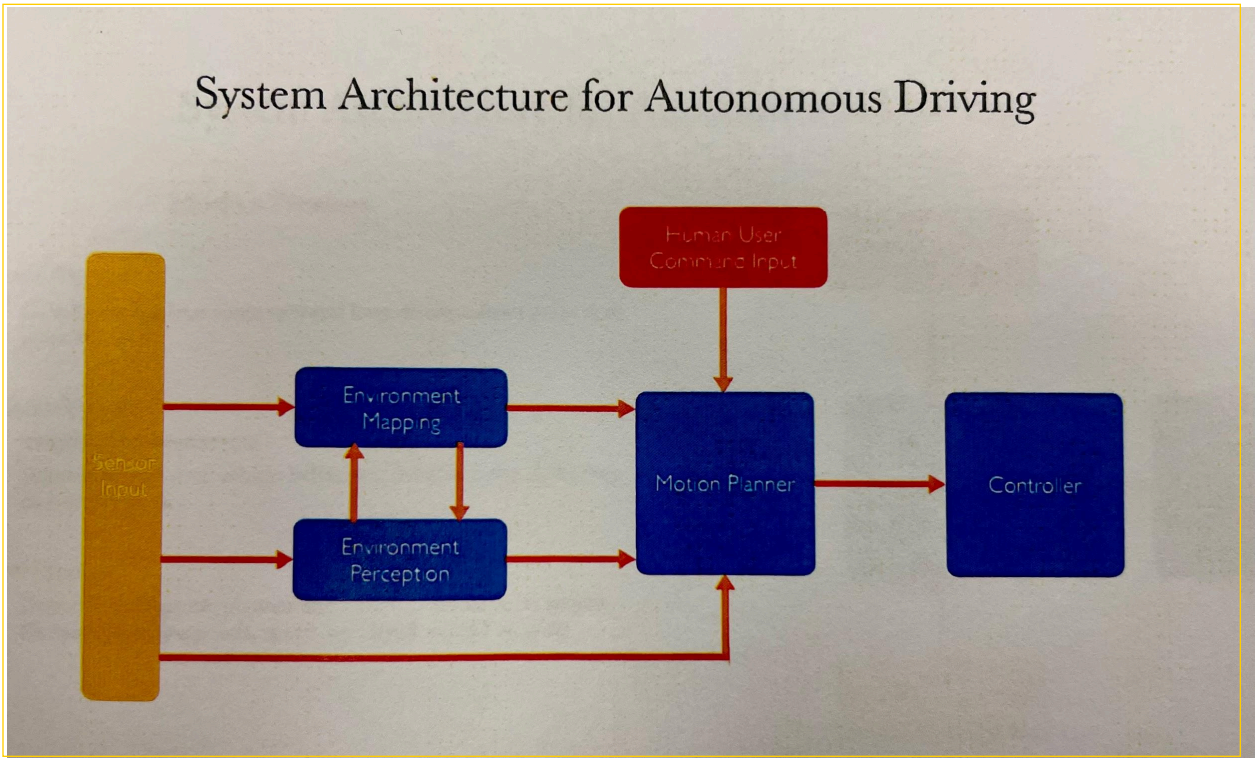
Five Levels of Vehicle Autonomy



- **Lvl 0: no automation**
  - The human driver is fully in control
- **Lvl 1: driver assistance**
  - Alerts the driver to dangers, or intervenes briefly by performing simple maneuver
  - E.g. cruise control
  - E.g. automatic reverse braking
- **Lvl 2: supervised occasional self-driving**
  - The human still has to sit in the car
  - Most of the time the human driver is in control
  - In some cases the car can take control and drive autonomously
  - E.g. it parks the car while you are in the driver's seat
- **Lvl 3: limited self-driving**
  - → The vehicle is in full control in some situations
  - Most of the time the car can drive autonomously
  - BUT the driver will have to take control of the car at some point
  - E.g. the car can go park
- **Lvl 4: full self-driving under certain conditions**
  - Conditions → usually = approved driving within certain areas
  - There is no wheel, since there does not have to be a driver
  - BUT it is geofenced → the car can only do autonomous driving in an approved area
  - E.g. waymo
- **Lvl 5: full self-driving**

- The car can use autonomous driving to travel anywhere with no human intervention

System Architecture for Autonomous Driving



For ROAR competition

- NO environment perception + NO human user command input

Sensor Input

- Spatial Sensing:
  - Short distance: ultrasound
  - Midrange distance: 3D depth camera + LIDAR
  - Long Distance: RGB Camera
- Global Position: GPS System
- Vehicle State Sensing:
  - Motor speed sensor
  - Wheel speed sensor
  - Steering servo sensor
  - Inertial measurement unit (IMU)

Environment Mapping and Perception

- Environment Mapping:
  - 3D localization

    - Road condition mapping: road, scenes, lanes
- Environment Perception:
    - Object detection & recognition
    - Object tracking

Motion Planner

- Mission Planner
    - Calculating the best route to travel long distance from point A to point B
    - E.g. Turn to get to destination
- Behavior Planner
    - Vehicle safety monitoring
    - Determines real-time vehicle behaviors: overtake, stop, following, on/off ramp, etc.
    - E.g. How the vehicle will turn (slow down, steer, etc.)
- Local Planner
    - Based on the mission planner and behavior planner –. Sets targets for immediate waypoints, speed, acceleration, and steering
    - E.g. Step by step how to turn from the vehicle's current position

Controller

- Longitudinal (Speed) Control
    - Control vehicle motor's speed and acceleration
    - Equivalent to driver adjusting acceleration and braking
- Lateral (Steering) Control
    - Control vehicle steering and orientation
    - Equivalent to driver adjusting the steering wheel
- Popular control algorithms
    - Pure Pursuit Control
    - PID (Proportional-Integral-Derivative) Control
    - MPC (Model-Predictive Control

PID Control

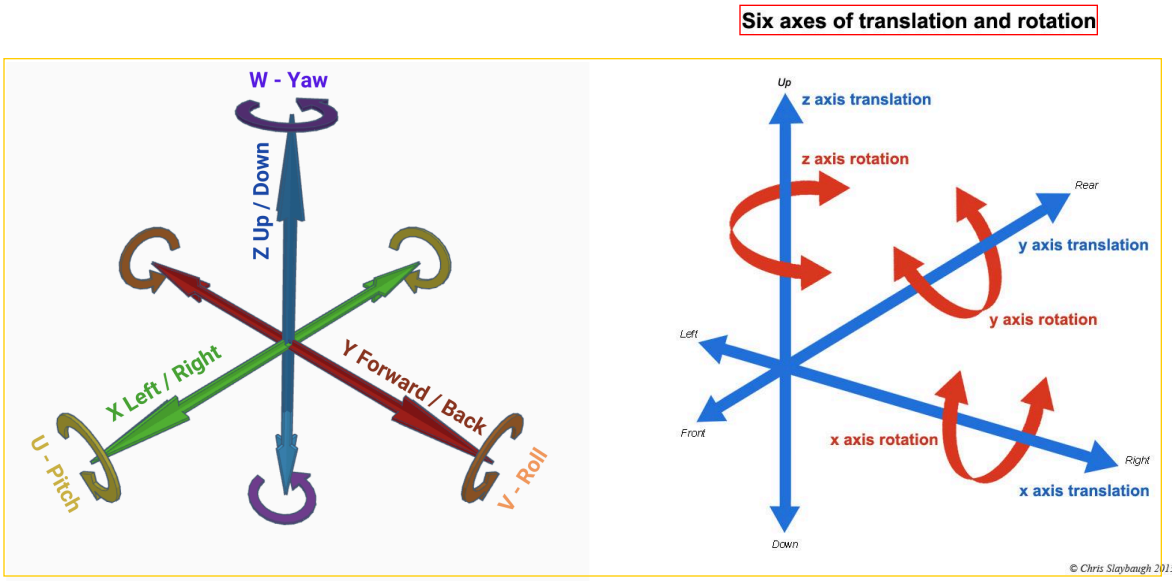
A PID (Proportional-Integral-Derivative) controller is a control system that uses feedback to regulate process variables by continuously adjusting the output to match a desired setpoint.

- Drive $\rightarrow$ just means to change the **state** of some system

Modeling Robot Kinematics in DoF

- A rigid-body motion has 6 **degrees of freedom** in 3D space
    - $(x, y, z, \alpha, \beta, \gamma)$

- Note: going back & forth → only 1 degree of freedom (bc its moving along one axis)
- $\alpha, \beta, \gamma \rightarrow$ represent turning along each x, y, z axis
  - Represents: pan, tilt, roll
- Note: turning right & left is similarly only 1 degree of freedom (bc its turning along one axis)



- Robots that are modeled as rigid body and have 6 DoF:
  - Quad-rotor helicopters
  - Fixed-wing airplanes
  - Underwater vehicles

- Q: How many DoF does BB-8 or R2D2 have? → 5 (everything except flying/jumping up)
- A rolling ball on the surface has five DoF
  - Planar Movement: (x, y)
  - Planar Rotation: γ
  - Body Pitch: β
  - Body Roll: α

- Q: How many DoF of the human arm?
  - 3 DoF at the Shoulder
  - 2 DoF at the Elbow
  - 2 DoF at the Wrist
- ⇒ Total 7 DoF

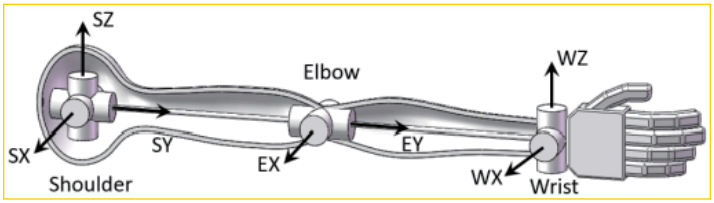


Figure 1. The DOF configuration of the humanoid arm

TABLE I. THE RANGE OF MOTION OF EACH DOF

Four-Wheel Ground Vehicle Model

Basic Four-Wheel Vehicle Model

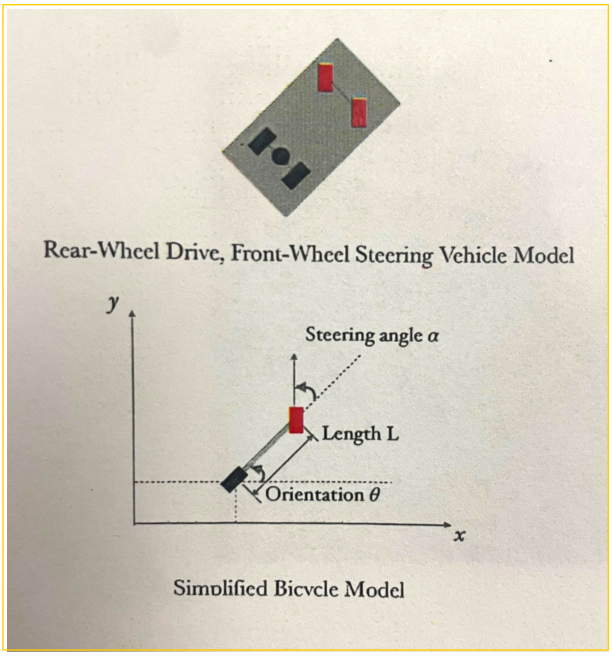
- Driving wheels at the rear
- Steering wheels at the front
- 3 DoF constrained on the ground surface: (x, y, θ)
  - Can only pan right & left

- Having full control of all these DoF → means we have achieved autonomous driving (Even tho the vehicle can tilt, like when going up a hill → the vehicle itself isn't exactly controlling the tilt)

Simplified Bicycle Model (simple way to think of vehicles' DoF)

- Four wheels simplified to two bicycle

- Vehicle center of mass at the center of the rear wheel: (x, y)
- Wheel-to-wheel length: L
- Orientation along the length-L body: θ
- Steering angle: α

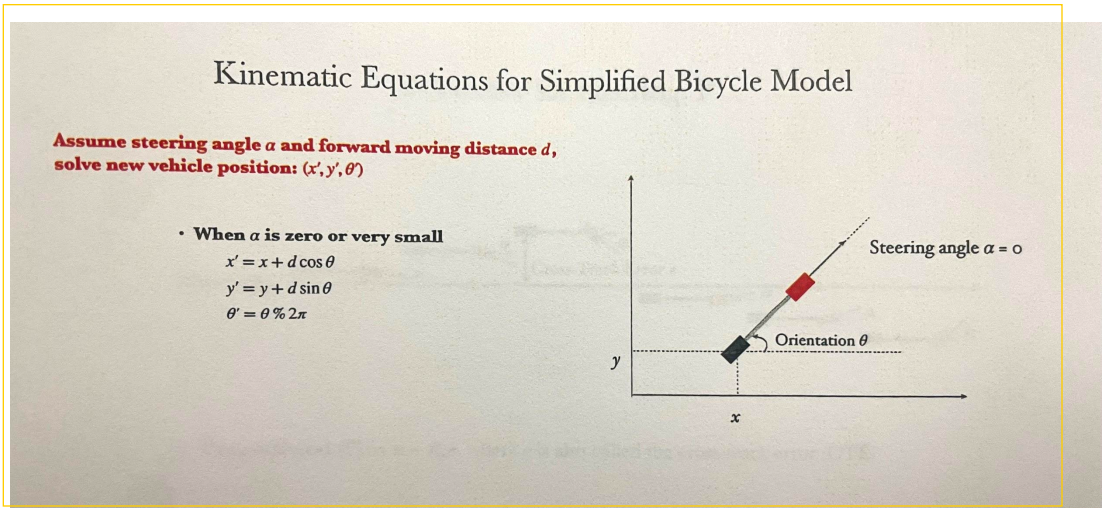


Note:

- Center of mass → at the rear wheel → so that's where we measure θ from
  - α just represents where we steer from (unlike θ, it's not a state/DoF)
- Rear-wheel drive, front-wheel steering
  - → Bc the acc path of the turn is followed by the rear wheels, while the front wheels are free to turn (bc of steering angle α)

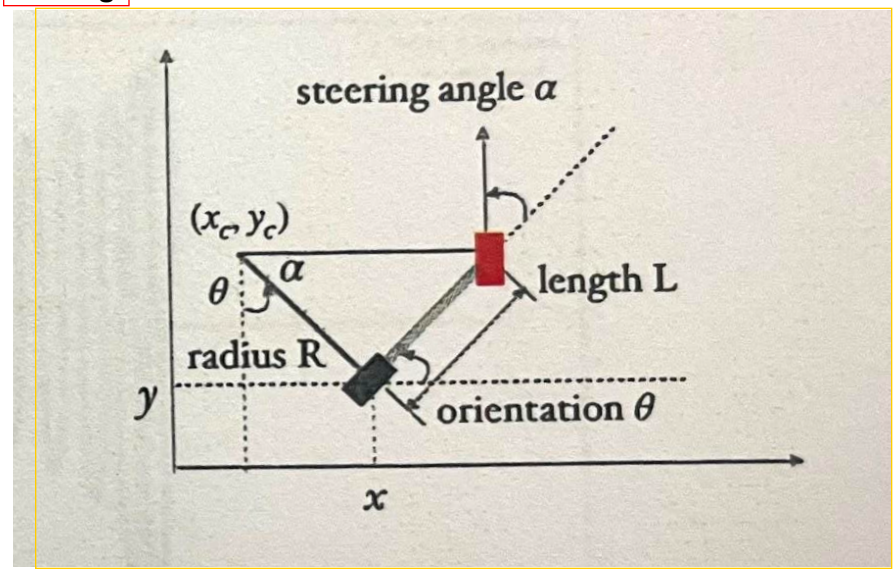
Kinematic Equations for Simplified Bicycle Model

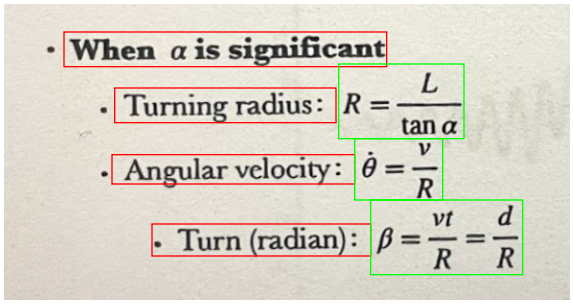
Moving forward:



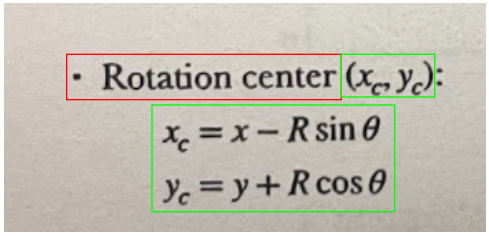
- d is total distance (d = vt)
- When α is zero or very small:
  - x' = x + dcosθ
  - y' = y + dsinθ
  - θ' = θ % 2π
- The red front wheels → shows new position x', y', θ'

Turning:

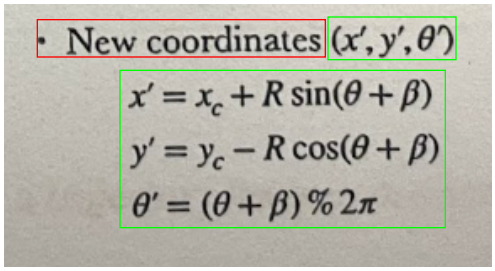




- Bc of the turning radius equation → longer vehicles → have to move their wheels further away to make a big turn
  - If α is big (you turn ur wheel a lot) → you can make a small turn
- $\alpha \rightarrow$ static
- $\beta \rightarrow$ represents how much you turn (not static, increases as you keep traveling through the turn)
  - β is the total travel distance/R



- Just use trig

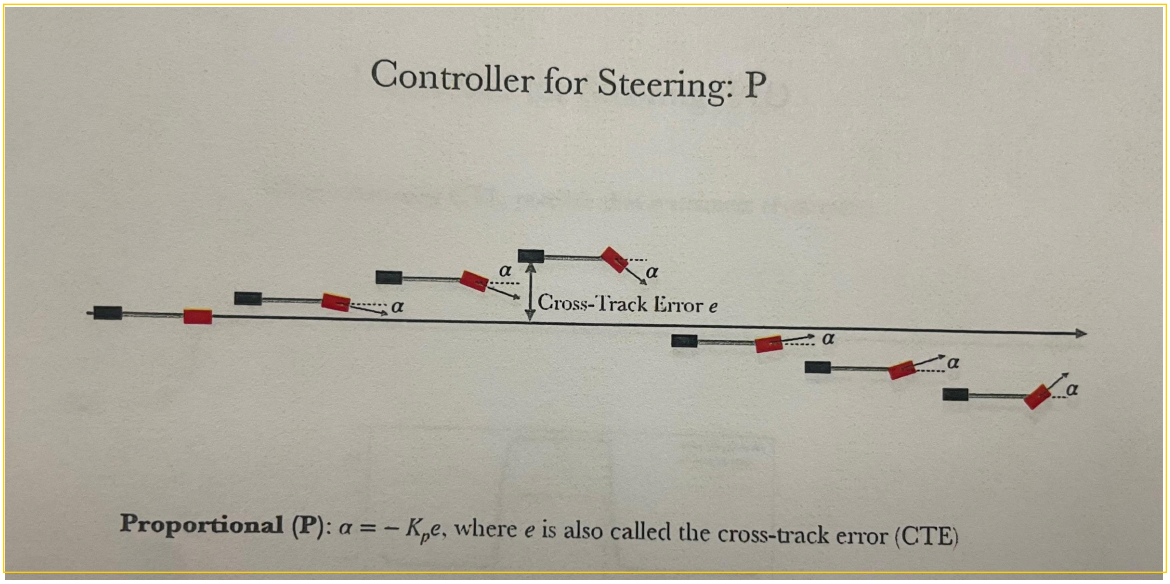


- Add β to $\theta \rightarrow$ bc we are turning β (radians)

PID Controller: Tracking Preset Trajectories

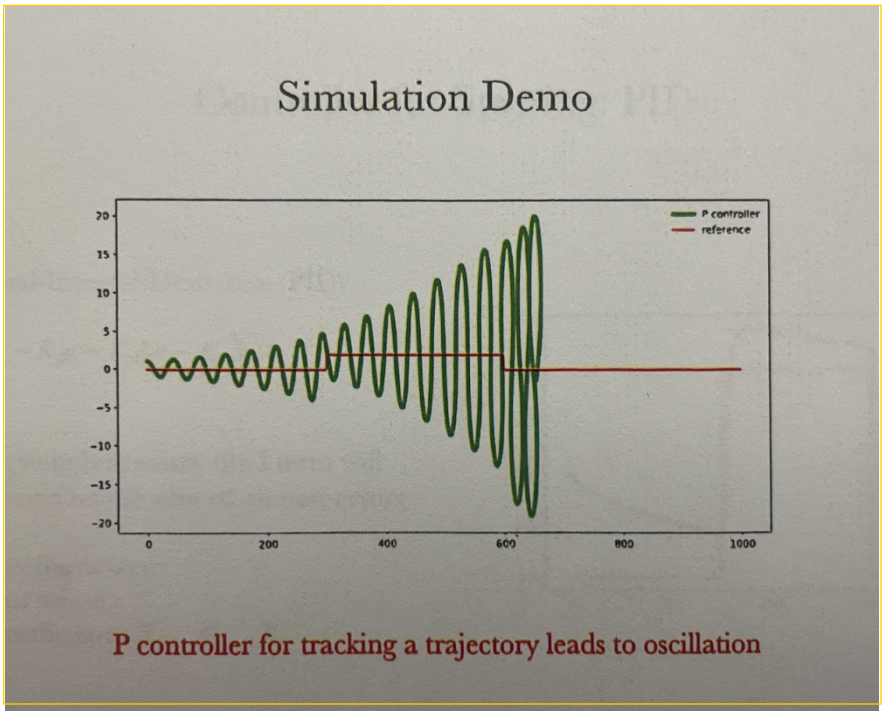
- $K_p \rightarrow$ coefficient you choose

Controller for Steering P

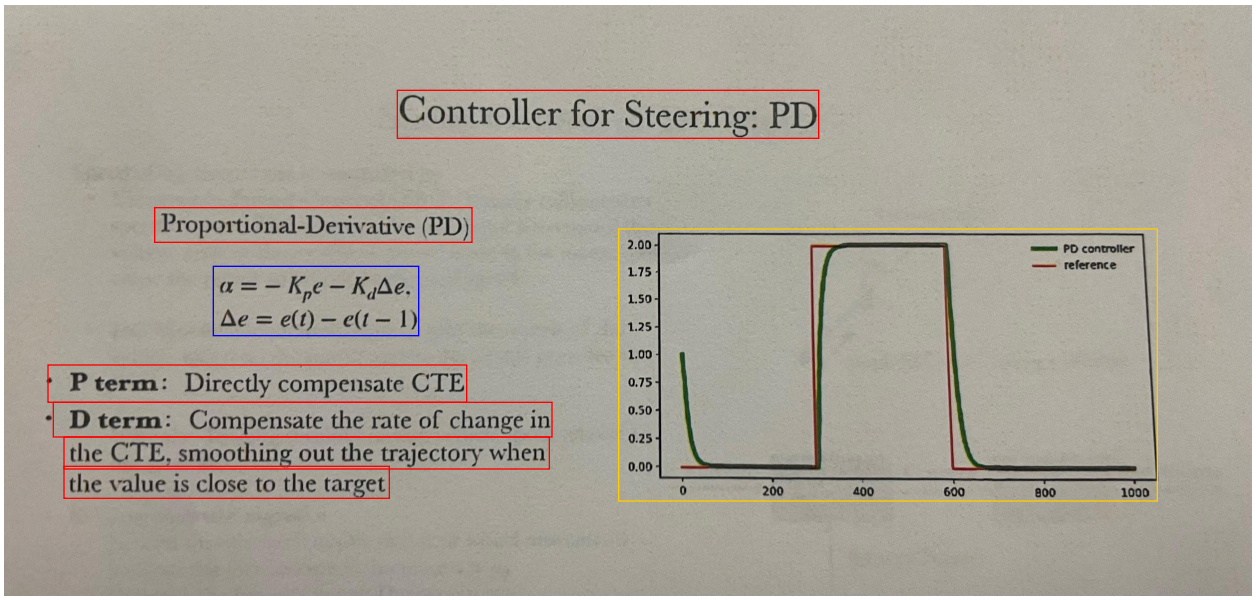


- Multiply the error (the difference from where u want to be) → by a coefficient you choose (K_p) to reduce the error

BUT problem:



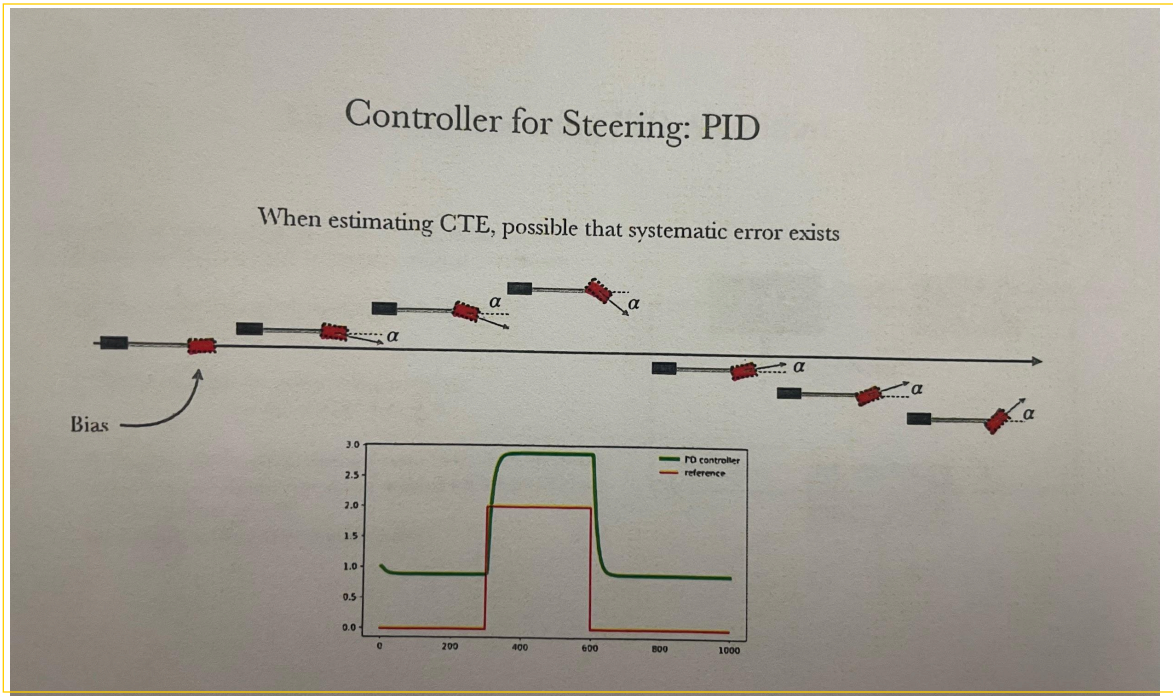
Controller for Steering PD



- D for derivative
- $\alpha \rightarrow$ takes into account the change in the $K_p e$
- **P term**: Directly compensates CTE (cross track error)
- **D term**: Compensates the rate of change in the CTE, smoothing out the trajectory when the value is close to the target
    - I.e. If the error is decreasing rapidly, D control will reduce the corrective action, preventing overshoot.

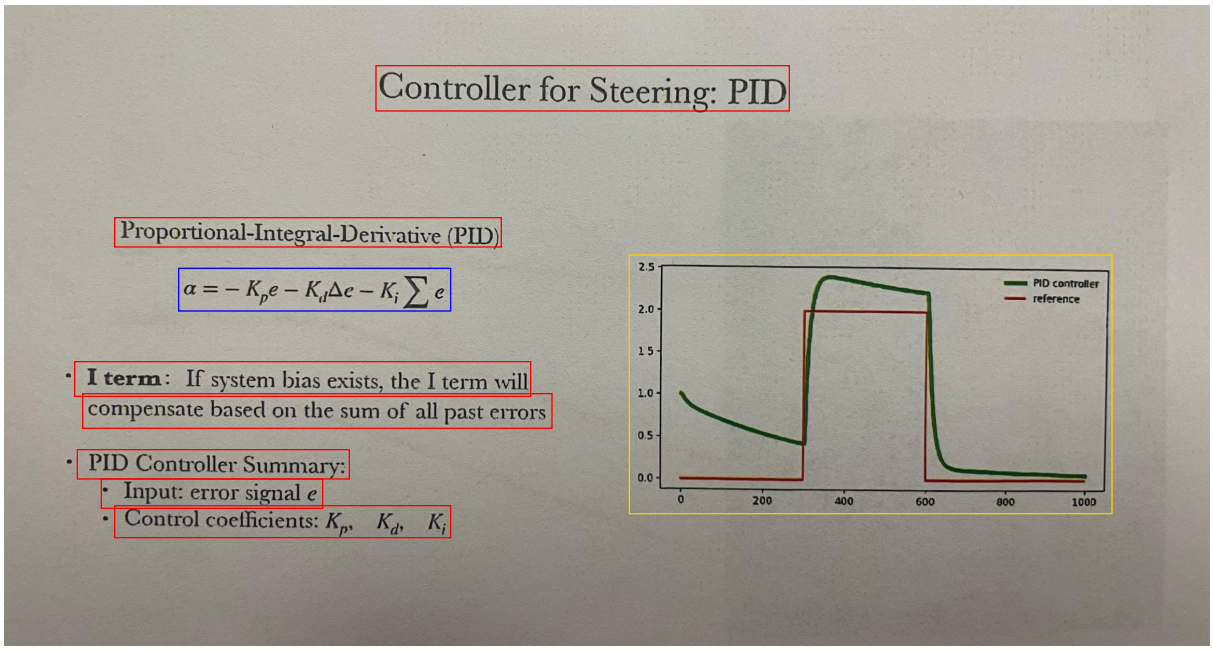
Controller for Steering PID

Problem:



- Systematic errors (like car misalignment) arise → shown by the bias
- If the car is slightly misaligned → the error will accumulate over time

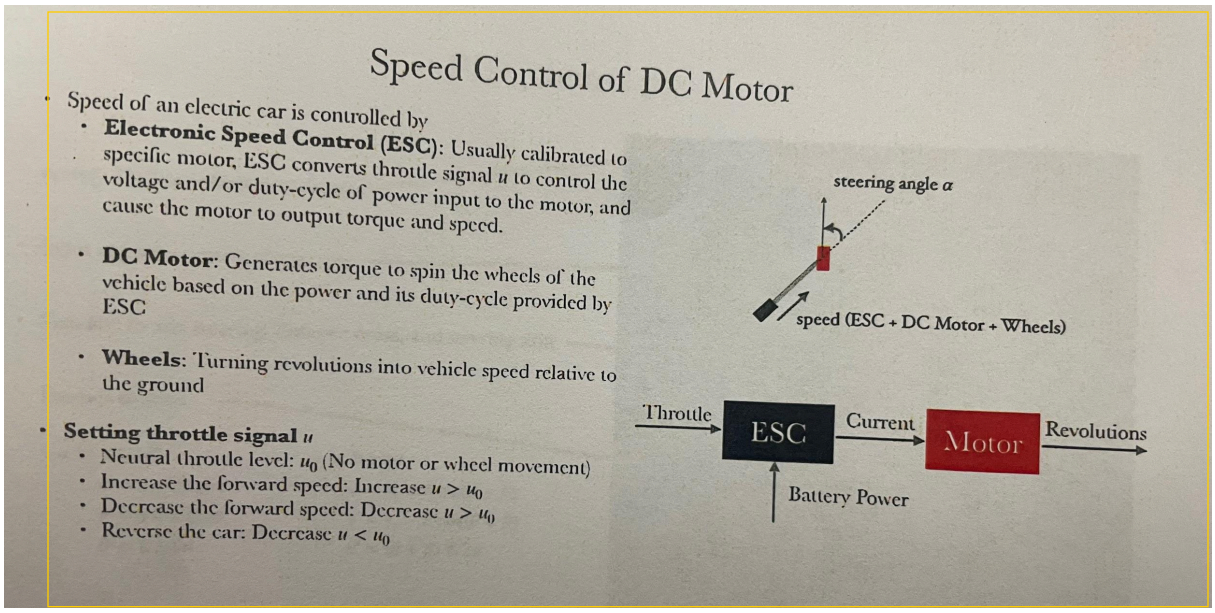
SO:



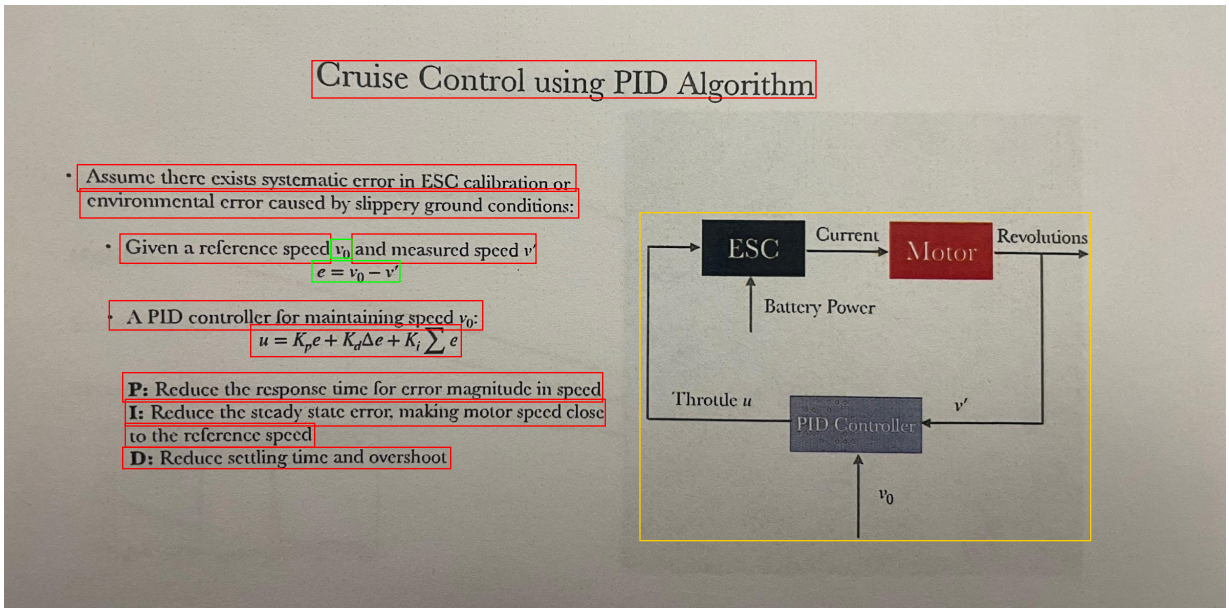
- I for integral
- Mechanism: The integral term accumulates the error over time, meaning that even a small, persistent error will add up.
- As the integral of the error grows (i.e. if we've been going to the wrong dir for a long time), it produces an increasing corrective action until the error is driven to zero.

- Result: This helps to eliminate the steady-state error that remains with PD control alone.

Speed Control of DC Motor



Cruise Control using PID Algorithm



- PID can applied to controlling anything
- Very simple calculations

PID_Control.py

```
## This is course material for Introduction to Modern Artificial Intelligence
## Example code: PID_control.py
## Author: Allen Y. Yang
##
## (c) Copyright 2020. Intelligent Racing Inc. Not permitted for commercial use

import random
import numpy as np
import matplotlib.pyplot as plt

class Vehicle2D(object):
    def __init__(self, length=10.0):
        """
        Creates 2D vehicle model and initializes location (0,0) & orientation 0.
        """
        self.x = 0.0
        self.y = 0.0
        self.orientation = 0.0
        self.length = length
        self.steering_noise = 0.0
        self.distance_noise = 0.0
        self.steering_drift = 0.0

    def set(self, x, y, orientation):
        """
        Sets the vehicle coordinates and orientation.
        """
        self.x = x
        self.y = y
        self.orientation = orientation % (2.0 * np.pi)

    def set_noise(self, steering_noise, distance_noise):
        """
        Sets the noise parameters.
        """
        # makes it possible to change the noise parameters
        self.steering_noise = steering_noise
        self.distance_noise = distance_noise

    def set_steering_drift(self, drift):
        """
```

```
        Sets the systematical steering drift parameter
        """
        self.steering_drift = drift

    def move(self, steering, distance, tolerance=0.001, max_steering_angle=np.pi /
4.0):
        """
        steering = front wheel steering angle, limited by max_steering_angle(max 90
degree)
        distance = total distance driven, most be non-negative
        """
        if max_steering_angle >np.pi/2:
            raise ValueError("Max steering angle cannot be greater than 90-degree.")

        if steering > max_steering_angle:
            steering = max_steering_angle
        if steering < -max_steering_angle:
            steering = -max_steering_angle
        if distance < 0.0:
            distance = 0.0

        # apply noise
        steering2 = random.gauss(steering, self.steering_noise)
        distance2 = random.gauss(distance, self.distance_noise)

        # apply steering drift
        steering2 += self.steering_drift

        # Execute motion
        turn = np.tan(steering2) * distance2 / self.length

        if abs(turn) < tolerance:
            # approximate by straight line motion
            self.x += distance2 * np.cos(self.orientation)
            self.y += distance2 * np.sin(self.orientation)
            self.orientation = (self.orientation + turn) % (2.0 * np.pi)
        else:
            # approximate bicycle model for motion
            radius = distance2 / turn
            cx = self.x - (np.sin(self.orientation) * radius)
            cy = self.y + (np.cos(self.orientation) * radius)
            self.orientation = (self.orientation + turn) % (2.0 * np.pi)
```

```
            self.x = cx + (np.sin(self.orientation) * radius)
            self.y = cy - (np.cos(self.orientation) * radius)

    def compute_error(self, track):
        min_distance = float('inf')
        for i in range(len(track)):
            distance = np.linalg.norm(np.array([self.x, self.y])- np.array(track[i]))
            if distance<min_distance:
                min_distance = distance
                error = self.y - track[i][1]  # This error only assumes cte based on
y-axis. Further extension is possible

        return error

    def run_PID(self, track, K_p, K_d, K_i, n=500, speed=1.0):
        x_trajectory = []
        y_trajectory = []

        prev_error = self.compute_error(track)
        cumulative_error = 0
        for _ in range(n):
            current_error = self.compute_error(track)
            cumulative_error += current_error
            diff = current_error - prev_error
            prev_error = current_error
            steer = -K_p * current_error - K_d * diff - K_i*cumulative_error
            self.move(steer, speed)
            x_trajectory.append(self.x)
            y_trajectory.append(self.y)

        return x_trajectory, y_trajectory

    def __repr__(self):
        return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)

vehicle = Vehicle2D()
vehicle.set(0, 1, 0)
vehicle.set_steering_drift(10/180.*np.pi)

track_length = 1000
targets = []; track_x = []; track_y = []
for i in range(300):
```

```
    targets.append([i, 0]); track_x.append(i); track_y.append(0)
for i in range(300,600):
    targets.append([i, 2]); track_x.append(i); track_y.append(2)
for i in range(600,track_length):
    targets.append([i, 0]); track_x.append(i); track_y.append(0)

# Set the parameters for K_p, K_d, and K_i
K_p = 0.2; K_d = 3; K_i = 0.0005
x_trajectory, y_trajectory = vehicle.run_PID(targets, K_p, K_d, K_i, track_length)
n = len(x_trajectory)

controller_legend = 'PID controller' if K_p!=0 and K_d!=0 and K_i!=0 else 'PD
controller'\
    if K_p!=0 and K_d!=0 else 'P controller' if K_p!=0 else 'unknown controller'
fig, ax1 = plt.subplots(1, 1, figsize=(8, 4))
ax1.plot(x_trajectory, y_trajectory, 'g', linewidth=4, label= controller_legend)
ax1.plot(track_x, track_y, 'r', label='target')
plt.legend()
plt.show()
```

Example Cartpole Exercise

```
## This is course material for Introduction to Modern Artificial Intelligence
## Example code: cartpole_dqn.py
## Author: Allen Y. Yang
##
## (c) Copyright 2020-2024. Intelligent Racing Inc. Not permitted for commercial use

# Please make sure to install openAI gym module
# pip install gym==0.17.3
# pip install pyglet==1.5.29

import gym
import time

# PID control parameters
Kp = 1.0   # Proportional gain
Ki = 0.0   # Integral gain
Kd = 0.5   # Derivative gain

# Environment settings
```

```
EPISODES = 10  # Number of episodes

done = False
while not done:
   env = gym.make('CartPole-v1')
   batch_size = 32

   # env.step(1) goes right
   # env.step(0) goes left
   K_p = 1; K_i = 0; K_d = 0.5;  n=500; speed=1.0; cumulative_error = 0

   for e in range(EPISODES):
       state = env.reset()
       i, previous_state = 0, 0

       for _ in range(n):
           env.render()
           current_error = state[2]
           cumulative_error += current_error
           prev_error = previous_state
           diff = current_error - prev_error

           action = K_p * current_error + K_d * diff + K_i*cumulative_error

           control_signal =  K_p * current_error + K_d * diff + K_i*cumulative_error

           # if control_signal > 0:
           #     action = 1
           # else:
           #     action = 0
           action = 1 if control_signal > 0 else 0

           next_state, _, done, _ = env.step(action)
           # env.step(action) returns four values: next state, reward, a boolean
indicating if the episode has ended, and some extra info
           # here we just use "_" placeholders to ignore the reward and extra info and
unpack the next_state and the done boolean
           env.render()

           previous_state = current_error
           state = next_state
```

```
        if done:
            done = False
            break
```