

# Robotic Navigation and Exploration Lab 1

Week 2: Kinematic Model & Path Tracking Control

Min-Chun Hu [anitahu@cs.nthu.edu.tw](mailto:anitahu@cs.nthu.edu.tw)  
CS, NTHU

# Requirement

- Python 3.X (Suggest to install miniconda/anaconda)
- Numpy
- Opencv-Python

## Issues in MAC

- Miniconda environment path setting for Mac
  - vim ~/.zshrc
  - export PATH=\$PATH:"~/[installation path]/miniconda3/bin"
  - source ~/.zshrc

```
CommandNotFoundError: Your shell has not been properly configured to use 'conda activate'.
```

- source ~/[installation path]/miniconda3/etc/profile.d/conda.sh

```
This application failed to start because no Qt platform plugin could be initialized.  
Reinstalling the application may fix this problem.
```

- pip install opencv-python==4.1.2.30

# Kinematic Model

- There are two files for motion models: [wmr\\_model.py](#) / [bicycle\\_model.py](#). In wheeled mobile robot (WMR) model, we control the car by angular velocity  $\omega$ . In bicycle model, we control the car by steer  $\delta$ .
- Basic Program Flow (main function)

```
car = KinematicModel(<parameters>) # Create the model
car.init_state(<start pose>) # Initialize the pose
while(True):
    car.control(<control parameter>) # Control the model
    car.update() # Update the state
    TODO
```

# Kinematic Model for WMR

State:

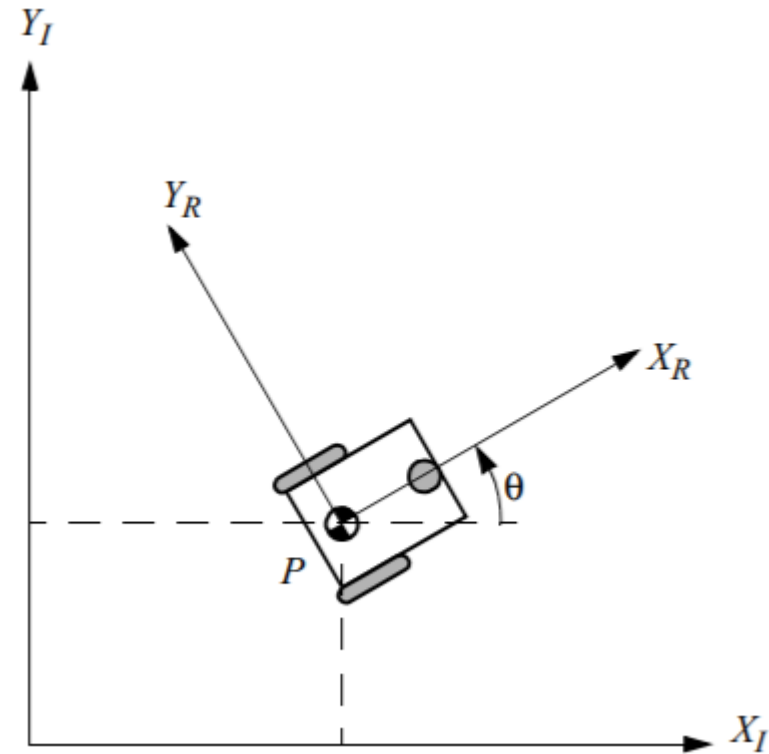
$$\xi_I = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Rotation Matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Kinematic Model:

$$\begin{aligned} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} &= R(\theta)^{-1} \begin{bmatrix} \dot{x}_R \\ \dot{y}_R \\ \dot{\theta} \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ 0 \\ \omega \end{bmatrix} \\ &= \begin{bmatrix} v \cos(\theta) \\ v \sin(\theta) \\ \omega \end{bmatrix} \end{aligned}$$



# Kinematic Model for WMR

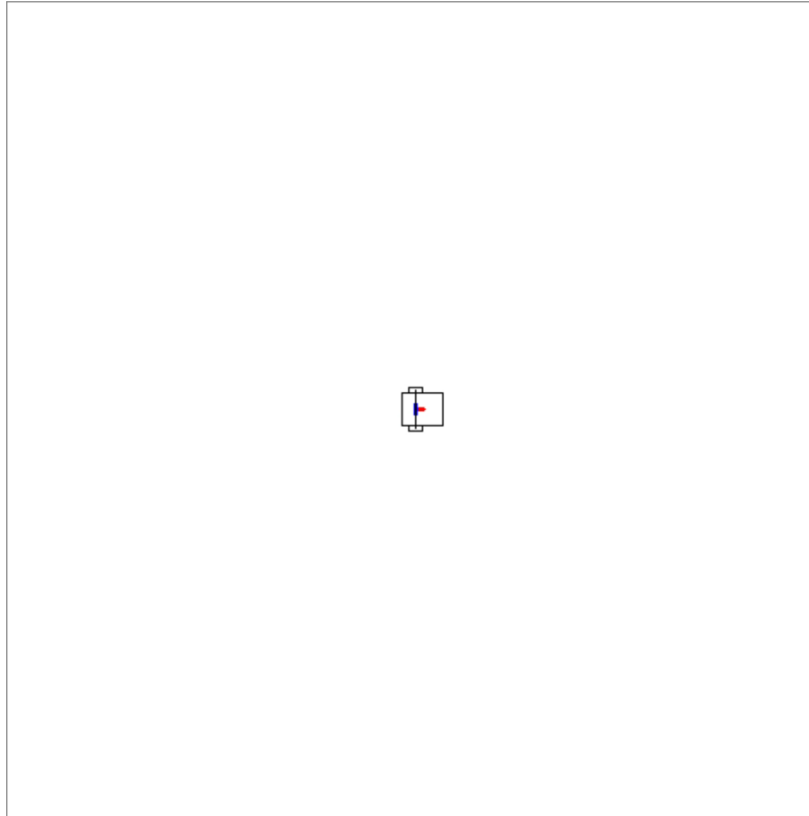
```
def update(self):
    self.v += self.a * self.dt

    # Speed Constrain
    if self.v > self.v_range:
        self.v = self.v_range
    elif self.v < -self.v_range:
        self.v = -self.v_range

    # Motion
    self.x += self.v * np.cos(np.deg2rad(self.yaw)) * self.dt
    self.y += self.v * np.sin(np.deg2rad(self.yaw)) * self.dt
    self.yaw += self.w * self.dt
    self.yaw = self.yaw % 360
    self.record.append((self.x, self.y, self.yaw))
    self._compute_car_box()
```

# Kinematic Model for WMR

- Keyboard Control



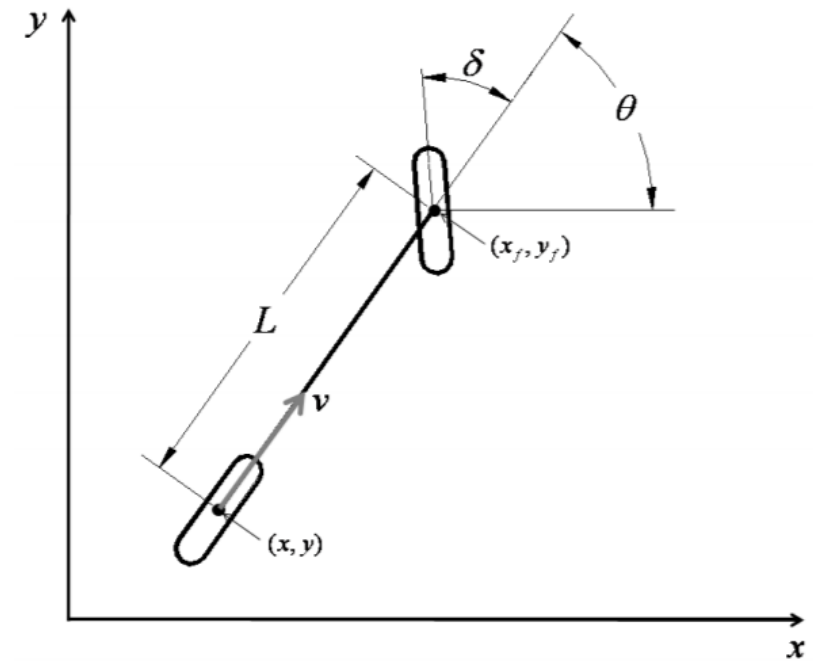
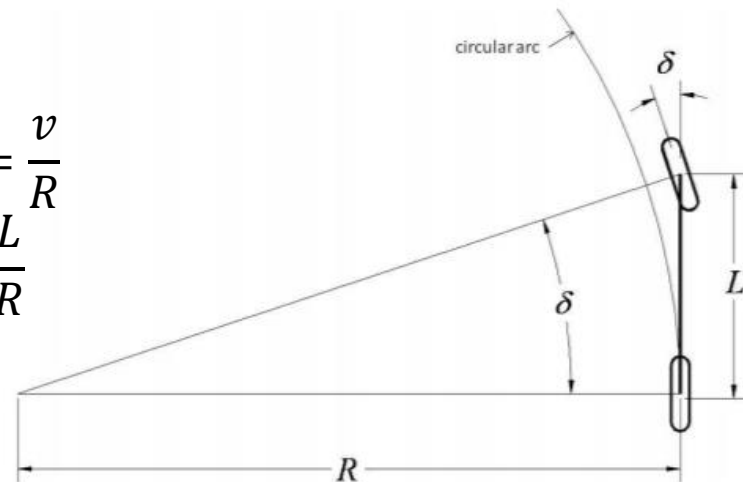
# Kinematic Bicycle Model

- Kinematic Model

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

- Some Property

$$\begin{aligned} R\dot{\theta} &= v \\ \frac{v \tan(\delta)}{L} &= \frac{v}{R} \\ \tan(\delta) &= \frac{L}{R} \end{aligned}$$





# Path Tracking Control

- All path tracking control class ([wmr\\_pid.py](#) / [wmr\\_pure\\_pursuit.py](#) / [bicycle\\_pure\\_pursuit.py](#) / [bicycle\\_stanley.py](#)) share the same abstract function.

**\_\_init\_\_(<parameters>):** Initialize the control algorithm.

**set\_path(path):** Set the path of the control algorithm.

**\_search\_nearest(pos):** Search the nearest points on the path.

**feedback(state):** Compute the feedback control signal (TODO).

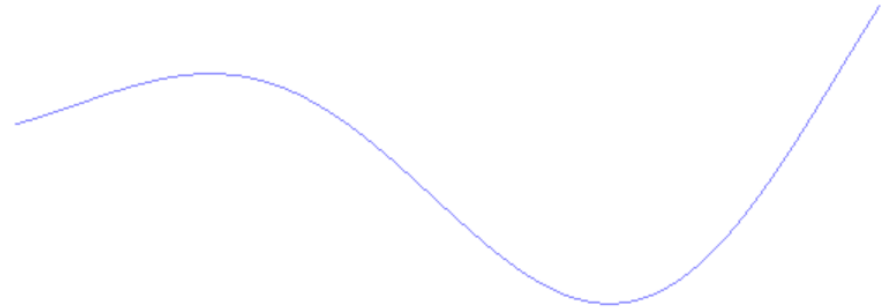
# Path Tracking Control

- Path generator provides two paths, **path1** is a line and **path2** is a curve.

Path 1



Path 2

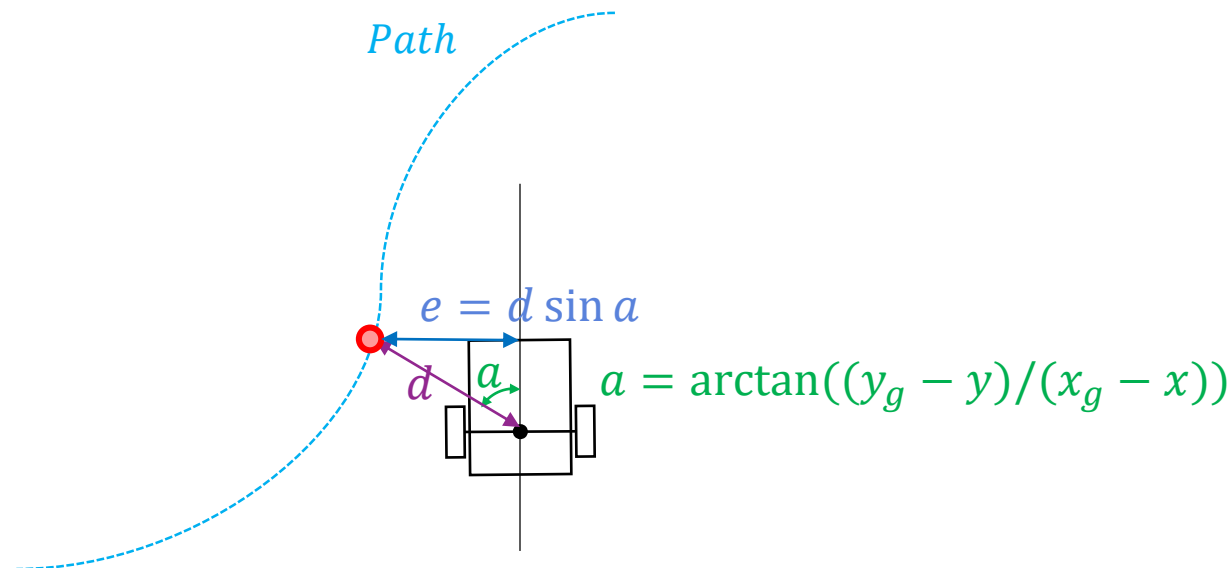
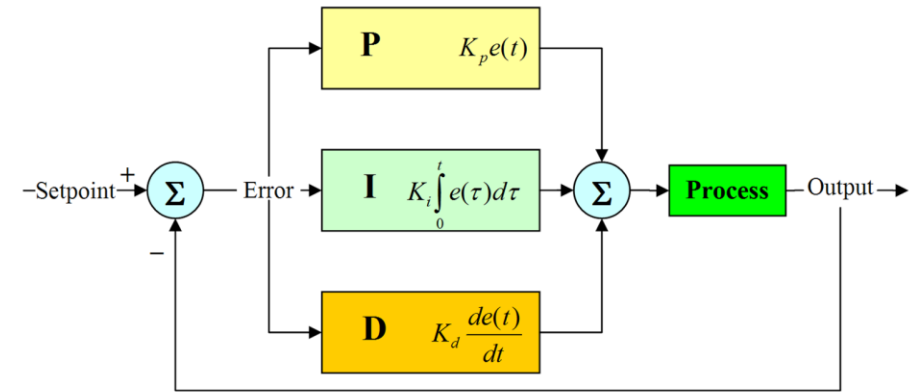


# PID Control

- **P**roportional / **I**ntegral / **D**ifferential Control

Discrete Form :

$$Output = K_p e(t) + K_i \sum_0^t e_t + K_d (e_t - e_{t-1})$$

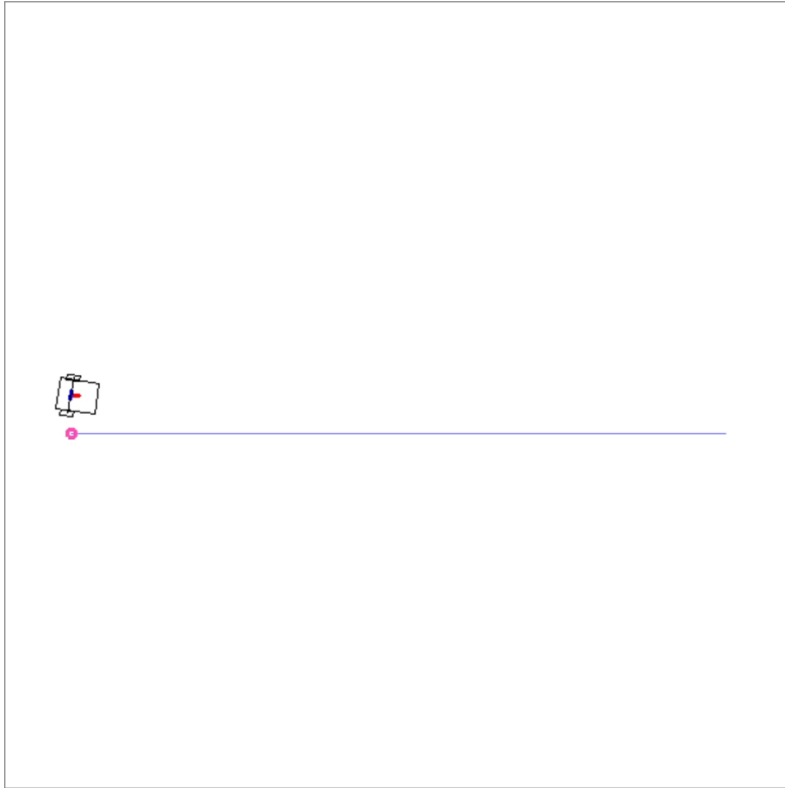


# PID Control

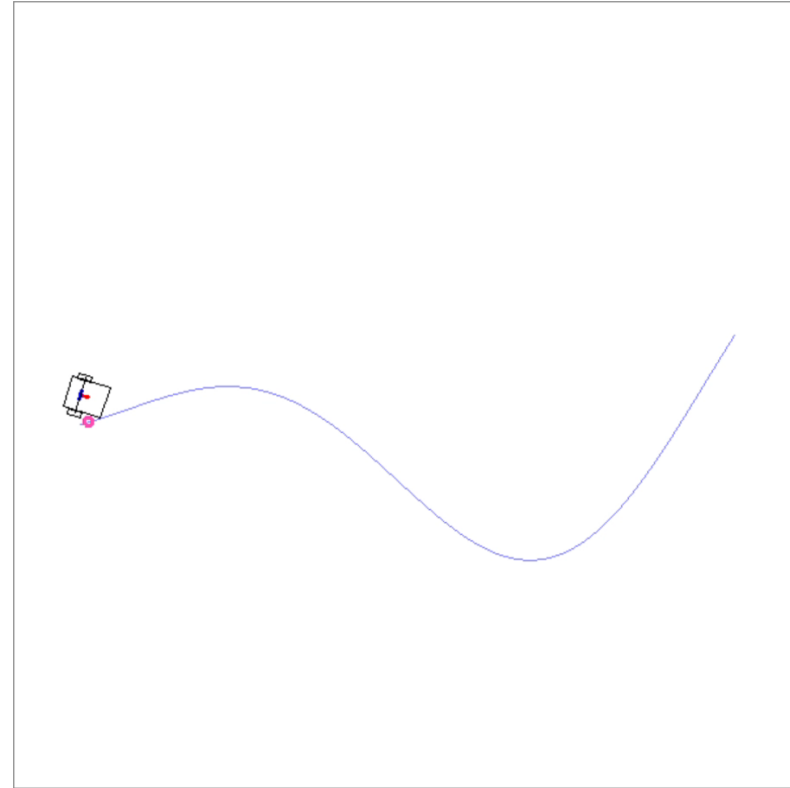
```
def feedback(self, state):  
    # Check Path  
    if self.path is None:  
        print("No path !!")  
        return None, None  
  
    # Extract State  
    x, y, dt = state["x"], state["y"], state["dt"]  
  
    # Search Nearest Target  
    min_idx, min_dist = self._search_nearest((x,y))  
  
    # PID Control  
    ang = np.arctan2(self.path[min_idx,1]-y,  
                    self.path[min_idx,0]-x)  
    ep = min_dist * np.sin(ang)  
    self.acc_ep += dt*ep  
    diff_ep = (ep - self.last_ep) / dt  
    next_w = self.kp*ep + self.ki*self.acc_ep + self.kd*diff_ep  
    self.last_ep = ep  
  
    return next_w, self.path[min_idx]
```

# PID Control

Path 1



Path 2



# Pure Pursuit Control for WMR

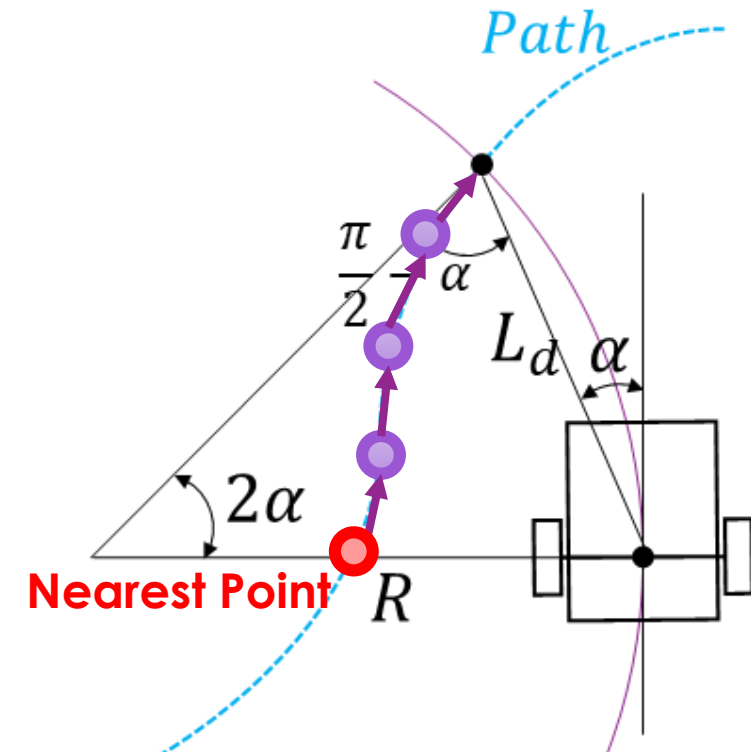
- Concept:
  - Modify the angular velocity to let the center achieve a point on path

$$\alpha = \arctan\left(\frac{y - y_g}{x - x_g}\right) - \theta$$

$$\omega = \frac{2 \sin(\alpha)}{L_d}$$

$L_d = (kv + L_{fc})$ , where  $k, L_{fc}$  are parameters.

1. Set a distance **Ld**.
2. Find the nearest point on the path.
3. Search the following point until the distance of the point larger than or equal to **Ld**.



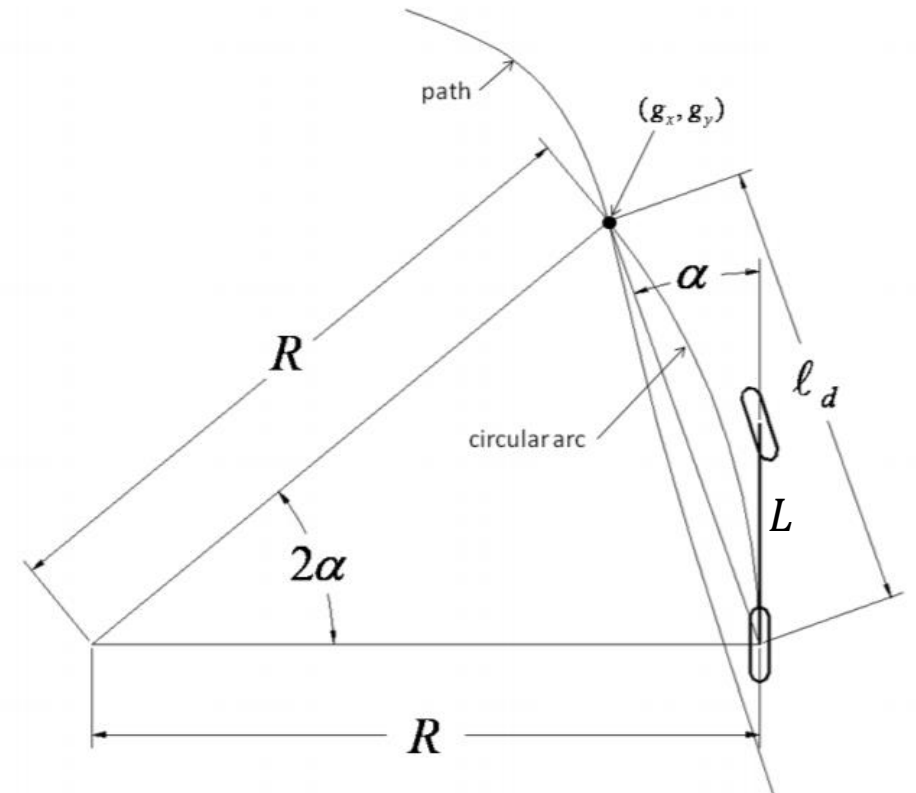
# Pure Pursuit Control for Bicycle Model

- Concept:
  - Control the steer to let the rear wheel achieve a point on the path.

$$\alpha = \arctan\left(\frac{y - y_g}{x - x_g}\right) - \theta$$

$$\delta = \arctan\left(\frac{2L\sin(\alpha)}{L_d}\right)$$

$L_d = (kv + L_{fc})$ , where  $k, L_{fc}$  are parameters.



# Stanley Control

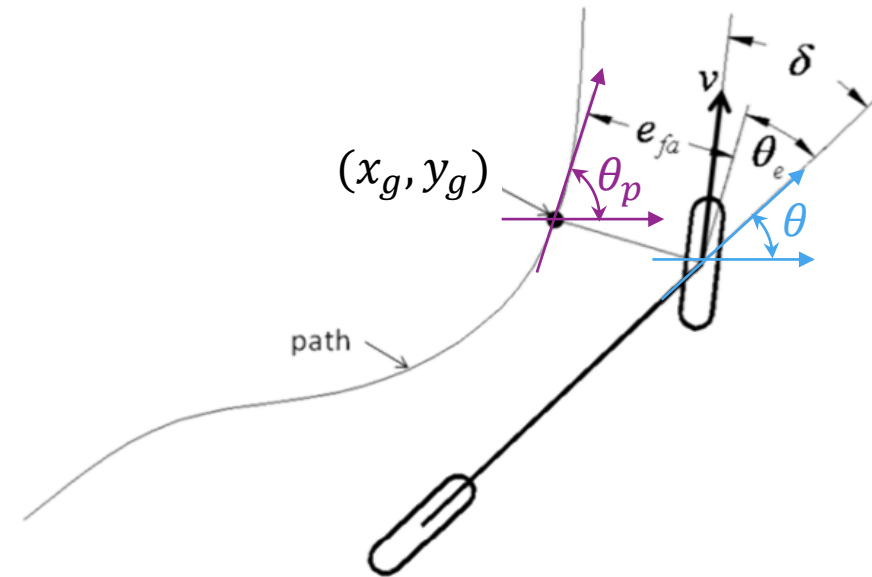
- Concept:
  - Exponential stability for front wheel feedback
- Some Implementation Details

$$\theta_e = \theta_p - \theta$$

$$\dot{e} = v_f \sin(\delta - \theta_e)$$

$$\delta = \arctan\left(-\frac{ke}{v_f}\right) + \theta_e$$

$$e = \begin{bmatrix} x - x_g \\ y - y_g \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta_p) \\ \sin(\theta_p) \end{bmatrix}$$





1. Page 14:

### Pure Pursuit Control for WMR

- Concept:
  - Modify the angular velocity to let the center achieve a point on path

$$\alpha = \arctan\left(\frac{y - y_g}{x - x_g}\right) - \theta$$

$$\omega = \frac{2 \sin(\alpha)}{L_d} \quad \leftarrow \quad \frac{2v \sin(\alpha)}{L_d}$$

$L_d = (kv + L_{fc})$ , where  $k, L_{fc}$  are parameters.

1. Set a distance **Ld**.
2. Find the nearest point on the path.
3. Search the following point until the distance of the point larger than or equal to **Ld**.

14

2. Page 14:

### Pure Pursuit Control for WMR

- Concept:
  - Modify the angular velocity to let the center achieve a point on path

$$\alpha = \arctan\left(\frac{y - y_g}{x - x_g}\right) - \theta \quad \leftarrow \quad \left(\frac{y_g - y}{x_g - x}\right) - \theta$$

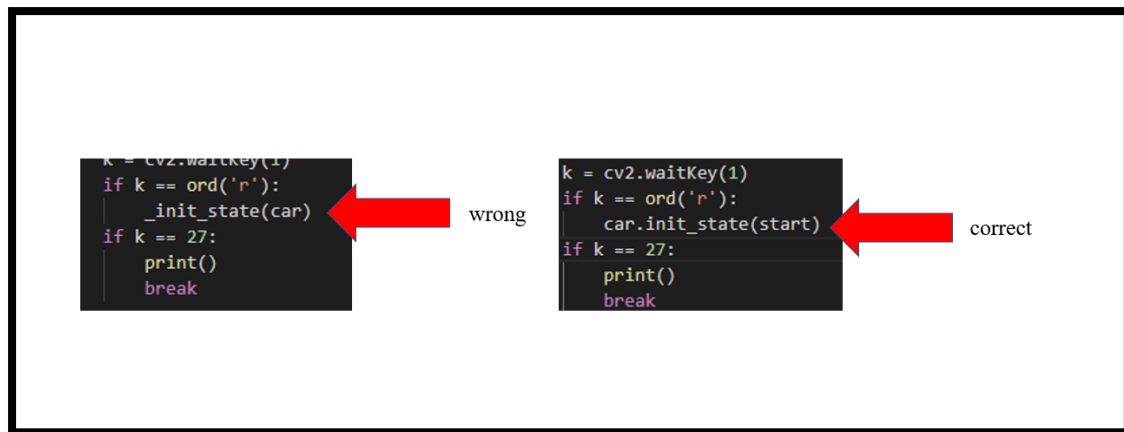
$$\omega = \frac{2 \sin(\alpha)}{L_d}$$

$L_d = (kv + L_{fc})$ , where  $k, L_{fc}$  are parameters.

1. Set a distance **Ld**.
2. Find the nearest point on the path.
3. Search the following point until the distance of the point larger than or equal to **Ld**.

14

3. initial function in all of the code: **initial function usage** in some file is wrong



Start may like this: (x, y, yaw)

```
start = (50, 300, 0)
```

4. Page 16

## Stanley Control


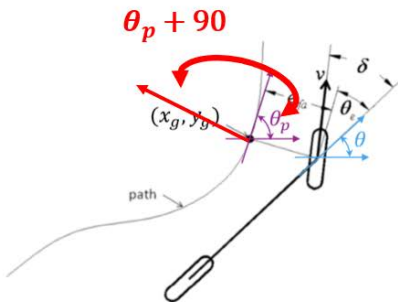
- Concept:
  - Exponential stability for front wheel feedback
- Some Implementation Details

$$\theta_e = \theta_p - \theta$$

$$\dot{e} = v_f \sin(\delta - \theta_e)$$

$$\delta = \arctan\left(-\frac{ke}{v_f}\right) + \theta_e$$

$$e = \begin{bmatrix} x - x_g \\ y - y_g \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta_p) \\ \sin(\theta_p) \end{bmatrix} \Rightarrow \theta_p + 90$$

16

5. Before use the **yaw**, you need to transform the yaw's **degree to radian**

After calculate the angle you need to transform back

Something like this:

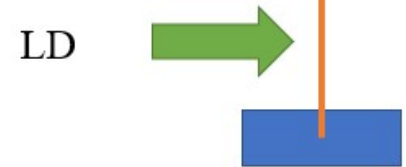
```
alpha = np.arctan2(target[1]-y, target[0]-x) - np.deg2rad(yaw)
next_w = np.rad2deg(2*v*np.sin(alpha) / Ld)
```



**END**  
point

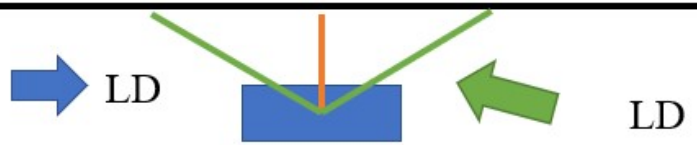
**LD = nearest distance**

Congratulation!  
you find the point that  
**distance equal to LD**



**LD < nearest distance**

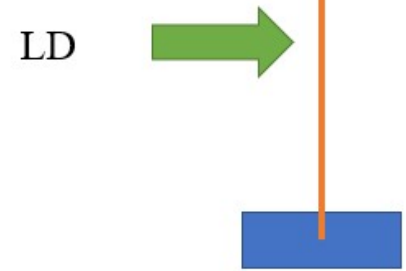
Model will  
**go back!!!**



**END**  
point

**LD > nearest distance**

You can't find a point that  
**distance equal to LD**  
You can just **use the nearest**  
**distance**

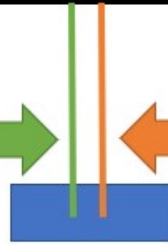


**END**  
point

**LD == nearest distance**

Congratulation!  
you find the point that  
**distance equal to LD**

LD



The nearest distance

**END**  
point

**LD < nearest distance**

LD



Model will  
**go back!!!**



LD

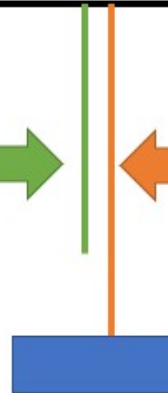
The nearest distance

**END**  
point

**LD > nearest distance**

You can't find a point that  
**distance equal to LD**  
You can just **use the nearest**  
**distance(point)**

LD



The nearest distance

**END**  
point