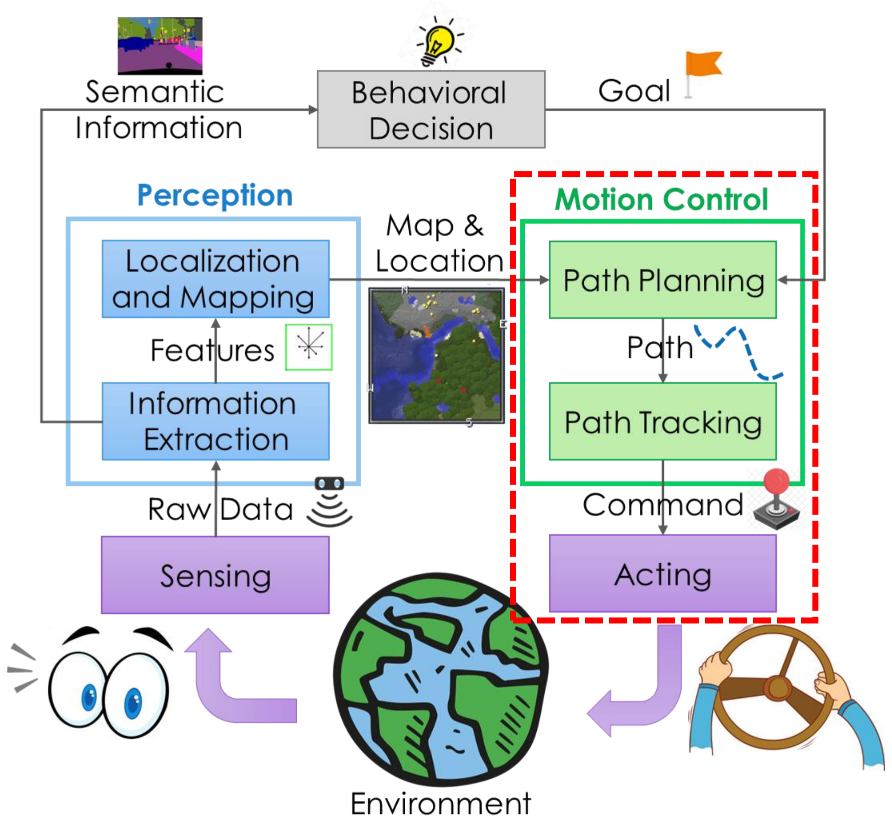
Robotic Navigation and Exploration

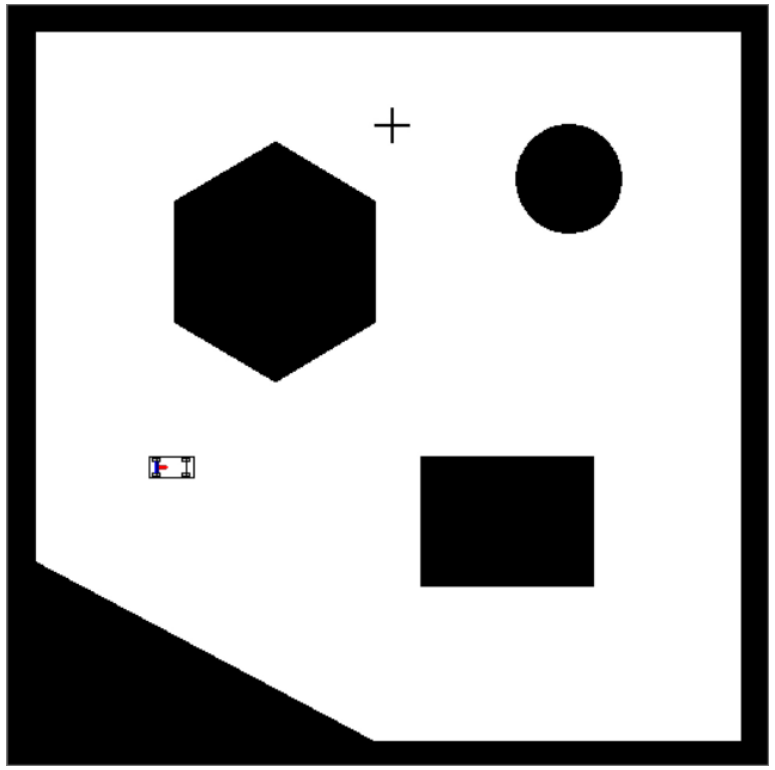
Lab1: Kinematic Model and Path Tracking Control

Min-Chun Hu anitahu@cs.nthu.edu.tw
CS, NTHU

Goal

- In Homework 1, you are going to complete the “Motion Control” part given a known map and localization information. In lab1 and lab2, we will lead you through the control and planning parts, respectively.





Requirement

- Python >= 3.6
- Numpy
- Opencv-Python

Lab Hint

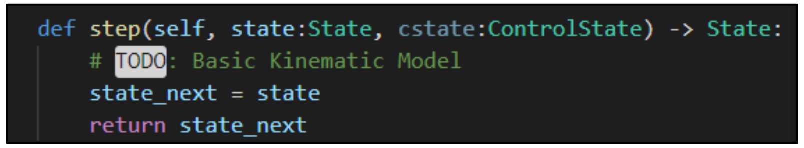
- The pages with the title contains "[Practice]" mean that there are codes that you should complete.
- The pages with the title contains "[Run]" mean that you have to run the code for testing.
- The bottom right will show the path of the codes that are related to that page.
- The codes we give are half-complete, you have to implement the codes after the comment of "TODO".

[Practice] Basic

[Run] Motion

./Simulation/utils.py

7

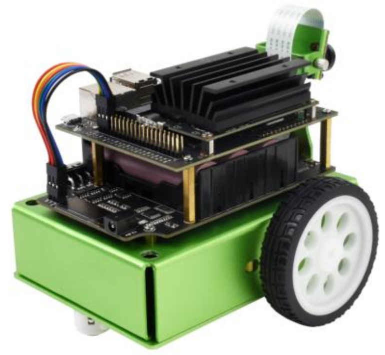


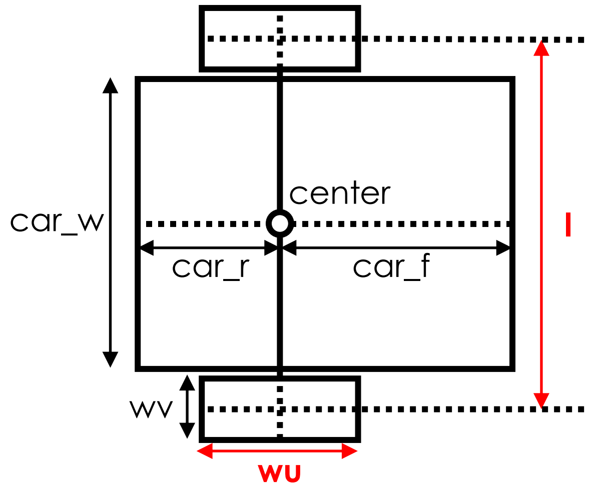
```python
def step(self, state:State, cstate:ControlState) -> State:
    # TODO: Basic Kinematic Model
    state_next = state
    return state_next
```

Kinematic Models

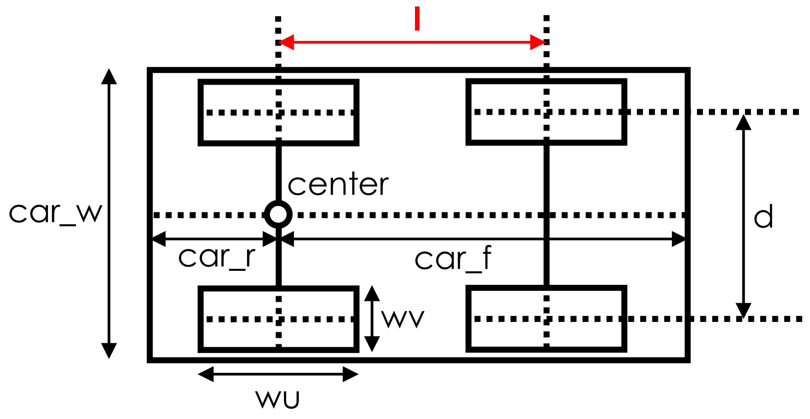
Parameter Settings of Vehicles

Differential Drive





Bicycle



./Simulation/simulator_*.py

States

- Kinematic-Related Physical Parameters
  - x, y, θ, v, ω (position, angle, velocity / angular velocity)
  - (denoted “x”, “y”, “yaw”, “v”, “w” in the codes)

```python
class State:
    def __init__(self, x=0.0, y=0.0, yaw=0.0, v=0.0, w=0.0):
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        self.v = 0.0
        self.w = 0.0
        self.update(x, y, yaw, v, w)

    def update(self, x=None, y=None, yaw=None, v=None, w=None):
        if x is not None:
            self.x = x
        if y is not None:
            self.y = y
        if yaw is not None:
            self.yaw = yaw
        if v is not None:
            self.v = v
        if w is not None:
            self.w = w
```

./Simulation/utils.py

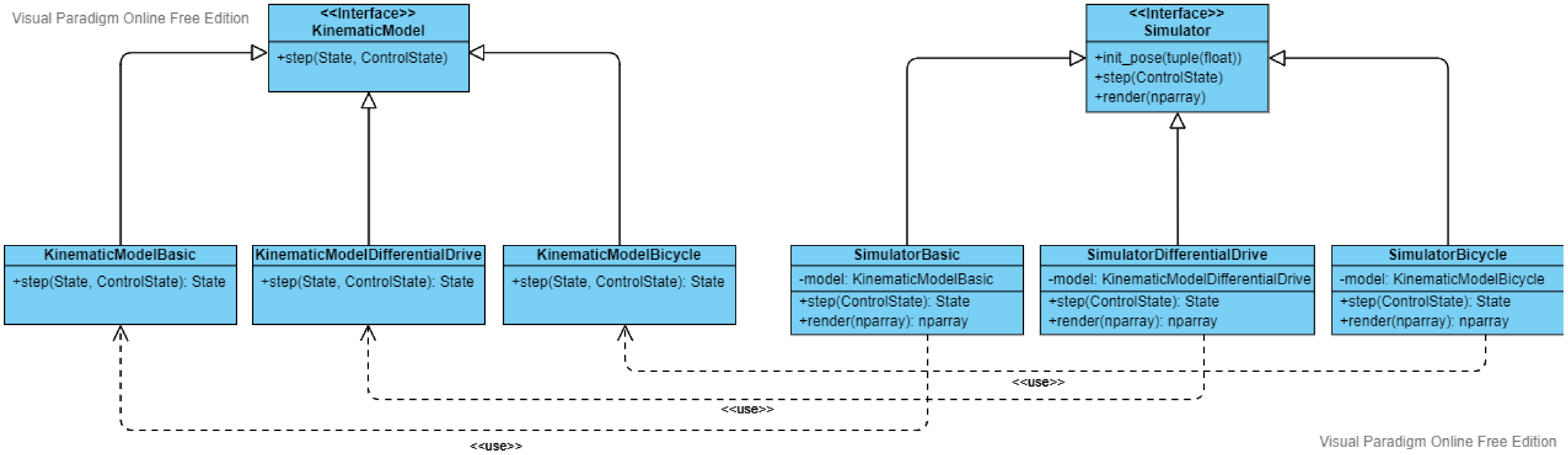
Control States

- Controlling-Related Parameter
  - Basic Model: v, ω (denoted as “v” and “w” in the codes)
  - Differential Drive: $\omega_{left}, \omega_{right}$ (denoted as “lw” and “rw” in the codes)
  - Bicycle Model: a, δ (denoted as “a” and “delta” in the codes)

```python
class ControlState:
    def __init__(self, control_type, *cstate):
        #  Support basic/diff_drive/bicycle
        self.control_type = control_type
        try:
            if control_type == "basic":
                self.v = cstate[0]
                self.w = cstate[1]
            elif control_type == "diff_drive":
                self.lw = cstate[0]
                self.rw = cstate[1]
            elif control_type == "bicycle":
                self.a = cstate[0]
                self.delta = cstate[1]
            else:
                raise NameError("Unknown control type!!")
        except NameError:
            raise
```

./Simulation/utils.py

Class Architecture for Simulation


Visual Paradigm Online Free Edition
<<Interface>>
KinematicModel
+step(State, ControlState)
<<Interface>>
Simulator
+init_pose(tuple(float))
+step(ControlState)
+render(nparray)
KinematicModelBasic
+step(State, ControlState): State
KinematicModelDifferentialDrive
+step(State, ControlState): State
KinematicModelBicycle
+step(State, ControlState): State
SimulatorBasic
-model: KinematicModelBasic
+step(ControlState): State
+render(nparray): nparray
SimulatorDifferentialDrive
-model: KinematicModelDifferentialDrive
+step(ControlState): State
+render(nparray): nparray
SimulatorBicycle
-model: KinematicModelBicycle
+step(ControlState): State
+render(nparray): nparray
<<use>>
<<use>>
<<use>>
Visual Paradigm Online Free Edition

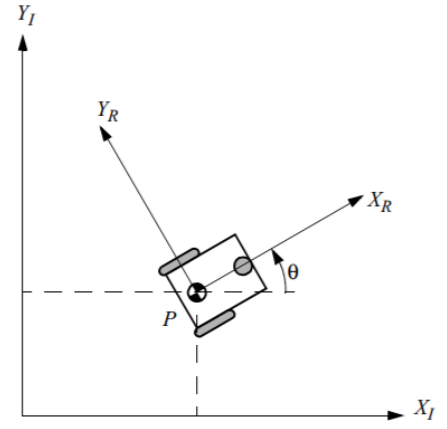
./Simulation/*.py

[Practice] Basic Kinematic Model

```
class KinematicModelBasic(KinematicModel):
    def __init__(self, dt):
        # Simulation delta time
        self.dt = dt

    def step(self, state:State, cstate:ControlState) -> State:
        v = cstate.v
        w = cstate.w
        x = state.x + v * np.cos(np.deg2rad(state.yaw)) * self.dt
        y = state.y + v * np.sin(np.deg2rad(state.yaw)) * self.dt
        yaw = (state.yaw + state.w * self.dt) % 360
        state_next = State(x, y, yaw, v, w)
        return state_next
```

Beware of the converting between degree and radian.



$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = R(\theta)^{-1}\begin{bmatrix}\dot{x}_R\\ \dot{y}_R\\ \dot{\theta}\end{bmatrix}$$

$$= \begin{bmatrix} cos\theta & -sin\theta & 0\\ sin\theta & cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}v\\ 0\\ \omega\end{bmatrix}$$

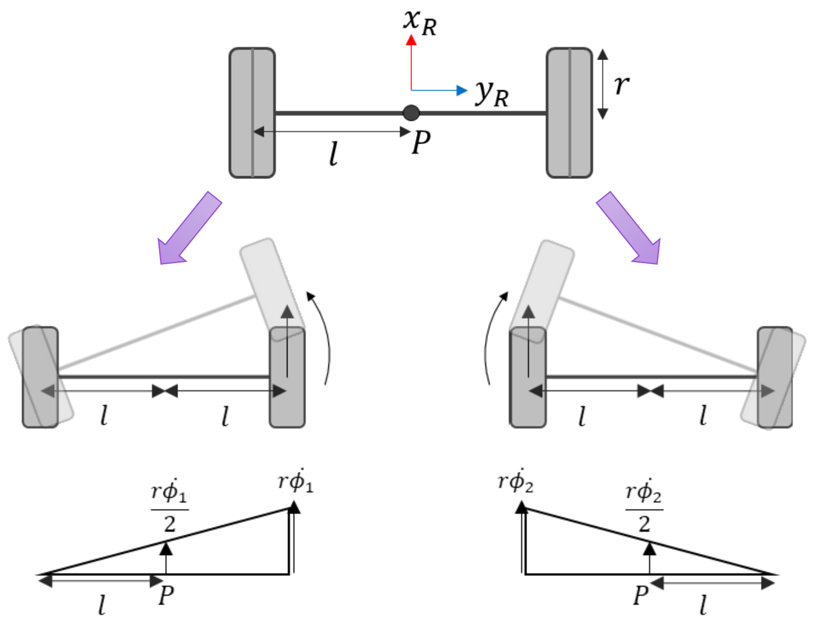
$$= \begin{bmatrix} v\cos(\theta)\\ v\sin(\theta)\\ \omega\end{bmatrix}$$

./Simulation/kinematic_basic.py

[Practice] Differential Drive Kinematic Model

```python
class KinematicModelDifferentialDrive(KinematicModel):
    def __init__(self, r, l, dt):
        # Simulation delta time
        self.r = r
        self.l = l
        self.dt = dt

    def step(self, state:State, cstate:ControlState) -> State:
        x1dot = self.r*np.deg2rad(cstate.rw) / 2
        w1 = np.rad2deg(self.r*np.deg2rad(cstate.rw) / (2*self.l))
        x2dot = self.r*np.deg2rad(cstate.lw) / 2
        w2 = np.rad2deg(self.r*np.deg2rad(cstate.lw) / (2*self.l))
        v = x1dot + x2dot
        w = w1 - w2
        x = state.x + v * np.cos(np.deg2rad(state.yaw)) * self.dt
        y = state.y + v * np.sin(np.deg2rad(state.yaw)) * self.dt
        yaw = (state.yaw + w * self.dt) % 360
        state_next = State(x, y, yaw, v, w)
        return state_next
```



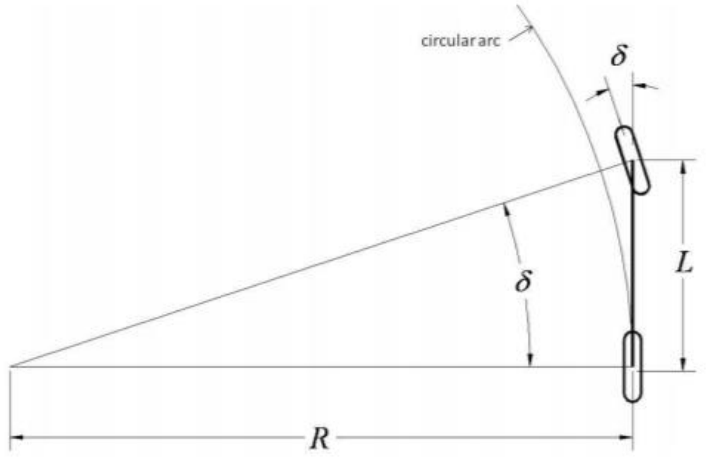
$$\begin{cases} v = \dfrac{r\dot{\phi}_1}{2} + \dfrac{r\dot{\phi}_2}{2} \\ \omega = \dfrac{r\dot{\phi}_1}{2l} - \dfrac{r\dot{\phi}_2}{2l} \end{cases}$$

./Simulation/kinematic_differential_drive.py

[Practice] Bicycle Kinematic Model

```python
class KinematicModelBicycle(KinematicModel):
    def __init__(self, l, dt):
        # Distance from center to wheel
        self.l = l
        # Simulation delta time
        self.dt = dt

    def step(self, state:State, cstate:ControlState) -> State:
        v = state.v + cstate.a*self.dt
        w = np.rad2deg(state.v / self.l * np.tan(np.deg2rad(cstate.delta)))
        x = state.x + v * np.cos(np.deg2rad(state.yaw)) * self.dt
        y = state.y + v * np.sin(np.deg2rad(state.yaw)) * self.dt
        yaw = (state.yaw + w * self.dt) % 360
        state_next = State(x, y, yaw, v, w)
        return state_next
```



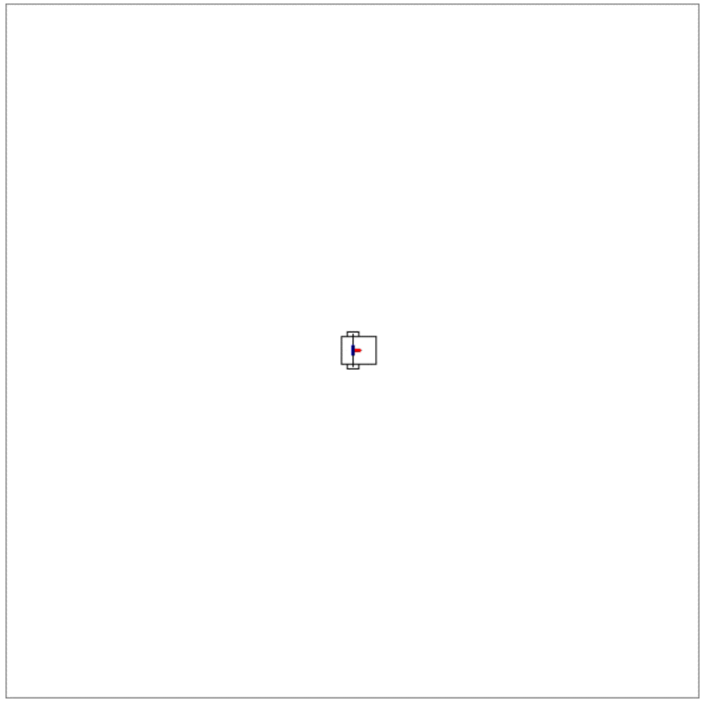
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \frac{\tan(\delta)}{L} \end{bmatrix} v$$

./Simulation/kinematic_bicycle.py

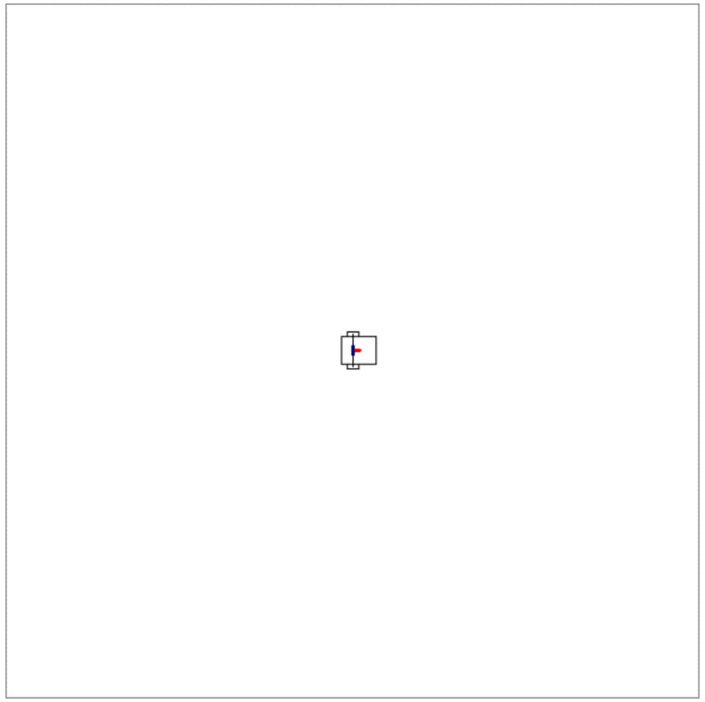
[Run] Motion Model

python 01_motion_model.py -s [basic/diff_drive/bicycle]

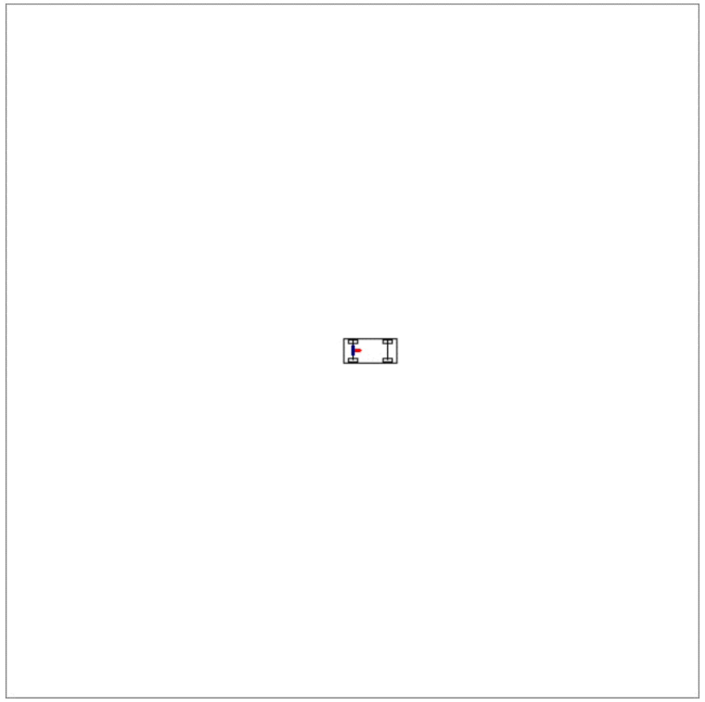
basic



diff_drive



bicycle



./01_motion_model.py

Path Tracking Control

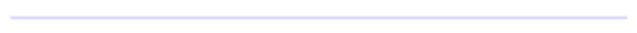
Path Example

- We provides two paths for testing the tracking algorithm, **path1** is a line and **path2** is a curve.

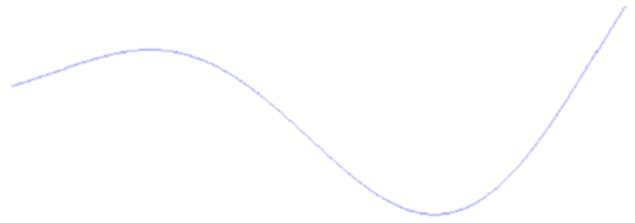
```python
def path1():
    cx = np.arange(0, 500, 1) + 50
    cy = [270 for ix in cx]
    cyaw = [0 for ix in cx]
    ccurv = [0 for ix in cx]
    path = np.array([(cx[i],cy[i],cyaw[i],ccurv[i]) for i in range(len(cx))])
    return path

def path2(p1 = 80.0):
    cx = np.arange(0, 500, 1) + 50
    cy = [np.sin(ix / p1) * ix / 4.0 + 270 for ix in cx]
    diff1 = [(np.cos(ix/p1)/p1*ix + np.sin(ix/p1))/4.0 for ix in cx]
    diff2 = [(-np.sin(ix/p1)/(p1**2)*ix + np.cos(ix/p1)/p1 + np.cos(ix/p1)/p1)/4.0 for ix in cx]
    cyaw = [np.rad2deg(np.arctan2(d1,1)) for d1 in diff1]
    ccurv = [np.abs(diff2[i])/np.power((1+diff1[i]**2),3/2) for i in range(len(cx))]
    path = np.array([(cx[i],cy[i],cyaw[i],ccurv[i]) for i in range(len(cx))])
    return path
```

Path 1



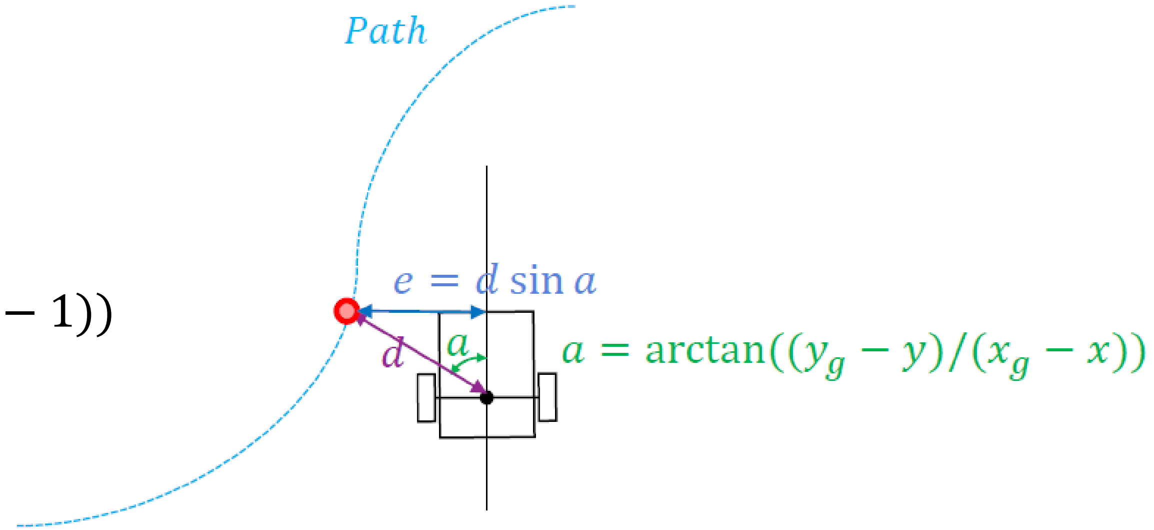
Path 2



./PathTracking/utils.py

[Practice] PID Control

$$Output = K_p e(t) + K_i \sum_{0}^{t} e_t + K_d(e(t) - e(t-1))$$



Basic Model

```
min_idx, min_dist = utils.search_nearest(self.path, (x,y))
target = self.path[min_idx]
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, target
```

Bicycle Model

```
min_idx, min_dist = utils.search_nearest(self.path, (x,y))
target = self.path[min_idx]
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_delta = self.kp*ep + self.ki*self.acc_ep +
self.kd*diff_ep
self.last_ep = ep
return next_delta, target
```

./PathTracking/controller_pid_*.py

[Practice] Pure Pursuit Control for Basic Model

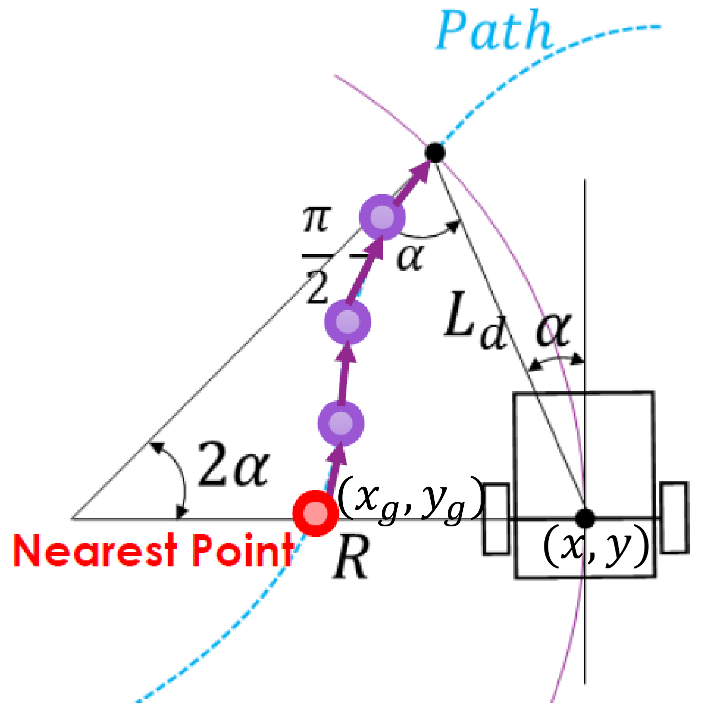
- Concept:
  - Modify the angular velocity to let the center achieve a point on path

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

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

$L_d = k * v + L_{fc}$, where k, L_{fc} are parameters.

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



./PathTracking/controller_pure_pursuit_basic.py

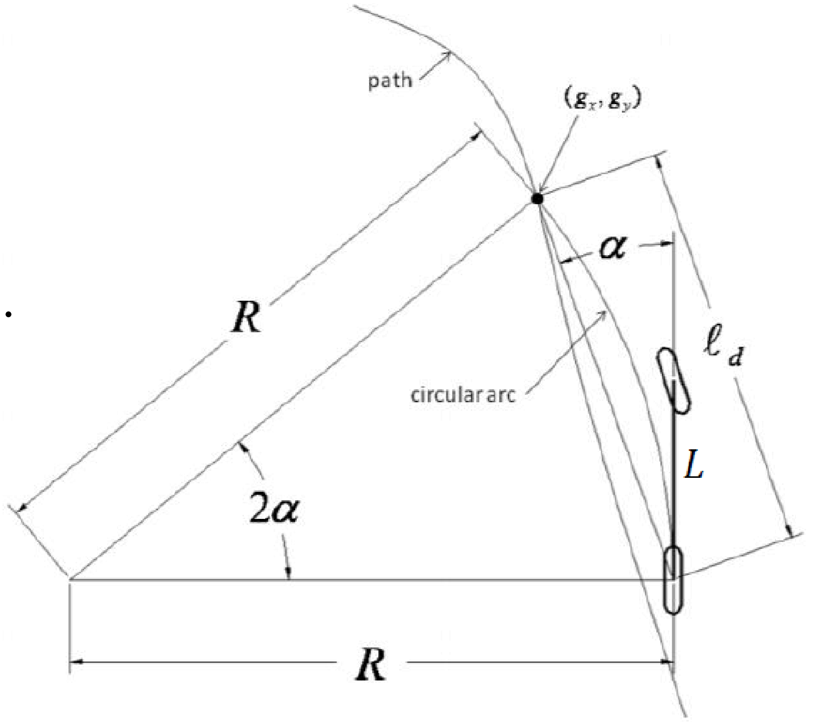
[Practice] Pure Pursuit Control for Bicycle Model

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

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

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

$L_d = k * v + L_{fc}$, where k, L_{fc} are parameters.

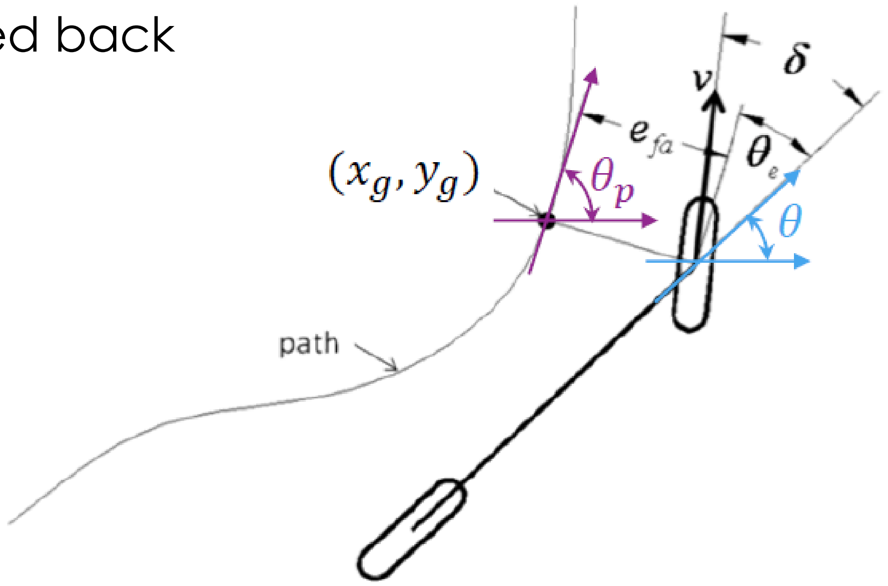


./PathTracking/controller_pure_pursuit_bicycle.py

[Practice] Stanley Control for Bicycle Model

- Concept:
  - Exponential stability for front wheel feed back
- Some Implementation Details

$$\theta_e = \theta_p - \theta$$
$$\dot{e} = v_f \sin(\delta - \theta_e)$$
$$\delta = \arctan(-\frac{ke}{v_f}) + \theta_e$$
$$e = \begin{bmatrix} x - x_g \\ y - y_g \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \cos(\theta_p + 90) \\ \sin(\theta_p + 90) \end{bmatrix}$$



- Hint: Beware to transform the angle into the boundary of -180~+180

./PathTracking/controller_stanley_bicycle.py

[Practice] LQR Control (Bonus)

- We have already completed the part of DARE solving.
- Following the steps:
  - Construct the matrix A, B, X of the linear approximation model.

$$Ax + Bu = \underset{\textbf{A}}{\begin{bmatrix} 1 & dt & 0 & 0 \\ 0 & 0 & v & 0 \\ 0 & 0 & 1 & dt \\ 0 & 0 & 0 & 0 \end{bmatrix}} \underset{\textbf{x}}{\begin{bmatrix} e \\ \dot{e} \\ \theta \\ \dot{\theta} \end{bmatrix}} + \underset{\textbf{B}}{\begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{v}{L} \end{bmatrix}} \underset{\textbf{u}}{\delta} \text{ (Bicycle Model)}$$

  - Solve DARE and get the matrix P of the value function.

$$P = Q + A^T PA - A^T PB(R + B^T PB)^{-1} B^T PA$$

  - Compute the optimal control.

$$u_t^* = -(R + B^T P_{t+1} B)^{-1} B^T P_{t+1} A x_t$$

./PathTracking/controller_lqr_*.py

[Practice] Control of Differential Drive

- We only implement the controller for basic and bicycle kinematic model. The control of the differential drive can be simply modified by the controller of basic model.

$$\dot{\phi}_2 = \left(v - \frac{r\dot{\phi}_1}{2}\right)\frac{2}{r} = \frac{2v}{r} - \dot{\phi}_1$$

$$\omega = \frac{r\dot{\phi}_1}{2l} - \frac{r\left(\frac{2v}{r} - \dot{\phi}_1\right)}{2l} = \frac{r\dot{\phi}_1 - v}{l}$$

$$\dot{\phi}_1 = \frac{v}{r} + \frac{\omega l}{r}$$

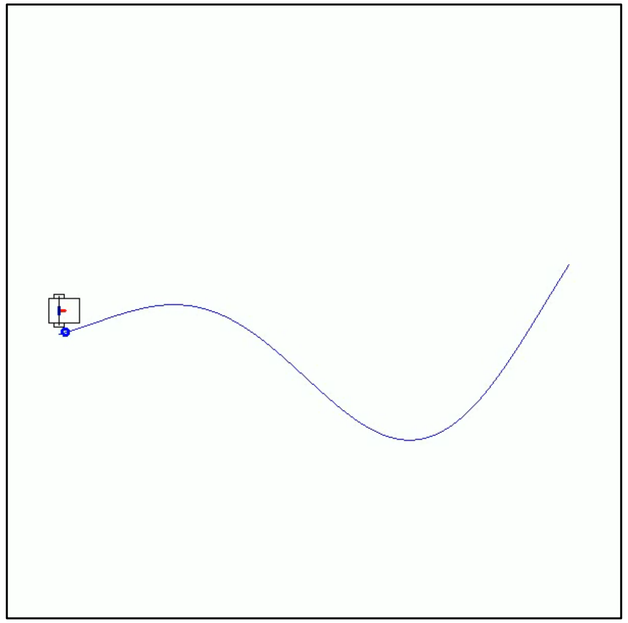
$$\dot{\phi}_2 = \frac{v}{r} - \frac{\omega l}{r}$$

./02_path_tracking.py

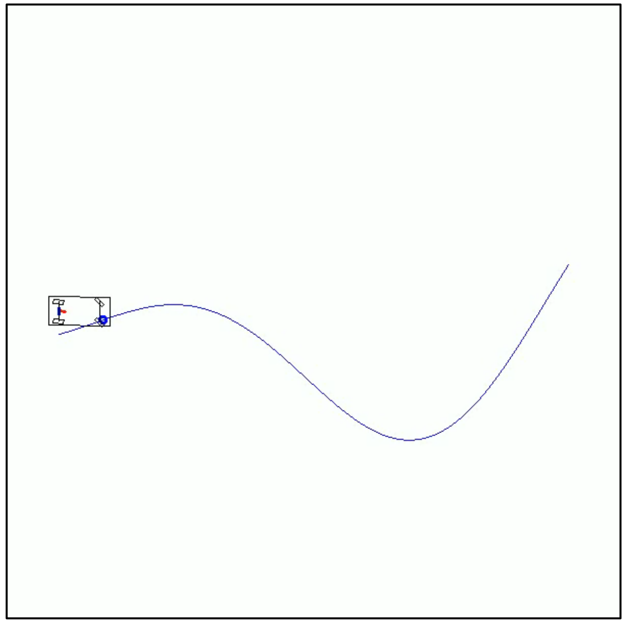
[Run] Path Tracking

```
python 02_path_tracking.py -s [basic/diff_drive/bicycle] (simulator)
                           -c [pid/pure_pursuit/stanley/lqr] (controller)
                           -t [1/2] (path type)
```

basic + pure_pursuit



bicycle + stanley



./02_path_tracking.py

Remind

- Check if you complete the “TODO” in the following files:
  - ./Simulation/kinematic_basic.py
  - ./Simulation/kinematic_differential_drive.py
  - ./Simulation/kinematic_bicycle.py
  - ./PathTracking/controller_pid_basic.py
  - ./PathTracking/controller_pid_bicycle.py
  - ./PathTracking/controller_pure_pursuit_basic.py
  - ./PathTracking/controller_pure_pursuit_bicycle.py
  - ./PathTracking/controller_stanley_bicycle.py
  - ./02_path_tracking.py
  - ./PathTracking/controller_lqr_basic.py (bonus)
  - ./PathTracking/controller_lqr_bicycle.py (bonus)

Q&A