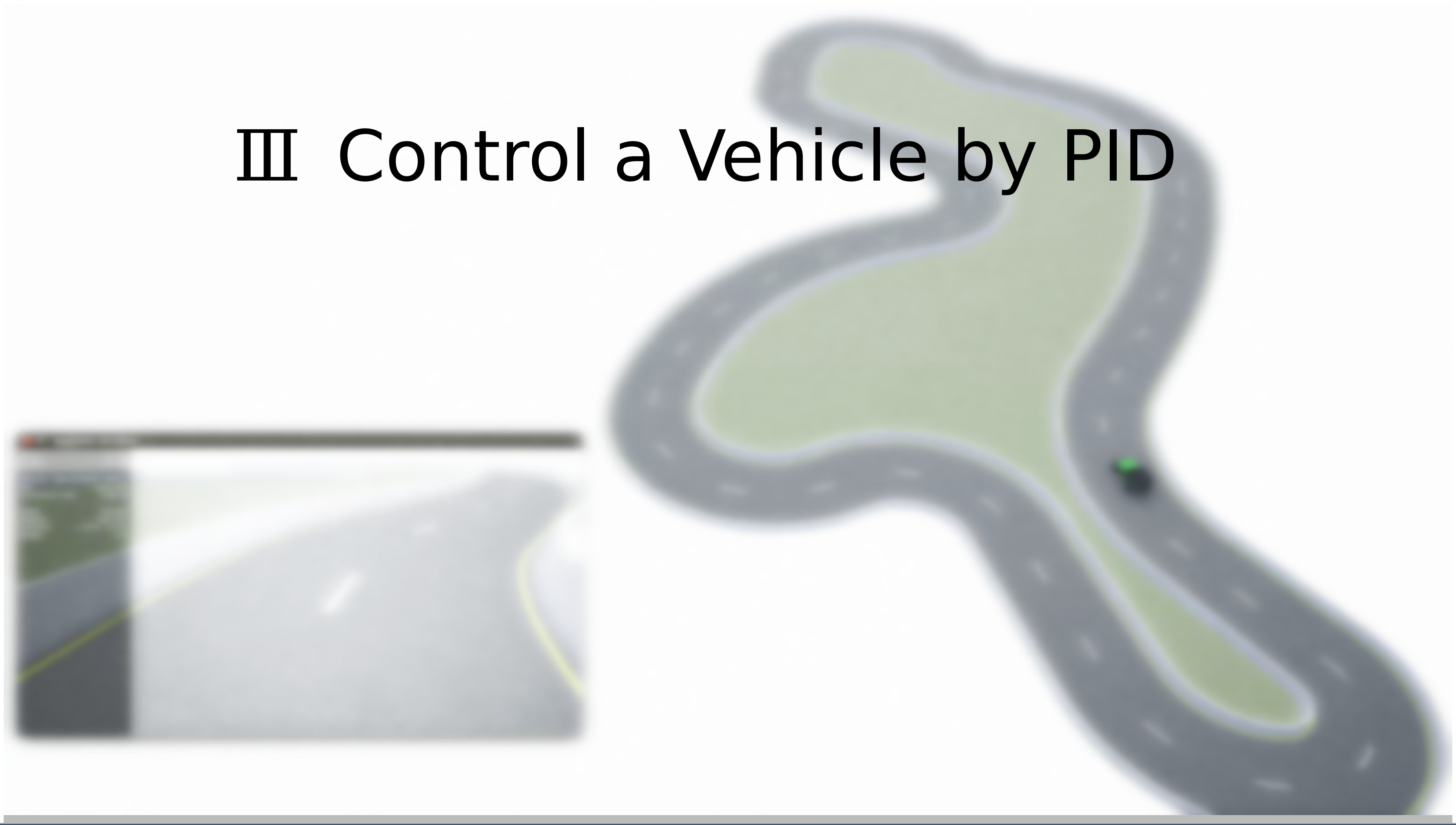
Ⅲ Control a Vehicle by PID



Lesson 1 - 2

MoCAD Experimental Course Schedule (Carla-python)

	Course title	Course contents	Projects
D1	Environment setup	1. Course introduction 2. Python Environment anaconda 3. Carla quick start installation and linux build 4. Spawning a vehicle in Carla with your own map (RoadRunner) 5. Carla core concepts	
D2	Running a vehicle by keyboard and collecting data	1. Control a vehicle by apply_control method and keyboard 2. Attach a rgb-image sensor on the vehicle 3. Simulation time-step 4. Try different sensors: RGB-camera, Depth-camera, Lidar, Obstacle ...	**Simple: Sensors** Control a vehicle by keyboard and use Carla python API to collect data from different sensors.
D3	Running a vehicle by PID control	1. Mapping and waypoint 2. Global path planning 3. Local planning 4. PID controller	
D4	Running a vehicle by behavior clone	1. Collecting data 2. Supervised learning 3. Training Neural Network 4. Control a vehicle by the trained NN	**Intermediate: Leader-follower instance** Use the keyboard to control the leader (first vehicle) and the second vehicle follows the leader by PID or behavior clone.
D5	Running a vehicle by reinforcement learning I	1. Introduce the reinforcement learning and DQN 2. Create an Carla environment 3. Building a DQN network 4. Python multi-threading 5. Training the network, agent interacts with Carla environment 6. Control a vehicle by the trained NN	
D6	Running a vehicle by reinforcement learning II	1. Continue action 2. Multi-class regression problem 3. Future work	**Complex: Racing** Use all the knowledge you have learned to control the vehicle so that it can complete a lap on the race road as quickly as possible.

Study Carla

Use Carla

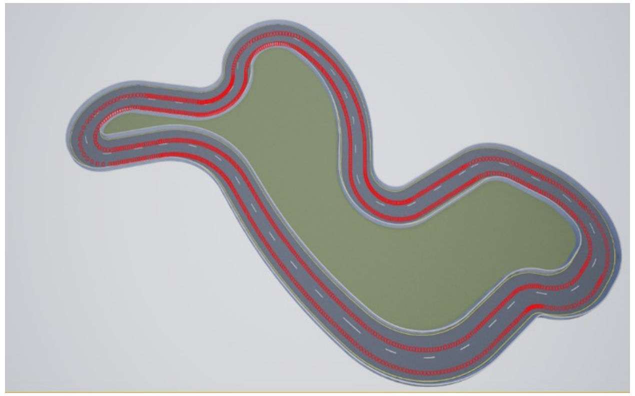
Lesson 1

- Summary

① RoadRunner Plugin and import map;
② Server – Client ；
③ Core concepts;
  a) Map waypoints;
  b) Vehicle spawn;
  c) Move the vehicle based on the waypoints;

- Demo

Carla Simulator



Waypoints

Vehicle spawn

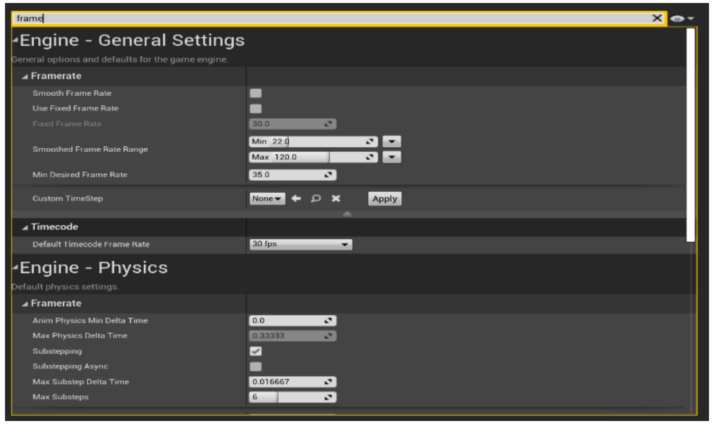
Lesson 2

- Summary

① Spawn sensors (rgb-camera);
② Server – client FPS;
③ **Synchronous or asynchronous;**

- Demo

rgb-camera



Server – client FPS

Sensors

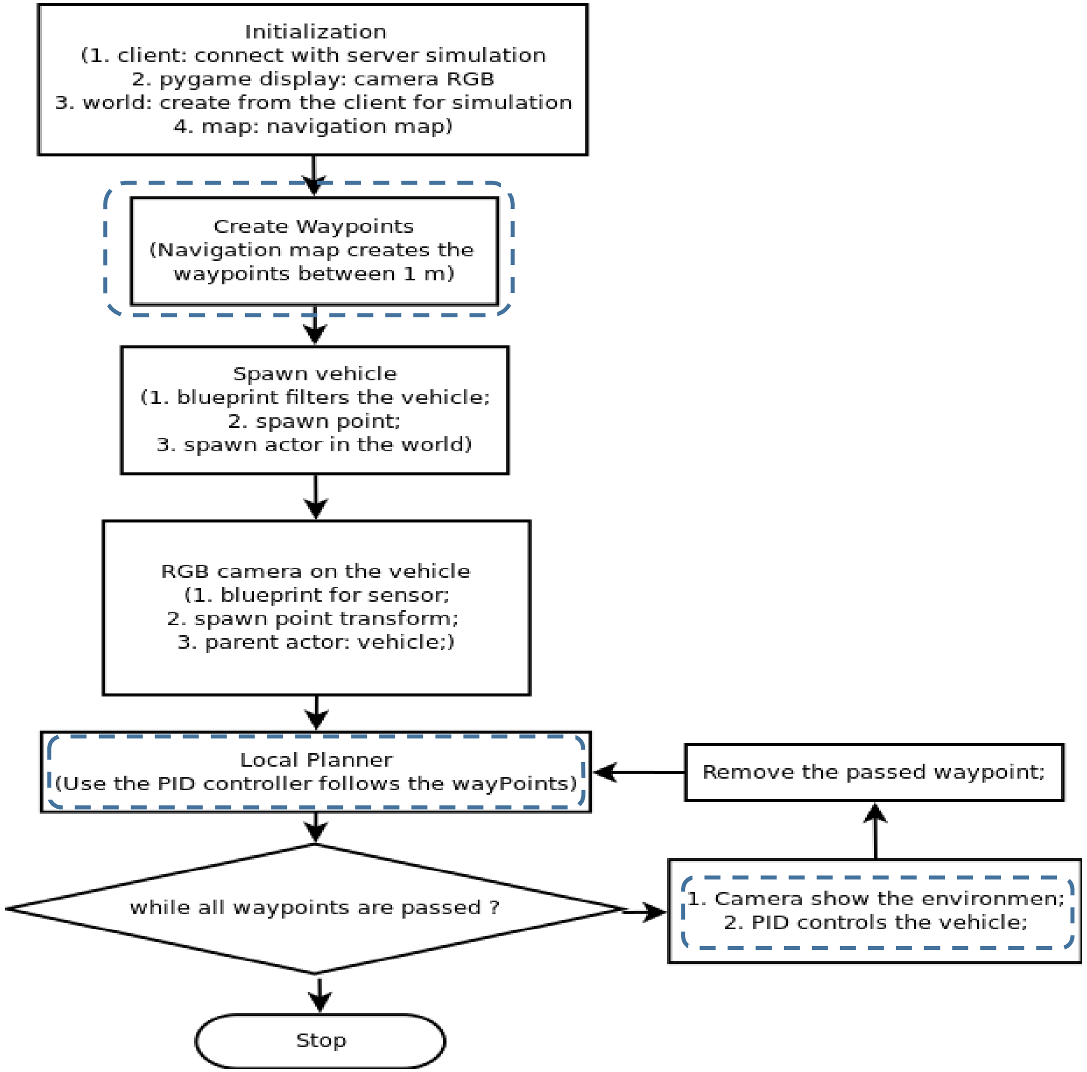
Outline

1. Mapping (UE4 RoadRunner)
2. Draw waypoints and Global path planning
3. Local planning step by step
4. PID controller

1. Mapping, draw waypoints

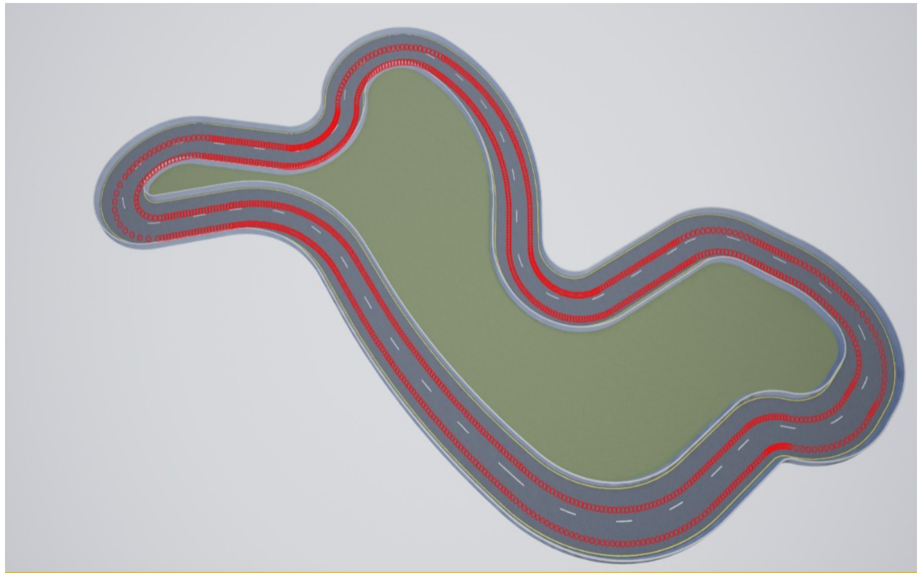
- UE4_RoadRunner_Carla (Day 1)

Control a vehicle by PID



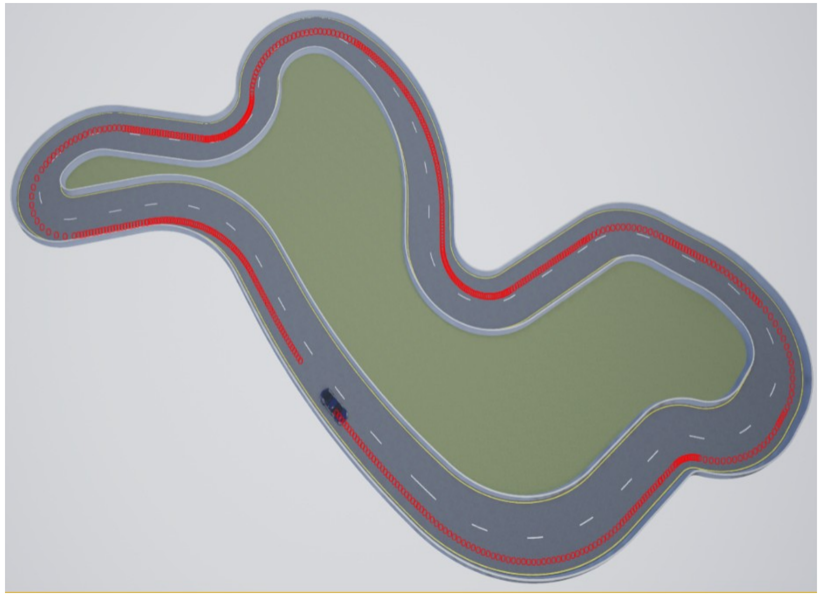
2. Draw waypoints and Global path planning

- A Global path



All waypoints

```
#########################
# --- plan the route --- #
#########################
# all waypoints based on the distance
map = world.get_map()
waypoints = map.generate_waypoints(distance=0.5)
# the waypoints on the samw lane
left_lane_w, right_lane_w = list(), list()
for idx, w in enumerate(waypoints):
    if idx % 2 == 0:
        right_lane_w.append(w)
    else:
        left_lane_w.append(w)
# planning route
planning_route = right_lane_w[-90:] + right_lane_w[:500]
for w in planning_route:
    world.debug.draw_string(w.transform.location, 'O', draw_shadow=False,
                                  color=carla.Color(r=255, g=0, b=0), life_time=60.0,
                                  persistent_lines=True)
```



A Global path

2. Draw waypoints and Global path planning

- PythonAPI

carla.Map

Class containing the road information and waypoint managing. Data is retrieved from an OpenDRIVE file that describes the road. A query system is defined which works hand in hand with carla.Waypoint to translate geometrical information from the .xodr to natural world points. CARLA is currently working with OpenDRIVE 1.4 standard.

Instance Variables

- name (*str*)
  The name of the map. It corresponds to the .umap from Unreal Engine that is loaded from a CARLA server, which then references to the .xodr road description.

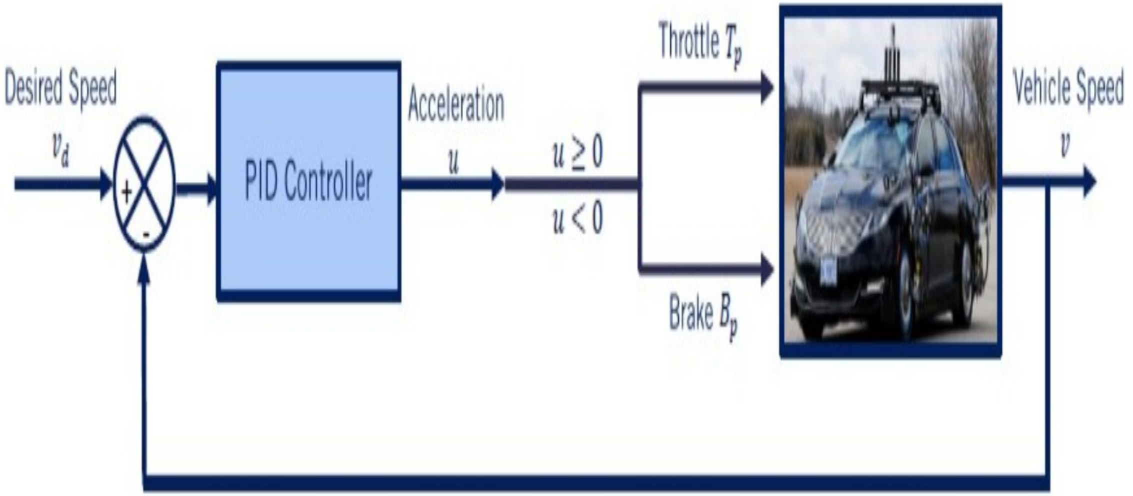
Methods

- __init__(self, name, xodr_content)
  Constructor for this class. Though a map is automatically generated when initializing the world, using this method in no-rendering mode facilitates working with an .xodr without any CARLA server running.
  - **Parameters:**
    - `name` (*str*) – Name of the current map.
    - `xodr_content` (*str*) – .xodr content in string format.
  - **Return:** *list(carla.Transform)*
- **generate_waypoints(self, distance)**
  Returns a list of waypoints with a certain distance between them for every lane and centered inside of it. Waypoints are not listed in any particular order. Remember that waypoints closer than 2cm within the same road, section and lane will have the same identificator.
  - **Parameters:**
    - `distance` (*float - meters*) – Approximate distance between waypoints.
  - **Return:** *list(carla.Waypoint)*

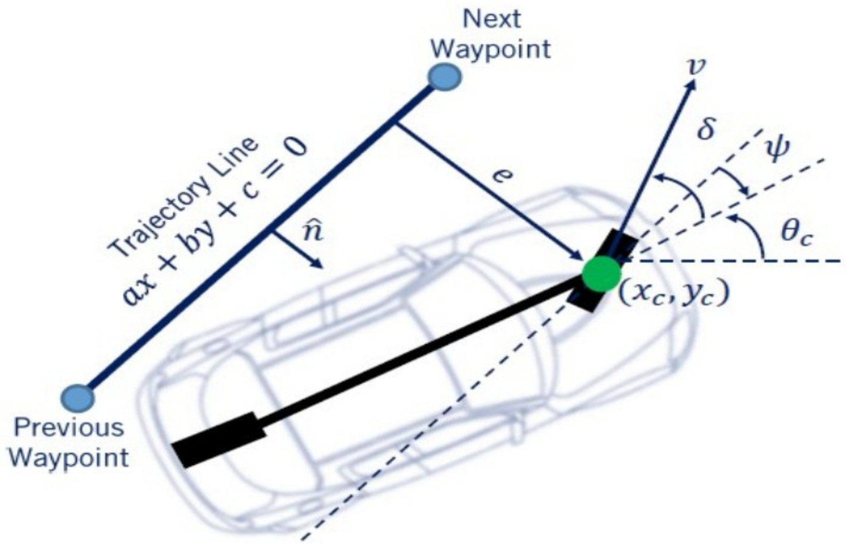
3. Local Planning

- Use PID controller to achieve the local planning

1. Longitudinal control (throttle and brake: velocity)



2. Lateral control (steer: orientation)



3. Local Planning

- Local planning： current waypoint and next waypoint

```python
def run_step(self, target_speed=None):
    """
    Execute one step of local planning which involves
    running the longitudinal and lateral PID controllers to
    follow the waypoints trajectory.
    """

    if target_speed is not None:
        self._target_speed = target_speed
    else:
        # self._target_speed = self._vehicle.get_speed_limit()  # the landmarkType.MaximumSpeed
        # modify the target speed, control the vehicle easily
        self._target_speed = 20

    # Buffering the waypoints
    if not self._waypoint_buffer:
        for i in range(self._buffer_size):
            if self.waypoints_queue:
                self._waypoint_buffer.append(self.waypoints_queue.popleft())
            else:
                break

    # Current vehicle waypoint
    self._current_waypoint = self._map.get_waypoint(self._vehicle.get_location())
```

1. Target： velocity and position

2. Current vehicle： velocity and position

2. Draw waypoints and Global path planning

- PythonAPI

Carla.Actor

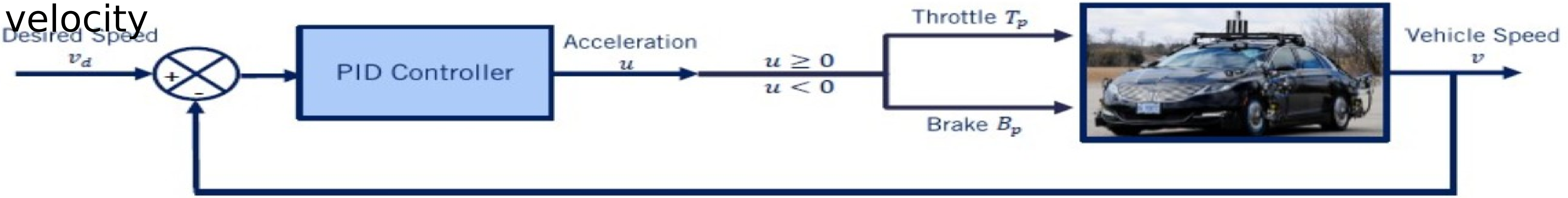
- **get_location**(self)
  Returns the actor's location the client recieved during last tick. The method does not call the simulator.
  - **Return:** *carla.Location* – *meters*
  - **Setter:** *carla.Actor.set_location*

Carla.Map

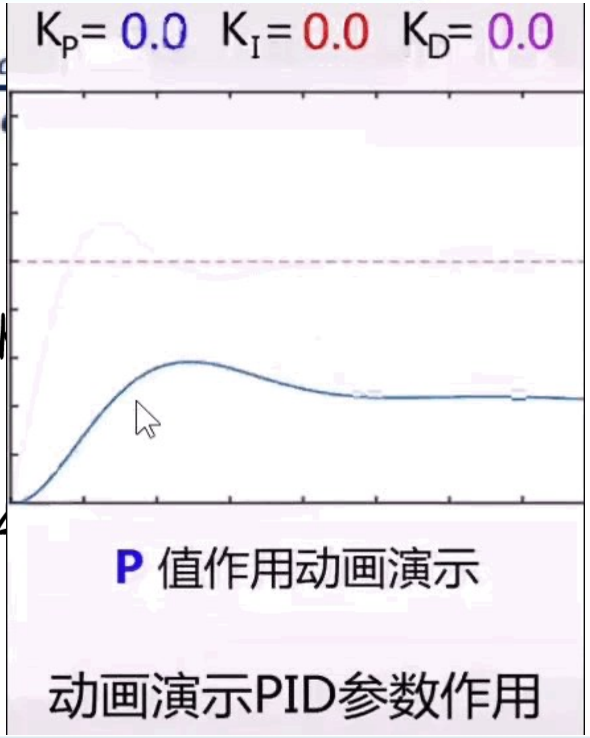
- **get_waypoint**(self, location, project_to_road=True, lane_type=carla.LaneType.Driving)
  Returns a waypoint that can be located in an exact location or translated to the center of the nearest lane. Said lane type can be defined using flags such as `LaneType.Driving & LaneType.Shoulder`. The method will return **None** if the waypoint is not found, which may happen only when trying to retrieve a waypoint for an exact location. That eases checking if a point is inside a certain road, as otherwise, it will return the corresponding waypoint.
  - **Parameters:**
    - `location` (*carla.Location* – *meters*) – Location used as reference for the carla.Waypoint.
    - `project_to_road` (*bool*) – If **True**, the waypoint will be at the center of the closest lane. This is the default setting. If **False**, the waypoint will be exactly in `location`. **None** means said location does not belong to a road.
    - `lane_type` (*carla.LaneType*) – Limits the search for nearest lane to one or various lane types that can be flagged.
  - **Return:** *carla.Waypoint*

4. PID controller

- PID （Proportion， Integration, Differentiation） controller： Longitudinal control velocity



$$u = K_P(v_d - v) + K_I \int_0^t (v_d - v)dt + K_D \frac{d(v_d}{}$$



① : Proportion Example (water level Control):

$U = k_p * Error$ $(k_p = 0.5)$

round 1 : Error = 0.8m

round 2 : Error = 0.4m

② ： Integration

$U = k_p * Error + k_i * \int Error\, dt$

Note : ① Make the control sy

③ ： Integration

$U = k_p * Error + k_i * \int Error\, dt + k_d * [Error(t) - Error(t-1)]$

4. PID controller

- PID controller： Longitudinal control velocity

No Carla PythonAPI

Numpy np.clip [-1.0, 1.0]

```
def _pid_control(self, target_speed, current_speed):
    """
    Estimate the throttle/brake of the vehicle based on the PID equations

        :param target_speed:  target speed in Km/h
        :param current_speed: current speed of the vehicle in Km/h
        :return: throttle/brake control
    """

    error = target_speed - current_speed
    self._error_buffer.append(error)

    if len(self._error_buffer) >= 2:
        _de = (self._error_buffer[-1] - self._error_buffer[-2]) / self._dt
        _ie = sum(self._error_buffer) * self._dt
    else:
        _de = 0.0
        _ie = 0.0

    return np.clip((self._k_p * error) + (self._k_d * _de) + (self._k_i * _ie), -1.0, 1.0)
```

$$u = K_P(v_d - v) + K_I\int_0^t (v_d - v)dt + K_D\frac{d(v_d - v)}{dt}$$

4. PID controller

- PID controller： Lateral control orientation

```python
def _pid_control(self, waypoint, vehicle_transform):
    """
    Estimate the steering angle of the vehicle based on the PID equations

        :param waypoint: target waypoint
        :param vehicle_transform: current transform of the vehicle
        :return: steering control in the range [-1, 1]
    """
    v_begin = vehicle_transform.location
    v_end = v_begin + carla.Location(x=math.cos(math.radians(vehicle_transform.rotation.yaw)),
                                     y=math.sin(math.radians(vehicle_transform.rotation.yaw)))

    v_vec = np.array([v_end.x - v_begin.x, v_end.y - v_begin.y, 0.0])
    w_vec = np.array([waypoint.transform.location.x -
                      v_begin.x, waypoint.transform.location.y -
                      v_begin.y, 0.0])
    _dot = math.acos(np.clip(np.dot(w_vec, v_vec) /
                             (np.linalg.norm(w_vec) * np.linalg.norm(v_vec)), -1.0, 1.0))

    _cross = np.cross(v_vec, w_vec)

    if _cross[2] < 0:
        _dot *= -1.0

    self._e_buffer.append(_dot)
    if len(self._e_buffer) >= 2:
        _de = (self._e_buffer[-1] - self._e_buffer[-2]) / self._dt
        _ie = sum(self._e_buffer) * self._dt
    else:
        _de = 0.0
        _ie = 0.0

    return np.clip((self._k_p * _dot) + (self._k_d * _de) + (self._k_i * _ie), -1.0, 1.0)
```

1.Orientation： vehicle and target waypoint

2. Proportional

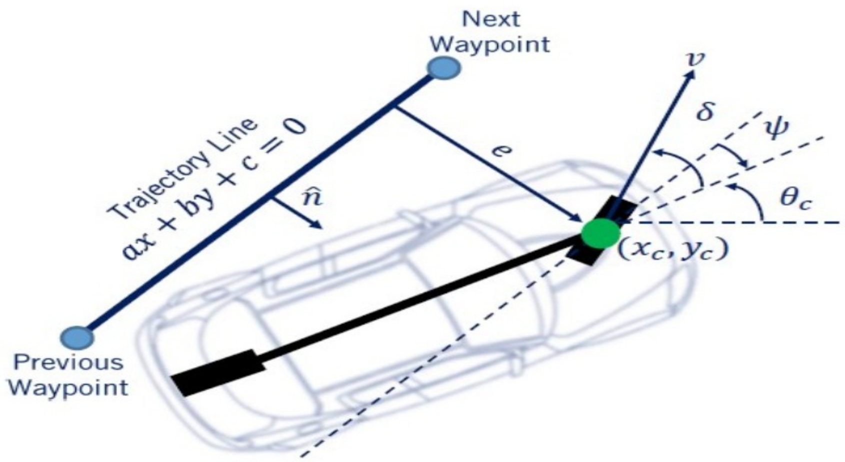
3. Integral and derivative

5. Better performance

1. Mapping， more waypoints
2. Limit max speed
3. Client FPS, Computer performance
4. Other controller

5. Better performance -- Stanley controller

- Stanley controller： Steer



- Cross track error:

$$e = \frac{ax_c + by_c + c}{\sqrt{a^2 + b^2}}$$

- Cross track steering:

$$\tan^{-1}\left(\frac{ke}{v}\right)$$

- Heading error:

$$\psi = \tan^{-1}\left(\frac{-a}{b}\right) - \theta_c$$

- Total steering input:

$$\delta = \psi + \tan^{-1}\left(\frac{ke}{v}\right)$$

```
# --- 1. calculate heading error --- #
first_location = waypoints[0].transform.location
last_location = waypoints[-1].transform.location
yaw_path = np.arctan2(last_location.y - first_location.y, last_location.x - first_location.x)
yaw_diff = yaw_path - vehicle_transform.rotation.yaw
yaw_diff = yaw_diff * 2 * np.pi / 360
if yaw_diff > np.pi:
    yaw_diff -= 2 * np.pi
if yaw_diff < - np.pi:
    yaw_diff += 2 * np.pi


# --- 2. calculate crosstrack error --- #
vehicle_location = vehicle_transform.location
current_xy = np.array([vehicle_location.x, vehicle_location.y])
min_error = 1000
min_waypoints = None
for w in waypoints:
    target_xy = np.array([w.transform.location.x, w.transform.location.y])
    c_error = np.sum((current_xy-target_xy)**2)
    if c_error < min_error:
        min_error = c_error
        min_waypoints = target_xy
crosstrack_error = min_error
yaw_cross_track = np.arctan2(vehicle_location.y - waypoints[0].transform.location.y,
                             vehicle_location.x - waypoints[0].transform.location.x)
```

Control a Vehicle by PID

- Design a new global path;
- Show the vehicle state information;
- Try different controller Stanley, PI, PD;

Carla over !

(J3016) Automation Levels

SAE (J3016) Automation Levels[57]

SAE Level	Name	Narrative definition		Execution of steering and acceleration/ deceleration	Monitoring of driving environment	Fallback performance of dynamic driving task	System capability (driving modes)
Human driver monitors the driving environment							
0	No Automation	The full-time performance by the human driver of all aspects of the dynamic driving task, even when "enhanced by warning or intervention systems"		Human driver	Human driver	Human driver	n/a
1	Driver Assistance	The driving mode-specific execution by a driver assistance system of "either steering or acceleration/deceleration"	using information about the driving environment and with the expectation that the human driver performs all remaining aspects of the dynamic driving task	Human driver and system			Some driving modes
2	Partial Automation	The driving mode-specific execution by one or more driver assistance systems of *both steering and acceleration/deceleration*		System			
Automated driving system monitors the driving environment							
3	Conditional Automation	The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task	with the expectation that the *human driver will respond appropriately to a request to intervene*	System	System	Human driver	Some driving modes
4	High Automation		*even if a human driver does not respond appropriately to a request to intervene*			System	Many driving modes
5	Full Automation		*under all roadway and environmental conditions* that can be managed by a human driver				All driving modes

Try more

- **Assisted driving**

① L0 □ Keyboard;
② L1 □ keyboard + Lane invasion detector □ Lane line keeping;
③ L2 □ keyboard + Obstacle detector □ Safe distance;

- **Autonomous driving**

① L3 □ PID + Human driver □ … ;
② L4 □
③ L5 □