

# Control

en CARLA

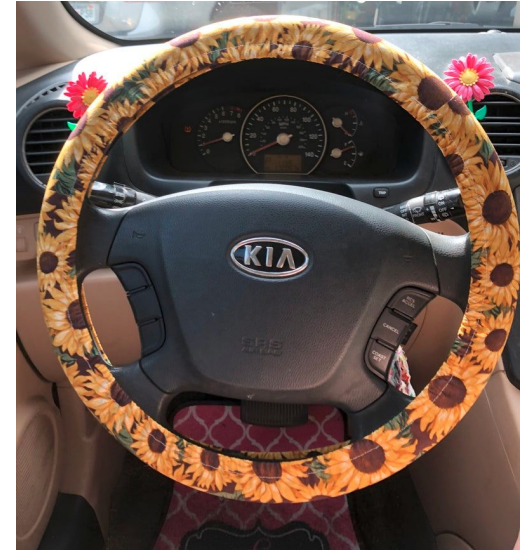
by. Rafael Peralta Blanco

# El control

Se refiere a modificar los actuadores del vehículo para alcanzar un estado de referencia.

Los actuadores del vehículo son:

- Acelerador.
- Freno.
- Giro del volante.



# El control

Entonces surgen las preguntas...

¿Cómo obtener un estado de referencia en CARLA?

¿Cómo modificar los actuadores del vehículo en CARLA?

# El mapa en CARLA

# El mapa

En CARLA se han implementado 8 ciudades.

Y podemos cambiar de una ciudad a otra.

En el archivo  
'helpers.py' se  
encuentra la  
función  
'load\_map'.

```
def load_map(env, mapa):  
    env.world = env.client.load_world(mapa)
```




NO  
CODE  
TIME

Por ejemplo un  
nombre del mapa  
puede ser  
'Town04'

# El mapa

Para este ejemplo, utilizaremos el mapa 'Town04'

```
def setup(env):  
    if env.world.get_map().name != 'Town04':  
        thread = threading.Thread(target=lambda: load_map(env, 'Town04'))  
        thread.start()  
  
    thread.join()
```



NO  
CODE  
TIME

En el archivo  
'helpers.py' en la  
función 'setup' se  
encuentra el  
cambio de mapa.

# El mapa

Son climas  
predefinidos.

También podemos cambiar el clima.

```
def change_weather(env, weather):  
    env.world.set_weather(getattr(carla.WeatherParameters, weather))
```

```
def setup(env):  
    if env.world.get_map().name != 'Town04':  
        thread = threading.Thread(target=lambda:load_map(env,'Town04'))  
        thread.start()  
  
        thread.join()  
  
    if env.world.get_weather() != 'ClearNoon':  
        thread = threading.Thread(target=lambda:change_weather(env,'ClearNoon'))  
        thread.start()  
  
        thread.join()
```

NO  
CODE  
TIME

# El mapa

En el archivo 'main.py' de la carpeta 'curso', vamos a agregar el cambio de mapa y clima.

```
from CarlaEnv import *
from Car import *

env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    input('Enter')

finally:
    env.destroy()
```

NO  
CODE  
TIME


Así debería estar  
el archivo  
'main.py'  
actualmente



# El mapa

Importamos el archivo 'helpers' al archivo 'main.py'.

Indentación



```
from CarlaEnv import *
from Car import *
from helpers import *

env = CarlaEnv()

try:
    model = env.blueprint_library.filter("mo

    spawn_points = env.map.get_spawn_points(
    spawn_point = spawn_points[0]
```

CODE  
TIME

Seguimos en el  
archivo 'main.py'.

# El mapa

Y llamamos a la función 'setup'.

▣▣ Indentación

```
try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    ▣▣ car = Car(env, model, spawn_point)
    ➡ setup(env)
    input('Enter')

finally:
    env.destroy()
```


Esta función  
necesita la  
conexión.

CODE  
TIME

# El mapa

Podemos generar todos los 'puntos del camino' en el mapa.

```
map_waypoints = env.map.generate_waypoints(gap)
```



Requiere la separación entre los puntos.



NO  
CODE  
TIME


# El mapa

Podemos obtener el 'punto del camino' más cercano al vehículo.

```
car_waypoint = env.map.get_waypoint(car_loc, project_to_road=True, lane_type=carla.LaneType.Driving)
```

También se pueden obtener los 'puntos del camino' cercanos a la posición actual.

```
car_waypoint.next(distance)
```



Debemos indicar  
la distancia a la  
que entre puntos.



NO  
CODE  
TIME

# El mapa

Con la función 'look' que se encuentra en el archivo 'helpers.py' se obtiene el 'punto del camino' más cercano a la posición del vehículo.

```
def look(env, car_loc, distance):  
  
    car_waypoint = env.map.get_waypoint(car_loc, project_to_road=True, lane_type=carla.LaneType.Driving)  
  
    nearest_waypoints = [w for w in car_waypoint.next(distance) if w.road_id not in city_roads]  
  
    if len(nearest_waypoints) == 0:  
        return car_waypoint  
  
    return nearest_waypoints[0]
```



NO  
CODE  
TIME

# El mapa

¿Cómo obtener un estado de referencia en CARLA? ✓

Listo.

Ahora falta responder esta pregunta:

¿Cómo modificar los actuadores del vehículo en CARLA?

# Actuadores del vehículo

# Actuadores del vehículo

En CARLA se tiene el siguiente comando.

```
# Controla el vehiculo con el acelerador, freno y volante  
self.vehicle.apply_control(carla.VehicleControl(throttle=throttle, brake=brake, steer=steer))
```

A red, stylized cloud graphic with a black outline, containing the text "NO CODE TIME" in white capital letters.

NO  
CODE  
TIME



# Actuadores del vehículo

Se necesita el actor de vehículo.

```
# Controla el vehiculo con el acelerador, freno y volante  
self.vehicle.apply_control(carla.VehicleControl(throttle=throttle, brake=brake, steer=steer))
```



NO  
CODE  
TIME

# Actuadores del vehículo

Los valores de aceleración se encuentran entre 0 y 1.

```
# Controla el vehiculo con el acelerador, freno y volante  
self.vehicle.apply_control(carla.VehicleControl(throttle=throttle, brake=brake, steer=steer))
```



Donde el 1 es la  
máxima  
aceleración.

NO  
CODE  
TIME

# Actuadores del vehículo

Los valores del freno también se encuentran entre 0 y 1.

```
# Controla el vehiculo con el acelerador, freno y volante  
self.vehicle.apply_control(carla.VehicleControl(throttle=throttle, brake=brake, steer=steer))
```



Donde el 1 es la máxima fuerza de frenado.

NO  
CODE  
TIME

# Actuadores del vehículo

Los valores del volante se encuentran entre -1 y 1.

```
# Controla el vehiculo con el acelerador, freno y volante  
self.vehicle.apply_control(carla.VehicleControl(throttle=throttle, brake=brake, steer=steer))
```



-1 es la izquierda  
y 1 es a la  
derecha.

NO  
CODE  
TIME

# Actuadores del vehículo

La clase 'Car' contiene el actor del vehículo.

```
from Camera import *  
  
class Car:  
    def __init__(self, env, model, spawn_point, camera_config={}):  
        # Invoca el vehiculo en el mundo  
        self.vehicle = env.spawn_actor(model, spawn_point)  
        self.camera = Camera(env, self.vehicle, camera_config)
```



NO  
CODE  
TIME

# Actuadores del vehículo

Vamos a implementar la función de control en la clase 'Car' que se encuentra en el archivo 'Car.py'.

```
from Camera import *

class Car:

    def __init__(self, env, model, spawn_point, camera_config={}):
        # Invoca el vehiculo en el mundo
        self.vehicle = env.spawn_actor(model, spawn_point)

        self.camera = Camera(env, self.vehicle, camera_config)

    def control(self, throttle, brake, steer):
        # Controla el vehiculo con el acelerador, freno y volante
        self.vehicle.apply_control(carla.VehicleControl(throttle=throttle, brake=brake, steer=steer))
```



CODE  
TIME

# Actuadores del vehículo

¿Cómo obtener un estado de referencia en CARLA? ✓

Listo.

¿Cómo modificar los actuadores del vehículo en CARLA? ✓

Listo.

Ya sabemos modificar los actuadores, pero:

¿Cómo elegir los valores adecuados?

# Control

Pure Pursuit o Algoritmo de persecución



# Algoritmo de persecución

Este algoritmo es uno de los más sencillos de implementar.

Considera que el vehículo mantiene una velocidad constante.

Solamente se encarga de modificar el volante.



# Algoritmo de persecución

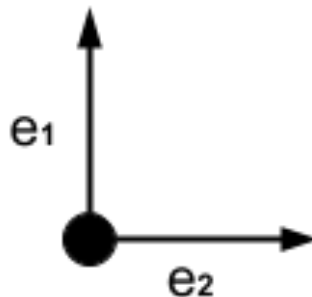
Además, este algoritmo necesita modelar el vehículo en un plano de 2 dimensiones.

Para esto, se necesita conocer el modelo cinemático de una sola tracción.

También llamado  
el modelo de la  
bicicleta.

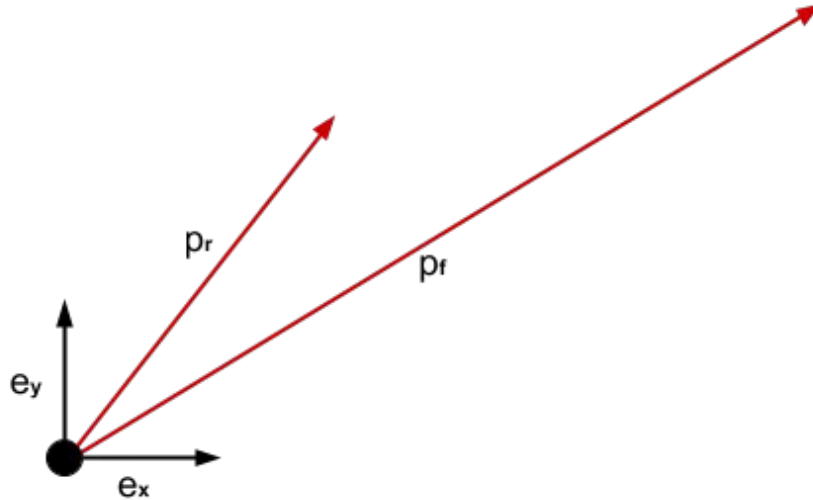
# Algoritmo de persecución: Modelo

El modelo de la bicicleta necesita una base vectorial.



# Algoritmo de persecución: Modelo

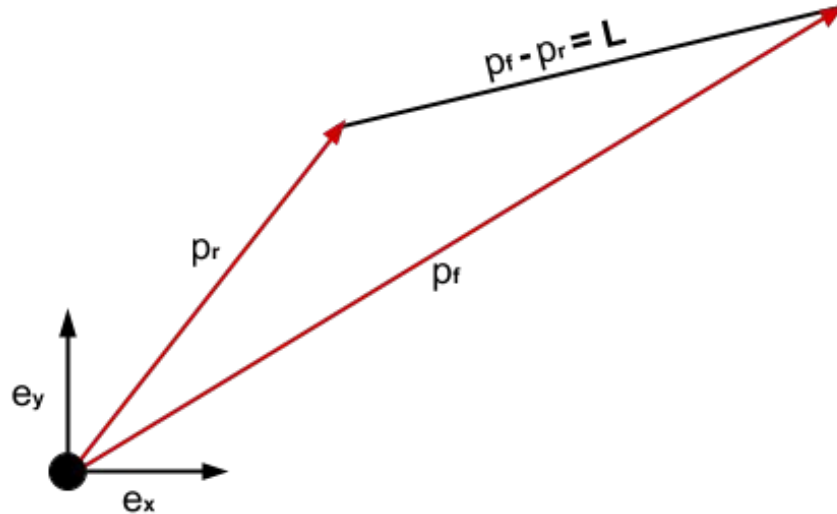
En este plano, debemos ubicar la posición de las llantas.



Delantera y  
trasera.

# Algoritmo de persecución: Modelo

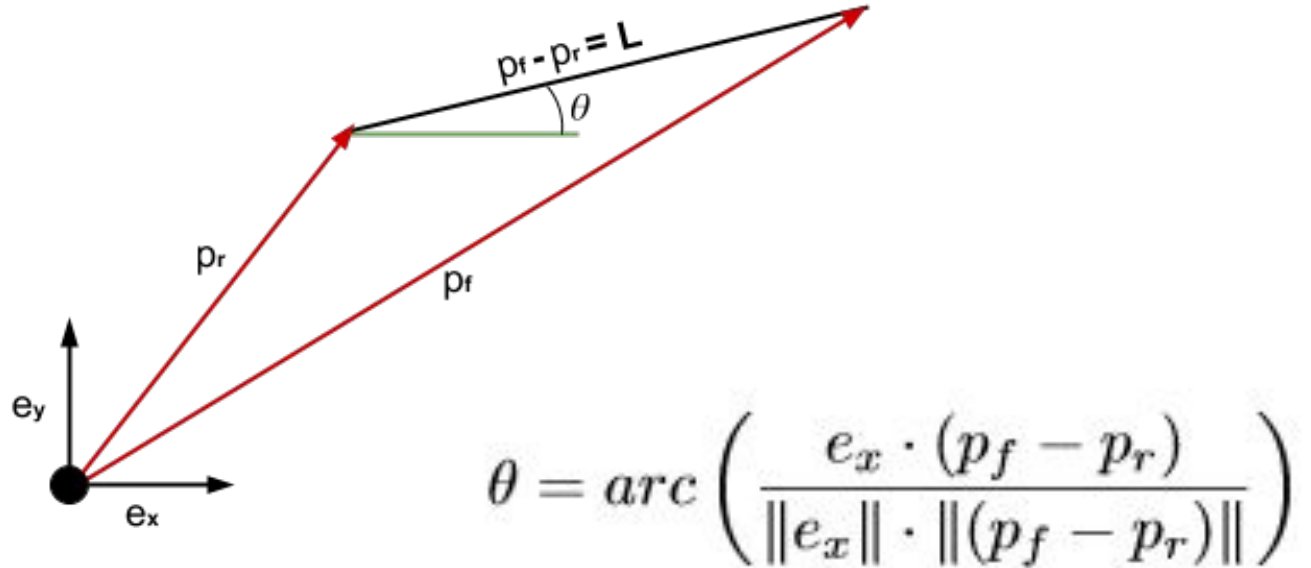
El modelo asume que las llantas se encuentran unidas por un eje rígido.



Longitud del  
vehículo.

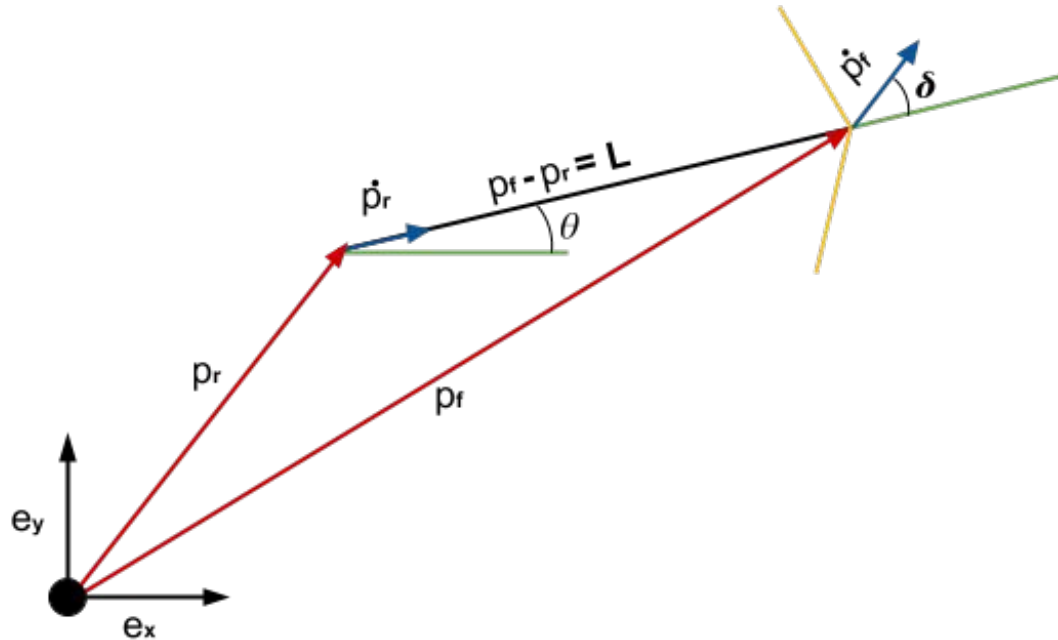
# Algoritmo de persecución: Modelo

La dirección del vehículo está representada por  $\theta$ .



# Algoritmo de persecución: Modelo

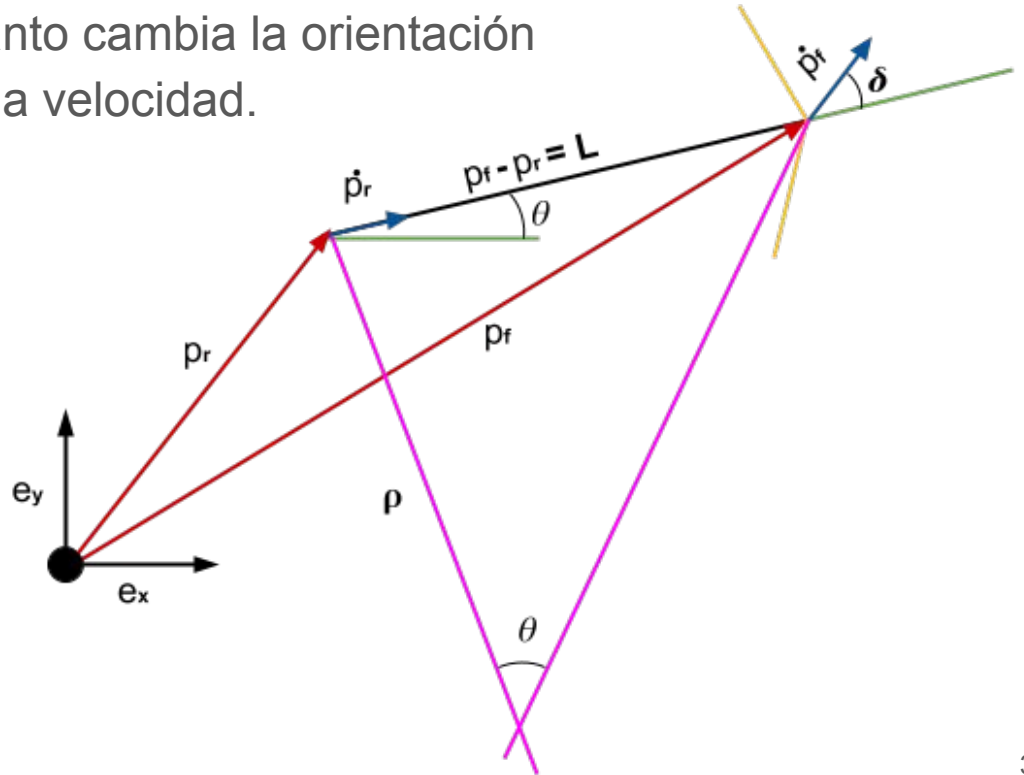
La dirección de la velocidad angular está definida por  $\delta$



# Algoritmo de persecución: Modelo

Con esta ecuación sabemos cuánto cambia la orientación del vehículo después de aplicar la velocidad.

$$\dot{\theta} = \frac{v_r}{L} \tan \theta$$





# ahora sí, el algoritmo

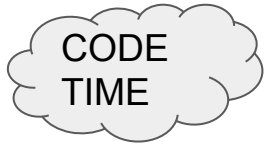
Pure Pursuit

# Algoritmo de persecución

En la carpeta 'curso' hay que crear el archivo 'PurePursuit.py'.

Primero vamos a importar la librería 'math' y las funciones de ayuda del archivo 'helpers.py'.

```
import math  
from helpers import *
```



# Algoritmo de persecución

Y continuando con el paradigma de programación orientado a objetos. Creamos la clase 'PurePursuit'.

```
import math
from helpers import *

class PurePursuit:

    def __init__(self, vehicle, target_speed, lookahead):

        self.target_speed = target_speed
        self.lookahead = lookahead

        self.L = vehicle.bounding_box.extent.x/2
        self.vehicle = vehicle
```

NO  
CODE  
TIME

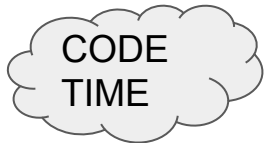
# Algoritmo de persecución

La clase necesita el actor del vehículo.

Porque el algoritmo requiere de la posición del vehículo en el mundo.

```
import math
from helpers import *

class PurePursuit:
    def __init__(self, vehicle, target_speed, lookahead):
```



Indentación

# Algoritmo de persecución

La clase necesita la aceleración deseada.

El algoritmo requiere de la velocidad deseada para hacer la actualización de posiciones, pero tenemos CARLA.

```
import math
from helpers import *

class PurePursuit:
    def __init__(self, vehicle, target_speed, lookahead):
```



CODE  
TIME

■ Indentación

# Algoritmo de persecución

Este algoritmo no calcula la entrada de la aceleración, por eso se necesita la aceleración deseada que se encuentra entre 0 y 1.

```
import math
from helpers import *

class PurePursuit:
    def __init__(self, vehicle, target_speed, lookahead):
```



CODE  
TIME

■ ■ Indentación

# Algoritmo de persecución

La clase también necesita la distancia a la que va a buscar el siguiente punto de referencia.

```
import math
from helpers import *

class PurePursuit:
    def __init__(self, vehicle, target_speed, lookahead):
```

CODE  
TIME

Indentación

# Algoritmo de persecución

Ahora guardamos las variables en la clase.

```
class PurePursuit:  
    ■■ def __init__(self, vehicle, target_speed, lookahead):  
        ■■ self.target_speed = target_speed  
        ■■ self.lookahead = lookahead  
        ➡ self.L = vehicle.bounding_box.extent.x/2  
        ■■ self.vehicle = vehicle
```

CODE  
TIME

La variable 'L'  
contiene la  
longitud del  
vehículo.

■■ Indentación



# Algoritmo de persecución

Vamos a crear el método 'step' en la clase 'PurePursuit'.

```
import math
from helpers import *

class PurePursuit:

    def __init__(self, vehicle, target_speed, lookahead):

        self.target_speed = target_speed
        self.lookahead = lookahead

        self.L = vehicle.bounding_box.extent.x/2
        self.vehicle = vehicle

    def step(self, env):
        # Posición del vehículo en el mundo
        loc_car = self.vehicle.get_location()

        # Obtenemos el punto a distancia 'lookahead' del vehículo.
        waypoint = look(env, self.vehicle, self.lookahead)
        # Y la posición del punto
        self.loc_waypoint = waypoint.transform.location
        # Distancia entre la posición del vehículo y el punto del camino
        H = distancia(loc_car, self.loc_waypoint)
        # Orientación del vehículo
        theta = math.radians(self.vehicle.get_transform().rotation.yaw)
        # ángulo necesario para alinearse con el punto.
        alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta

        # Cuánto hay que girar las llantas
        delta = math.atan((2*self.L*math.sin(alpha))/H)

        return (self.target_speed, 0.0, delta)
```

NO  
CODE  
TIME

NO  
CODE  
TIME

Aquí es donde se  
calcula el control.

NO  
CODE  
TIME

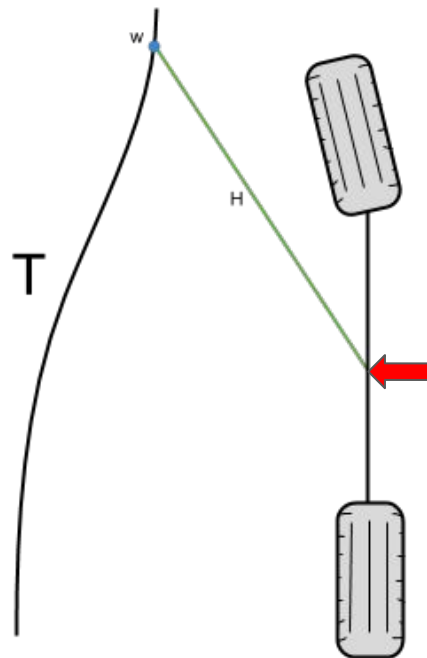
NO  
CODE  
TIME

# Algoritmo de persecución

Primero necesitamos la posición del actor del vehículo en el mundo.

```
def step(self, env):  
    # Posición del vehículo en el mundo  
    loc_car = self.vehicle.get_location()  
  
    # Obtenemos el punto a distancia 'lookahead' del vehículo.  
    waypoint = look(env, self.vehicle, self.lookahead)  
    # Y la posición del punto  
    self.loc_waypoint = waypoint.transform.location  
    # Distancia entre la posición del vehículo y el punto del camino  
    H = distancia(loc_car, self.loc_waypoint)  
    # Orientación del vehículo  
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)  
    # ángulo necesario para alinearse con el punto.  
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta  
  
    # Cuánto hay que girar las llantas  
    delta = math.atan((2*self.L*math.sin(alpha))/H)  
  
    return (self.target_speed, 0.0, delta)
```

NO  
CODE  
TIME



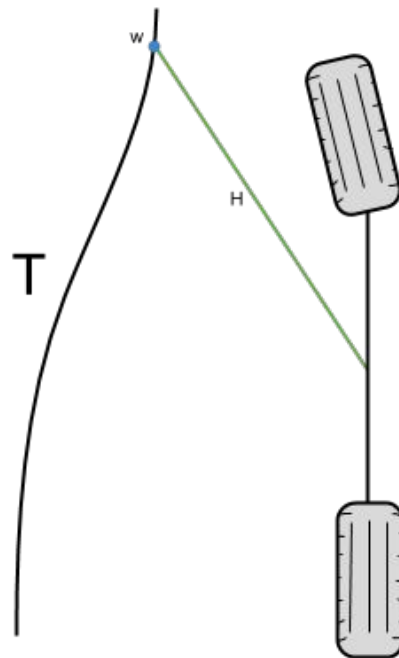
# Algoritmo de persecución

También necesitamos la posición de un punto de referencia.

```
def step(self, env):  
    # Posición del vehículo en el mundo  
    loc_car = self.vehicle.get_location()  
  
    # Obtenemos el punto a distancia 'lookahead' del vehículo.  
    waypoint = look(env, self.vehicle, self.lookahead)  
    # Y la posición del punto  
    self.loc_waypoint = waypoint.transform.location  
    # Distancia entre la posición del vehículo y el punto del camino  
    H = distancia(loc_car, self.loc_waypoint)  
    # Orientación del vehículo  
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)  
    # ángulo necesario para alinearse con el punto.  
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta  
  
    # Cuánto hay que girar las llantas  
    delta = math.atan((2*self.L*math.sin(alpha))/H)  
  
    return (self.target_speed, 0.0, delta)
```

NO  
CODE  
TIME

En CARLA lo  
tenemos en el  
archivo 'helpers.py'



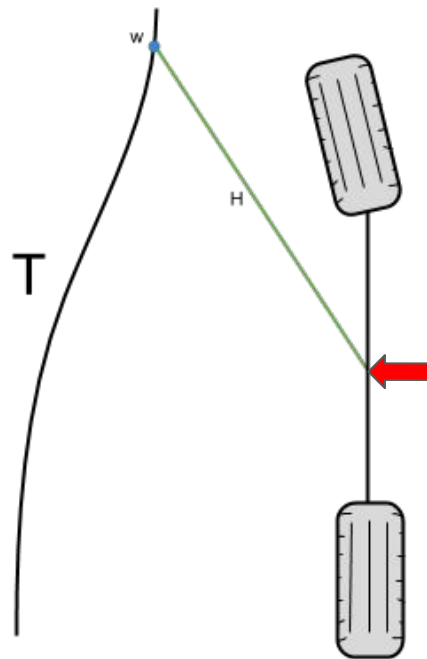
# Algoritmo de persecución

La H es la distancia entre el punto de referencia y la posición del vehículo.

```
def step(self, env):  
    # Posición del vehículo en el mundo  
    loc_car = self.vehicle.get_location()  
  
    # Obtenemos el punto a distancia 'lookahead' del vehículo.  
    waypoint = look(env, self.vehicle, self.lookahead)  
    # Y la posición del punto  
    self.loc_waypoint = waypoint.transform.location  
    # Distancia entre la posición del vehículo y el punto del camino  
    H = distancia(loc_car, self.loc_waypoint)  
    # Orientación del vehículo  
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)  
    # ángulo necesario para alinearse con el punto.  
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta  
  
    # Cuánto hay que girar las llantas  
    delta = math.atan((2*self.L*math.sin(alpha))/H)  
  
    return [self.target_speed, 0.0, delta]
```

NO  
CODE  
TIME

En el archivo 'helpers.py'  
se encuentra la función de  
distancia entre puntos

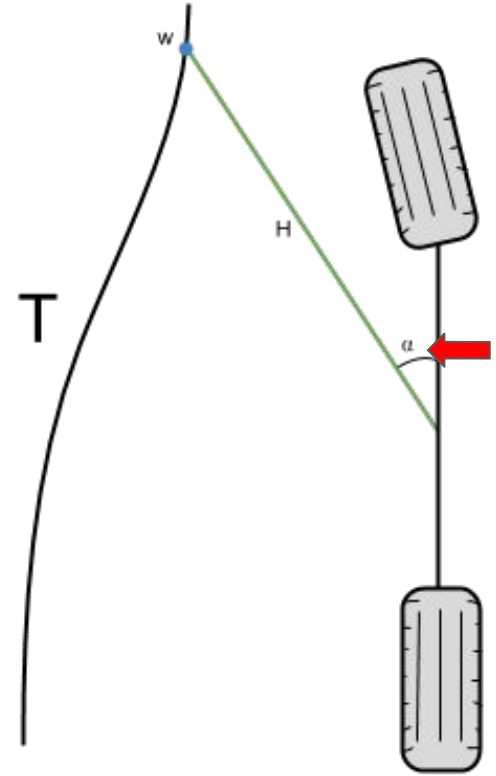


# Algoritmo de persecución

$\alpha$  es el ángulo que determina cuánto debe girar el vehículo para alinearse con el punto de referencia.

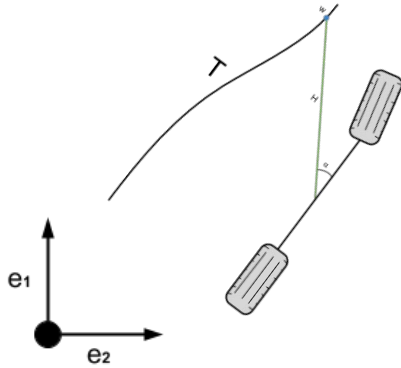
El problema, es que el verdadero ángulo está influenciado por la orientación del vehículo con respecto al mundo.

NO  
CODE  
TIME

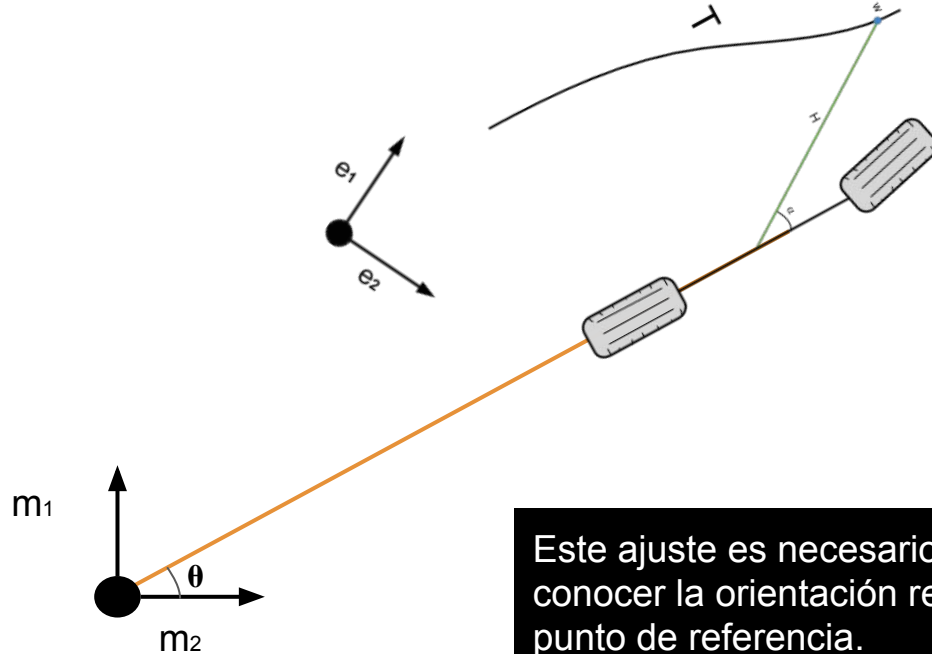


# Algoritmo de persecución

## Marcos de referencia



NO  
CODE  
TIME



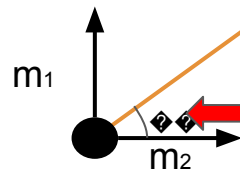
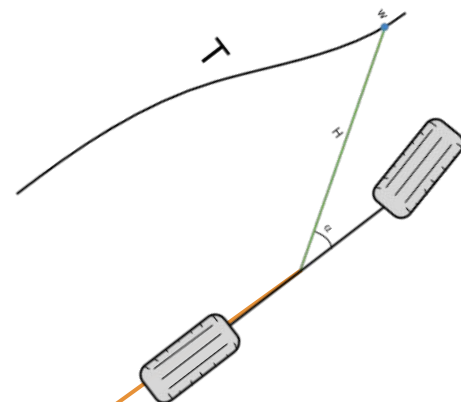
Este ajuste es necesario para conocer la orientación real del punto de referencia.

# Algoritmo de persecución

$\theta$  es la orientación del vehículo en el mundo.

```
def step(self, env):  
    # Posición del vehículo en el mundo  
    loc_car = self.vehicle.get_location()  
  
    # Obtenemos el punto a distancia 'lookahead' del vehículo.  
    waypoint = look(env, self.vehicle, self.lookahead)  
    # Y la posición del punto  
    self.loc_waypoint = waypoint.transform.location  
    # Distancia entre la posición del vehículo y el punto del camino  
    H = distancia(loc_car, self.loc_waypoint)  
    # Orientación del vehículo  
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)  
    # ángulo necesario para alinearse con el punto.  
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta  
  
    # Cuánto hay que girar las llantas  
    delta = math.atan((2*self.L*math.sin(alpha))/H)  
  
    return [self.target_speed, 0.0, delta]
```

NO  
CODE  
TIME

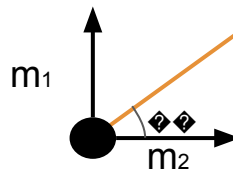
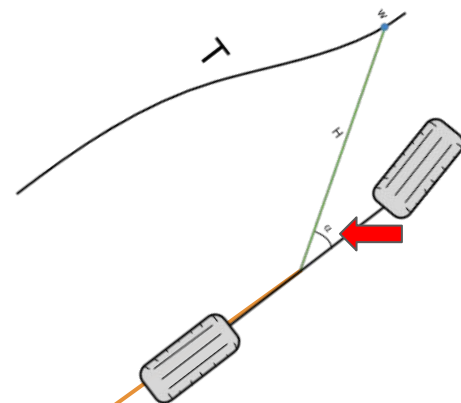


# Algoritmo de persecución

Ahora calculamos el ángulo  $\alpha$

```
def step(self, env):  
    # Posición del vehículo en el mundo  
    loc_car = self.vehicle.get_location()  
  
    # Obtenemos el punto a distancia 'lookahead' del vehículo.  
    waypoint = look(env, self.vehicle, self.lookahead)  
    # Y la posición del punto  
    self.loc_waypoint = waypoint.transform.location  
    # Distancia entre la posición del vehículo y el punto del camino  
    H = distancia(loc_car, self.loc_waypoint)  
    # Orientación del vehículo  
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)  
    # ángulo necesario para alinearse con el punto.  
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta  
    # Cuánto hay que girar las llantas  
    delta = math.atan((2*self.L*math.sin(alpha))/H)  
    return [self.target_speed, 0.0, delta]
```

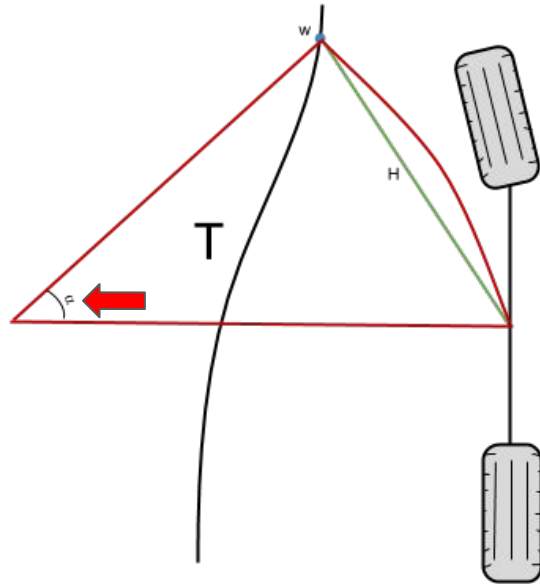
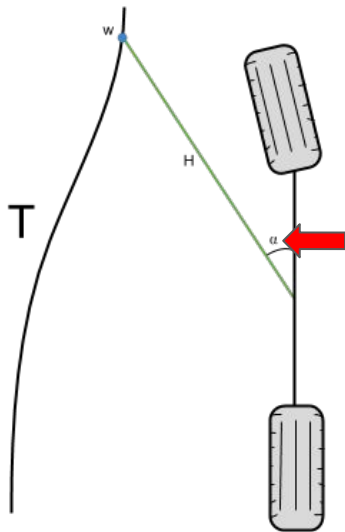
NO  
CODE  
TIME





# Algoritmo de persecución

$\alpha$  es el ángulo del semicírculo que pasa por la posición del vehículo y la posición del punto de referencia.



# Algoritmo de persecución

Por eso se necesita la ecuación del semicírculo para determinar el ángulo de giro del volante.

$$k = \frac{2\sin(\alpha)}{H} \quad w = \frac{2v_r \sin(\alpha)}{H}$$

Esta es la ecuación del semicírculo influenciada por la velocidad.

# Algoritmo de persecución

Por otra parte, en la ecuación del cambio de orientación del modelo de la bicicleta.

$$\dot{\theta} = \frac{v_r}{L} \tan \theta$$

Luego despejamos el ángulo de giro.

$$\theta = \operatorname{atan} \left( \frac{L \cdot \dot{\theta}}{v_r} \right)$$

# Algoritmo de persecución

Queremos unir las dos cosas. La ecuación del semicírculo y la ecuación del cambio de ángulo del modelo de la bicicleta influenciado por la velocidad.

$$\theta = \operatorname{atan}\left(\frac{L \cdot \dot{\theta}}{v_r}\right) \quad w = \frac{2v_r \sin(\alpha)}{H}$$

# Algoritmo de persecución

Resulta que el cambio del ángulo con respecto a la velocidad  $\dot{\theta}$  es la  $w$

$$\theta = \text{atan} \left( \frac{L \cdot \dot{\theta}}{v_r} \right) \quad w = \frac{2v_r \sin(\alpha)}{H}$$

# Algoritmo de persecución

$$w = \frac{2v_r \sin(\alpha)}{H}$$

Entonces sustituimos la  $w$  por la  $\dot{\theta}$  en la primer ecuación.

$$\theta = \operatorname{atan} \left( \frac{L \cdot \dot{\theta}}{v_r} \right) \quad \theta = \operatorname{atan} \left( \frac{L \cdot w}{v_r} \right)$$

# Algoritmo de persecución

El ángulo que debemos girar las llantas está dado por la siguiente ecuación:

$$\theta = \operatorname{atan}\left(\frac{2 \cdot L \cdot \sin(\alpha)}{H}\right)$$

# Algoritmo de persecución

Listo, ya calculamos el control del volante.

```
def step(self, env):
    # Posición del vehículo en el mundo
    loc_car = self.vehicle.get_location()

    # Obtenemos el punto a distancia 'lookahead' del vehículo.
    waypoint = look(env, self.vehicle, self.lookahead)
    # Y la posición del punto
    self.loc_waypoint = waypoint.transform.location
    # Distancia entre la posición del vehículo y el punto del camino
    H = distancia(loc_car, self.loc_waypoint)
    # Orientación del vehículo
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)
    # ángulo necesario para alinearse con el punto.
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta

    # Cuánto hay que girar las llantas
    delta = math.atan((2*self.L*math.sin(alpha))/H)

    return (self.target_speed, 0.0, delta)
```

$$\theta = \text{atan} \left( \frac{2 \cdot L \cdot \sin(\alpha)}{H} \right)$$

NO  
CODE  
TIME



# Algoritmo de persecución

Por último regresamos el control calculado por el algoritmo de persecución.

```
def step(self, env):  
    # Posición del vehículo en el mundo  
    loc_car = self.vehicle.get_location()  
  
    # Obtenemos el punto a distancia 'lookahead' del vehículo.  
    waypoint = look(env, self.vehicle, self.lookahead)  
    # Y la posición del punto  
    self.loc_waypoint = waypoint.transform.location  
    # Distancia entre la posición del vehículo y el punto del camino  
    H = distancia(loc_car, self.loc_waypoint)  
    # Orientación del vehículo  
    theta = math.radians(self.vehicle.get_transform().rotation.yaw)  
    # ángulo necesario para alinearse con el punto.  
    alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta  
  
    # Cuánto hay que girar las llantas  
    delta = math.atan((2*self.L*math.sin(alpha))/H)  
  
    return (self.target_speed, 0.0, delta]
```

CODE  
TIME

El primer valor es el  
acelerador, el segundo el freno  
y el tercero el ángulo del  
volante.

Indentación

# Algoritmo de persecución

```
import math
from helpers import *

class PurePursuit:
    def __init__(self, vehicle, target_speed, lookahead):
        self.target_speed = target_speed
        self.lookahead = lookahead

        self.L = vehicle.bounding_box.extent.x/2
        self.vehicle = vehicle

    def step(self, env):
        # Posición del vehículo en el mundo
        loc_car = self.vehicle.get_location()

        # Obtenemos el punto a distancia 'lookahead' del vehículo.
        waypoint = look(env, self.vehicle, self.lookahead)
        # Y la posición del punto
        self.loc_waypoint = waypoint.transform.location
        # Distancia entre la posición del vehículo y el punto del camino
        H = distancia(loc_car, self.loc_waypoint)
        # Orientación del vehículo
        theta = math.radians(self.vehicle.get_transform().rotation.yaw)
        # ángulo necesario para alinearse con el punto.
        alpha = math.atan2((self.loc_waypoint.y - loc_car.y), (self.loc_waypoint.x - loc_car.x)) - theta

        # Cuánto hay que girar las llantas
        delta = math.atan((2*self.L*math.sin(alpha))/H)

        return (self.target_speed, 0.0, delta)]
```

Indentación


CODE  
TIME

Hasta este  
momento deben  
tener algo así.

Hay que aplicar el control

# Aplicar el control

En el archivo 'main.py' de la carpeta 'curso'. Vamos a importar la clase 'PurePursuit'.



```
from CarlaEnv import *
from Car import *
from PurePursuit import *
from helpers import *

env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

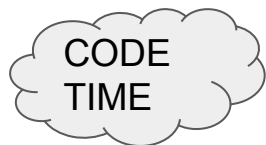
    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    input('Enter')

finally:
    env.destroy()
```



Indentación

# Aplicar el control

Vamos a crear un objeto que se encargue del control.

```
env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    ➔ controller = PurePursuit(car.vehicle, 0.5, 15.0)

    input('Enter')

finally:
    env.destroy()
```

NO  
CODE  
TIME

# Aplicar el control

El actor del vehículo para obtener su posición.

```
env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)

    input('Enter')

finally:
    env.destroy()
```



NO  
CODE  
TIME

# Aplicar el control

Aceleración constante que se va a aplicar al control.

```
env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)
    input('Enter')

finally:
    env.destroy()
```

NO  
CODE  
TIME

# Aplicar el control

Distancia del punto de referencia con respecto a la posición del vehículo.

```
env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)
    input('Enter')

finally:
    env.destroy()
```

NO  
CODE  
TIME



# Aplicar el control

Pueden jugar con estos dos valores.

```
env = CarlaEnv()

try:
    ■■ model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)

    input('Enter')


finally:
    env.destroy()
```



■■ Indentación

# Aplicar el control

En el archivo 'main.py' vamos a definir la función 'control' para aplicar el control.



```
from PurePursuit import *  
from helpers import *  
def control(env, car, controller):  
  
    env = CarlaEnv()  
  
    try:  
        model = env.blueprint_library.filter("n
```

NO  
CODE  
TIME

# Aplicar el control

Necesitamos la conexión al servidor.

```
from PurePursuit import *  
from helpers import *  
def control(env, car, controller):  
  
    env = CarlaEnv()  
  
    try:  
        model = env.blueprint_library.filter("n
```

NO  
CODE  
TIME

# Aplicar el control

Necesitamos el automóvil en el que se va a aplicar el control.

```
from PurePursuit import *  
from helpers import *  
def control(env, car, controller):  
  
    env = CarlaEnv()  
  
    try:  
        model = env.blueprint_library.filter("n
```



NO  
CODE  
TIME

# Aplicar el control

Y también el controlador que va a calcular el control.

```
from PurePursuit import *  
from helpers import *  
→ def control(env, car, controller):  
  
    env = CarlaEnv()  
  
    try:  
        ■■ model = env.blueprint_library.filter("n
```



■■ Indentación

# Aplicar el control

Primero hay que calcular el control.

```
def control(env, car, controller):  
    → throttle, brake, steer = controller.step(env)
```

En la función 'step' de la clase 'PurePursuit' se hace el cálculo del control.

NO  
CODE  
TIME

# Aplicar el control

Después hay que aplicar el control

```
def control(env, car, controller):  
    ── throttle, brake, steer = controller.step(env)  
    ─→ car.control(throttle, brake, steer)
```

En la función 'control' de la clase 'Car' se aplica el control.



── Indentación

# Aplicar el control

Esto es para poder ver el punto de referencia que está viendo el vehículo en el servidor.

```
def control(env, car, controller):  
    ── throttle, brake, steer = controller.step(env)  
       car.control(throttle, brake, steer)  
    → show(env, controller.loc_waypoint)
```

Se marcará como un círculo rojo.

La función 'show' se encuentra definida en el archivo 'helpers'.



─ Indentación



# Aplicar el control

Debemos aplicar el control cada vez que el simulador se actualice, para asegurarnos de que el vehículo va a orientarse con mayor precisión al punto de referencia.

```
try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)
    → env.world.on_tick(lambda ws: control(env, car, controller))

    input('Enter')

finally:
    env.destroy()
```

NO  
CODE  
TIME

# Aplicar el control

Esta función se ejecuta cada vez que el simulador se actualiza. Y aquí es donde llama la función de control para aplicarlo sobre el vehículo.

```
try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[0]

    car = Car(env, model, spawn_point)

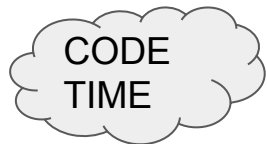
    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)

    env.world.on_tick(lambda ws: control(env, car, controller))

    input('Enter')

finally:
    env.destroy()
```



Indentación

# Aplicar el control

Es momento de ver el automóvil navegar sobre el simulador CARLA.

El archivo  
'main.py' debe  
verse así.

```
from CarlaEnv import *
from Car import *
from PurePursuit import *

from helpers import *

def control(env, car, controller):

    throttle, brake, steer = controller.step(env)
    car.control(throttle, brake, steer)

    show(env, controller.loc_waypoint)

env = CarlaEnv()

try:
    model = env.blueprint_library.filter("model3")[0]

    spawn_points = env.map.get_spawn_points()
    spawn_point = spawn_points[2]

    car = Car(env, model, spawn_point)

    setup(env)

    controller = PurePursuit(car.vehicle, 0.5, 15.0)

    env.world.on_tick(lambda ws: control(env, car, controller))

    input('Enter')

finally:
    env.destroy()
```



# Fin de la segunda parte

¿Preguntas?

Dudas o aclaraciones:  
[rperalta@cicese.edu.mx](mailto:rperalta@cicese.edu.mx)