# Autonomous Applications

~ by Yipeng Zhou

## 1. Architecture

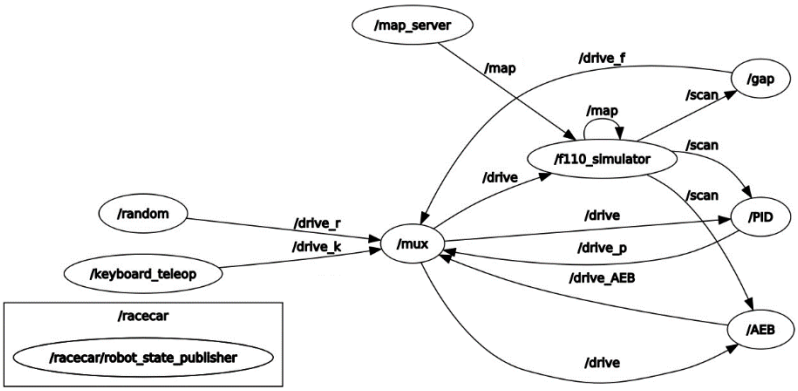


Figure 1.1: Architecture

Figure 1.1 shows the relations between different nodes. In this project I create six nodes, which are listed in Table 1.1 below.

| node | subscribed topic | published topic |
|---|---|---|
| keyboard_teleop | | /mode<br>/drive_k |
| random | | /drive_r |
| gap | /scan | /drive_f |
| mux | /drive_AEB<br>/mode<br>/drive_r<br>/drive_f<br>/drive_p<br>/drive_k | /drive |
| AEB | /drive<br>/scan | /drive_AEB |
| PID | /drive<br>/scan | /drive_p |

Table 1.1: Nodes

Among them *keyboard_teleop*, *random*, *gap* and *PID* adopt different control strategies; *keyboard_teleop* is also used to select different control modes/strategies; *mux* according to selection implements one of these modes to control the car; *AEB* is used to brake the car when a collision is about to occur.

## 2. Nodes

### 2.1 keyboard_teleop

In this lab I create a node called "*keyboard_teleop*". This node publishes two topics: "*/mode*" and "*/drive_k*". This node can determine the selection of mode (*random*, *gap*, *PID* or *keyboard*) and the action of the car (go forward, go backward, steer left, steer right and stop) through keyboard input. The topic */mode* and */drive_k* published by this node will be subscribed by the node *mux*, and the corresponding mode will be implemented by *mux*.

This node will first through the function *getch()* get input from the keyboard. Then it calls the function *keyToMode()* to obtain the corresponding mode according to this input and publishes

this mode message through topic */mode*. If the obtained mode is *keyboard*, this node will then call the function *keyToIndex()* to obtain the corresponding index. This index will be used to extract the corresponding speed and steering angle from the two-dimensional array "*mapping*". Then the node will multiply the obtained speed and angle by the corresponding limit coefficient to make and publish the message through topic */drive_k*.

We must consider a special case: the node *keyboard_teleop* publishes two topics (*/mode* and */drive_k*) in the same cycle. For example, we want to change the mode to *keyboard* and let the car go backward. According to the function *keyToMode()* and *keyToIndex()*, we just need to press key 's' once, the node will publish the mode 'k' through */mode* and the backward action through */drive_k* in a cycle. But we are not sure which messages the node *mux* will receive first. If *mux* receives the mode message first, it will switch the current mode to *keyboard*, and then let the car go backward. But if *mux* receives the action message from /*drive_k* first, since the current mode is not *keyboard*, *mux* will not let the car go backward. Then *mux* receives the mode message and switches the mode to *keyboard*. In the latter case, in order to complete the mode switch to *keyboard* and make the car go backward, we need to press key 's' twice. In order to eliminate this uncertainty, I add a loop before the node *keyboard_teleop* publishes the topic /*drive_k* (line 138 ~ 144 in *keyboard_teleop.cpp*). Its purpose is to wait for a period of time so that after *mux* receives the mode message, then the node *keyboard_teleop* publishes the topic */drive_k*. But this attempt does not achieve the expected result: *mux* will first obtain the message from topic /*drive_k*, and then obtain the message from */mode*. This may be due to compiler optimization or pre-execution by the processor. So please keep in mind that sometimes you need to press the key twice to switch the mode to *keyboard* and manipulate the car.

## 2.2 random

In this lab I create a node called "*random*". This node publishes one topic "*/drive_r*". This node can take actions randomly (go forward, go backward, steer left and steer right). Every action has the same probability (25%).

This node will first generate a pseudo-random number (0, 1, 2 or 3). The generated random number will be used as the index to extract the corresponding speed and steering angle from the two-dimensional array "*mapping*". Then the node will multiply the obtained speed and angle by the corresponding limit coefficient to make and publish the message through topic */drive_r*. The frequency of all the above operations is 2HZ.

## 2.3 gap

In this lab I create a node called "*gap*". This node subscribes one topic: "*/scan*"; and publishes one topic "*/drive_f*". This node adopts the strategy "follow the gap" and tries different methods in order to implement this strategy successfully, that is, the car goes where the distance between it and its environment is the biggest and performs complete laps without crashing.

The node continuously receives the radar data from the topic */scan* and calls the function *callback()*. In this function, the data from the topic */scan* are first copied into the vector "*temp*". Then we will look for the maximum and minimum distance between the car and its environment and their corresponding directions (*index_max_distance* and *index_min_distance*). Since we only care about the data in the front half of the car, we only need to compare the date from $270^{th}$ to $810^{th}$ of *temp*. Next, I set the speed to 2.0m/s, use the default initial position of the car and try different methods to calculate the *steering_angle*. At last, the node makes the message and publishes it through topic */drive_f*.

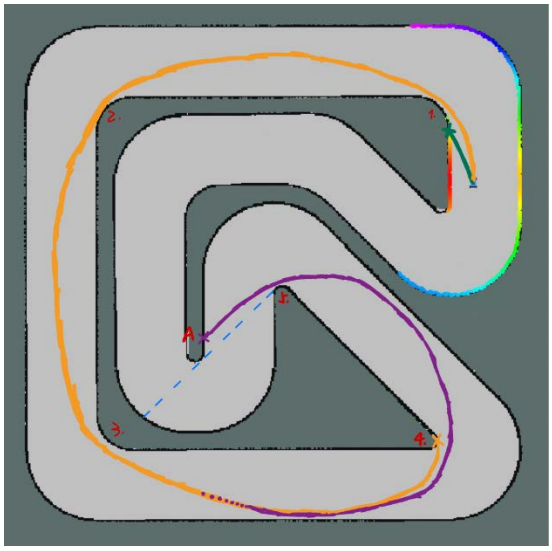

Figure 2.3.1: Collision

### 2.3.1 Method 1

Firstly, we directly use the angle between the direction of the maximum distance (*index_max_distance*) and the direction of the car (540$^{th}$) as the *steering_angle*. For convenience, we will call this angle *delta_angle* from now on.

$$delta\_angle = (index\_max\_distance - 540) * scan \rightarrow angle\_increment \quad (2.3.1)$$

But with this method the car will collide with the wall at the corner 1 (green trajectory in *Figure 2.3.1*). This is because the *steering_angle* is a little large when the *delta_angle* is small. To solve this problem, we take a different strategy to compute *steering_angle* according to *delta_angle* in method 2.

### 2.3.2 Method 2

We want to let *steering_angle* be very small when *delta_angle* is small; and let *steering_angle* increase rapidly when *delta_angle* becomes large. Therefore, I use power function to transform *delta_angle* into *steering_angle*.

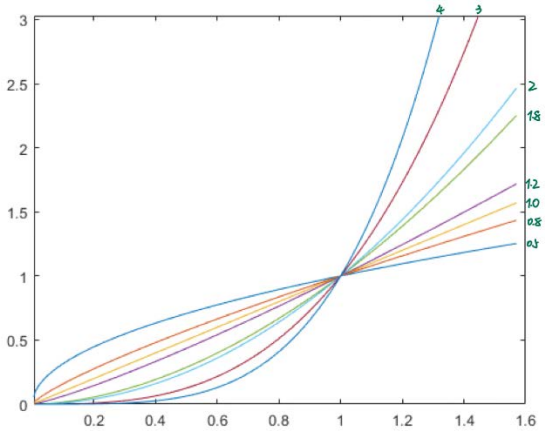


Figure 2.3.2: Power function

Figure 2.3.2 shows the changing trend of the power function with different exponents. The power functions with exponent greater than 1 can satisfy our needs. I use the *delta_angle* as base and set exponent to 5 (if the exponent is too large, the car will understeer at corner 1; if the exponent is too small, oversteer at corner 1). Here we need to consider two different situations where *delta_angle* is positive or negative. With a negative *delta_angle*, we should let the car turn right, so *steering_angle* should be negative and vice versa. With this method, the

car can successfully pass the corner 1, corner 2 and corner 3. But this method also has problems. It will make the car start to turn too early on the long straight, which makes the trajectory of the car always close to the inner side of the racetrack. This is not conducive to passing corners. Besides, at the sharp corners, like corner 4 and corner 5, *delta_angle* is large. Therefore, *steering_angle* will become very large at this sharp corners. These two problems will cause the car to collide with the wall at corner 4 (yellow trajectory in *Figure 2.3.1*). To solve these problems, I improve the strategies and get method 3.

### 2.3.3 Method 3

Based on method 2, I add a restriction on the maximum *steering_angle*, thus avoiding steering angle of the car too large at the sharp corners. Here we set the range of *steering_angle* is [-0.145, 0.145] and in this way the car can successfully pass corner 4. But after the car passes corner 5, it will collide with the wall in area A (purple trajectory in *Figure 2.3.1*). The reason is still that the car turns too early in the straight, so that it is too close to the inner side of the racetrack when it passes corner 5. Although the car can successfully pass corner 5, it has not enough distance to adjust its direction and eventually collide in area A. I have considered adopting a narrower comparison range than [270$^{th}$, 810$^{th}$], but this strategy cannot solve this problem, that is, the car turns too early on the straight. In method 4 I propose a strategy that can really solve this problem.

### 2.3.4 Method 4

I do not want the car to start turning when the *delta_angle* is small, so that the car will not turn too early on the straight. For this reason, I add a restriction on the basis of method 3: only when *abs(index_max_distance – 540)* larger than 80, the car starts to turn. But this method has a fatal flaw: after the car passes corner 5, the *delta_angle* here is very small (blue dotted line in *Figure 2.3.1*); therefore, the car will not adjust its direction here and eventually collide in area A. I need to try other methods.

### 2.3.5 Method 5

I give up the strategy adopted in method 4 and on the basis of method 3 I adjust the speed of the car dynamically. I want the car run fast in the straights and run slowly at the corners, so that the car has more time to adjust its direction at the corners. I use the maximum distance between the front half of the car and the environment to adjust the speed, and set the upper limit of the speed to 4m/s. But this method cannot solve the contradiction between corner 1 and the racetrack after corner 5: In order to pass corner 1, we want that when *delta_angle* is small, *steering_angle* is very small; But after the car passes corner 5, due to the small *delta_angle* here, *steering_angle* will also be very small now. So the car cannot adjust its direction fast and will collide in the area A. On the other hand, if when *delta_angle* is small, *steering_angle* is large; This is good for the car to adjust its direction fast after passing corner 5, but it cannot pass corner 1. To solve this contradiction, I need to figure out other methods.

### 2.3.6 Method 6

Based on method 3, I let the car turn to the opposite direction, when the car is about to collide with the wall. For this I use the minimum distance between the front half of the car and the environment as a basis. When *min_distance* is smaller than 0.8m, the car will turn a certain angle in the opposite direction of *min_distance*. This method works well, and the car can perform complete laps without collision.

### 2.3.7 Method 7

I also figure out another method, which can also work well. This method is based on method 3 and keep the direction of the car always parallel to the wall to a certain extent. For this I use the direction of *min_distance* as a basis. When the direction of *min_distance* too close to the direction of the car (540$^{th}$), I will make the car turn to the opposite direction of the *minimum_distance*. So that the direction of *min_distance* deviates from the direction of the car. The only disadvantage of this method is that the car will constantly swing from side to side with high speed, like 4m/s.

## 2.4 mux

In this lab I create a node called "*mux*". This node subscribes six topics: "*/drive_AEB*", "*/mode*", "*/drive_r*", "*/drive_f*", "*/drive_p*" and "*/drive_k*"; and publishes one topic "*/drive*". This node can according to the data from topic */mode* select the message stream of one of the driving nodes (*keyboard*, *random*, *gap* and *PID*) and repeat them to the */drive* topic. The node can also unconditionally execute commands from the topic */drive_AEB*.

In main function I set the initial mode is *keyboard,* and when the node *mux* receives new mode data from topic */mode*, it will call the function *callback_mode()* to record the new mode data in the global variable *mode_from_k*. In addition, once *mux* receives a new message from the topic */drive_r*, */drive_f*, */drive_p* or */drive_k,* it will call the corresponding function: *callback_r()*, *callback_f()*, *callback_p()* or *callback_k()*. In the corresponding function, *mux* will according to the current value of the global variable *mode_from_k* check whether the mode matches. If the mode matches, *mux* will through */drive* publish the received data of speed and steering angle from the corresponding topic. But once *mux* receives a new message from */drive_AEB*, it will immediately repeat this message to */drive* without checking any conditions and then set *mode_from_k* to mode *keyboard* in the function *callback_AEB()*.

## 2.5 AEB

In this lab I create a node called "*AEB*". This node subscribes two topics: "*/drive*" and "*/scan*"; and publishes one topic "*/drive_AEB*". This node can detect an incoming collision with the *time_threshold* being 0.25 when the car is moving forward. And with the *time_threshold* being 0.5, this node can detect an incoming collision when the car is moving forward or backward. Once an incoming collision is detected, this node will make the car stop immediately and enforce the module to solely consider inputs from the *keyboard* node.

Once the node receives new data from the topic "*/drive"*, it saves the current speed of the car in the global variable *velocity_nominal* by calling the function *callback_velocity()*. Later we will use *velocity_nominal* to calculate *TTC* (time to collision).

In addition, the node continuously receives the radar data from the topic */scan* and calls the function *callback_scan()*. In this function, the data from the topic */scan* are first copied into the vector "*temp*". Then we compute the *TTC* for every beam returned by Lidar. For this we need to first calculate the projected speed for each beam direction:

$$velocity\_projected \;=\; velocity\_nominal \;*\; cos\,(delta\_angle) \tag{2.5.1}$$

*delta_angle* is the angle between each beam direction ($i^{th}$) and the direction of the car ($540^{th}$):

$$delta\_angle \;=\; (i - 540) \;*\; scan \rightarrow angle\_increment \tag{2.5.2}$$

Then we can use the radar scanning distance *temp[i]* and *velocity_projected* in each beam direction to calculate *TTC*:

$$TTC \;=\; temp[i] \,/\, velocity\_projected \tag{2.5.3}$$

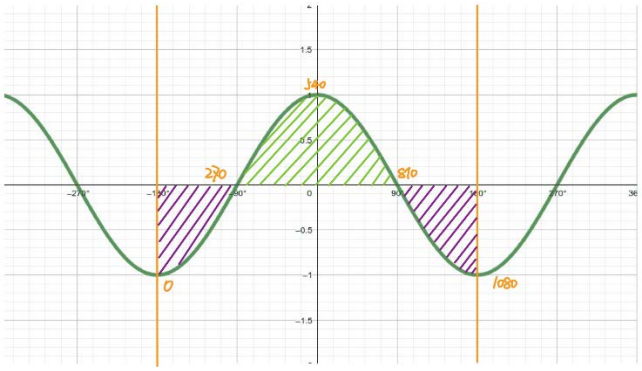


Figure 2.5.1: *cos(delta_angle)*

First, let us analyze the changes in *cos(delta_angle).* In Figure 2.5.1 we can see, *cos(delta_angle)* is positive when the beam directions are [270$^{th}$, 810$^{th}$]; and *cos(delta_angle)* is negative when the beam directions are [0$^{th}$, 270$^{th}$) or (810$^{th}$, 1079$^{th}$]. Now let's consider two different situations: one is that the car is driving forward; the other is that the car is driving backward. When the car is driving forward, the speed of the car is positive, then *velocity_projected* and *TTC in* [270$^{th}$, 810$^{th}$] are positive and in [0$^{th}$, 270$^{th}$) or (810$^{th}$ , 1079$^{th}$] are negative. Since the car is moving forward now, we only care about the upcoming collision of the front half of the car, that is, the beam directions are [270$^{th}$, 810$^{th}$]. So I use the condition "*TTC > 0*" to select this range. On the other hand, when the car is reversing, the speed is negative, then *velocity_projected* and *TTC in* [270$^{th}$, 810$^{th}$] are negative but in [0$^{th}$, 270$^{th}$) or (810$^{th}$, 1079$^{th}$] are positive. Since the car is moving backward now, we only care about the upcoming collision of the back half of the car, that is, the beam directions are in [0$^{th}$, 270$^{th}$) and (810$^{th}$, 1079$^{th}$]. I can also use "*TTC > 0*" to select this range. So in short, "*TTC > 0*" can help us choose correct range, whether the car is driving forward or backward. Once the node detects an upcoming collision (*TTC > 0 && TTC < time_threshold*) in the loop, it will set the speed and steering angle to 0 and publish this message through topic */drive_AEB*. In this way, the node only publishes necessary and important messages. Then the loop will end because there is no need to calculate the *TTC* in other beam directions. Once the node *mux* receives the new message from topic */drive_AEB,* it will immediately publish the speed and steering angle data in */drive_AEB* through the topic "*/drive*", and set the mode to *keyboard* mode.

When we set *time_threshold* is 0.5 (please note that the *time_threshold* in *AEB.cpp* is currently 0.25, change it to 0.5 to test), the node *AEB* can detect the upcoming collision well, whenever the car is moving forward or reversing. However, with *time_threshold* being 0.25, *AEB* can only work well when the car is moving forward. It cannot detect an incoming collision when the car is moving backward. I haven't figured out the reason, but I guess the problem comes from the simulator.

I must emphasize that when you end testing the node *AEB*, please adjust *time_threshold* back to 0.25, because other nodes, especially the node "*gap*", have been only fully debugged and tested when *time_threshold* is 0.25.

## 2.6 PID

In this lab I create a node called "*PID*". This node subscribes two topics: *"/drive*" and "*/scan*"; and publishes one topic "*/drive_p*". This node adopts PID control, allows the car to always maintain the distance of 1.5m from the right wall, and run the complete laps smoothly without collision at a speed of more than 2m/s.

Once the node receives new data from the topic */drive*, it saves the current speed of the car in the global variable *current_velocity_nominal* by calling the function *callback_drive()*. This is for the needs of subsequent calculations (here we do not directly use the fixed speed set by this node *PID* for the car, because the car may switch from other modes to *PID* mode).

In addition, the node continuously receives the radar data from the topic */scan* and calls the function *callback_scan()*. In this function, the data from the topic */scan* are first copied into the vector "*temp*". Then in order to obtain the minimum distance between the car and the wall on the right (*min_distance_to_rightwall*) and the direction of this minimum distance (*index_min_distance_to_rightwall*), the node looks for the minimum value from the 1$^{st}$ data of the vector "*temp*" to the 540$^{th}$ data (the 1$^{st}$ data to the 540$^{th}$ data of vector "*temp*" correspond to the radar data on the right side of the car).

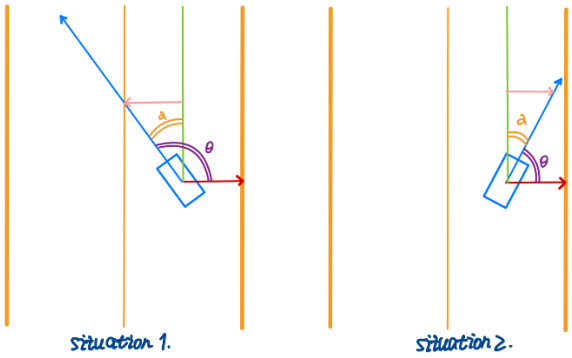


Figure 2.6.1: Two situations about calculating the anticipated distance

Then the node starts to calculate the error. *Error* consists of two parts: one is the difference between the objective distance (*distance_objective* = 1.5m) and the actual distance (*min_distance_to_rightwall*); the second part is the anticipated projection distance (*distance_anticipate*). Here we consider two different situations in Figure 2.6.1. In Situation 1, the direction of the car (blue line in *Figure 2.6.1*) is biased to the left. Therefore, the angle *Theta* between the direction of the car and the direction of the minimum distance from the right wall (*index_min_distance_to_rightwall,* red line in *Figure 2.6.1* ) is greater than 90°. So the angle *Alpha* between the direction of the car and the direction of the wall (green line in *Figure 2.6.1*) is equal to

$$\alpha = \theta - \pi/2 \tag{2.6.1}$$

The anticipate distance (pink line in *Figure 2.6.1*) is

$$distance\_anticipate = current\_velocity\_nominal * sin(\alpha) \tag{2.6.2}$$

In this situation, the anticipate distance makes the car away from the right wall. So *error* is

$$error = distance\_objective - min\_distance\_to\_rightwall - distance\_anticipate \tag{2.6.3}$$

However in Situation 2, the direction of the car is biased to the right and *Theta* is smaller than 90°. So *Alpha* is equal to

$$\alpha = \pi/2 - \theta \tag{2.6.4}$$

We also use the Equation (2.6.2) to calculate *distance_anticipate*. But in this situation, the anticipate distance makes the car close to the right wall. So here *error* is:

$$error = distance\_objective - min\_distance\_to\_rightwall + distance\_anticipate \tag{2.6.5}$$

Due to

$$\sin(\pi/2 - \theta) = -\sin(\theta - \pi/2) \tag{2.6.6}$$

we can transform Equation (2.6.5) into Equation (2.6.3) if we use Equation (2.6.1) to calculate *Alpha* in both situations.

Finally, the node has calculated *error* in the function *callback_scan()*, and then I start to design the PID controller to control the car so that it maintains the distance of 1.5m from the right wall when it runs. Firstly, I only use P controller and let *steering_angle* direct equal to

$$steering\_angle = Kp * error \tag{2.6.7}$$

The range of *error* is [-1.5, 1.5] and I set the range of *steering_angle* is [-0.7, 0.7]. In order to prevent the value of *steering_angle* from exceeding this range, I set *Kp* to 0.45. At last, the

node makes and publishes the message (*drive_st_msg*) to the topic */drive_p* in the function *callback_scan()*. With the P controller, the car can run the complete laps smoothly without collision at a speed of 2m/s.

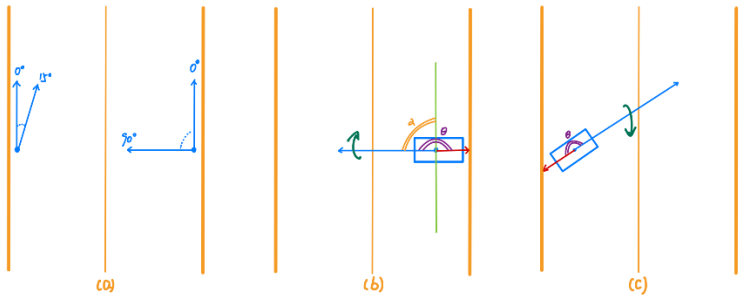


Figure 2.6.2: Different test situations

Then I test many situations where the car is in different initial positions. For example, let the initial position of the car be close to the right wall or left wall (*Figure 2.6.2 (a)*). For the convenience of description, we let *beta_right* represent the angle between the initial direction of the car (blue line in *Figure 2.6.2*) and the right wall; and let *beta_left* represent the angle between the initial direction of the car and the left wall. By the way, if we place the initial position of the car too close to the wall, it may trigger AEB. The car can quickly adjust its posture and run smoothly along the racetrack, when its initial position is close to the right wall and the rang of *beta_right* is [0°, 90°]. However, when the initial position is close to the left wall, the car can only work well with the rang of *beta_left* from 0° to 15°. If *beta_left* is langer than 15°, the car cannot adjust its direction successfully and collide with the right wall. In order to explain this phenomenon, we first discuss why the car can run successfully with the initial position close to the right wall and then think about the reason why the car do not perform well with the initial position close to the left wall. In the Figure 2.6.2 (b) we consider the most extreme case where the initial position is close to the right wall, that is, *beta_right* equal to 90°. Bule line stands for the direction of the car and red line is the direction of the minimum distance from the right wall. In this case, *Theta* is 180° and due to Equation (2.6.1) *Alpha* should be 90°. Therefore, with the current speed of car being 2m/s, *distance_anticipate* equals to 2m. If we assume the *min_distance_to_rightwall* now is 0.5m, then according to Equation (2.6.3) *error* is -1m. So P controller will produce a negative *steering_angle* and make the car turn right. With the help of this negative *steering_angle*, the car can quickly adjust its direction. But when the initial position close to the left wall and *beta_left* is bigger than 15° (*Figure 2.6.2 (c)*), the situation is very different. In this case, the direction of *min_distance_to_rightwall* (red line in *Figure 2.6.2*) no longer points to the right wall, but to the left wall. Because *Theta* is the angle between the direction of the car and the direction of *min_distance_to_rightwall*, *Theta* equals to 180° now. Therefore, due to Equation (2.6.1) *Alpha* should be 90° and with the current speed of the car being 2m/s, *distance_anticipate* equals to 2m. If we assume the *min_distance_to_rightwall* (red line in *Figure 2.6.2,* points to the left wall) is 0.5m now, then using Equation (2.6.3) we can calculate that, *error* is -1m. That means, P controller will let the car turn right and make the posture of the car worse than before until colliding with the right wall. To completely solving this problem is hard, but I figure out a method, which can alleviate the problem (see line 35 ~ 38 in *PID.cpp*). After P controller let the car turn right and makes the situation worse, the *distance_anticipate* will become larger and at some point make *error* smaller than -1.5m. Therefore, once *error* is less than -1.5m, I will make the error be 1.5m, so that P controller will make the car turn left. In this way the car can adjust its direction. This method can only handle the cases where *beta_left* is not greater than 60° with the speed of 2m/s (when we increase the speed, the feasible range of *beta_left* will shrink). Because when *beta_left* is bigger than 60°,

the situation is too bad that the car does not have enough distance and time to adjust its direction.

We have discussed enough about P controller. Next, I gradually increase the speed of the car. The car will start to sway from side to side during running and become unstable. And the higher the speed, the greater the swing of the car from side to side. For example, when the speed is 2.8m/s, we can observe a significant swing. This kind of sway can even cause the car to collide with the wall. To solve this problem, we add the D controller on the basis of the P controller. D controller performs control according to the rate of change of *error*. So I add a loop in main function and set the fixed frequency to 10HZ. In this way the rate of change of *error* is just the difference between *pre_error* and *error.* And we need to update the value of *pre_error* every cycle*.* Now the *steering_angle* equal to

$$steering\_angle = Kp * error + Kd * (pre\_error - error) \quad (2.6.8)$$

In order to find an appropriate value of *Kp*, I increase it from 0 and then observe the influence of D controller on the trajectory of the car. When *Kp* is too small, D controller cannot well restrain the side-to-side swing of the car. On the other hand, with too large *Kp,* D controller will excessively affect the work of P controller and make the swing amplitude increase. When the speed is 2.8m/s and with default initial position of the car, 0.20 is a good choice for *Kp*, it will reduce the swing amplitude and the car can run more smoothly than before.

Finally, let us discuss the impact of I controller. I controller performs control based on the accumulation of past errors. Usually the role of I controller is to eliminate the residual error. But in fact, we do not need this role here. When we add I controller to PD controller and make *steering_angle* equal to

$$steering\_angle = Kp * error + Kd * (pre\_error - error) + Ki * sum\_error \quad (2.6.9)$$

I controller will produce a large value compared to PD controller and make the car very unstable. The car will sway violently from side to side and finally hit the wall (*Ki* = 0.10).