Design of Robotic Systems

The Parallel Parking Algorithm

Lin Li

1. Introduction

In lab4, we tried to let the robotic car do a parallel parking between two rectangular shape obstacles. We first simulated the motion using MATLAB code and then applied the algorithm on the real car.

In our simulation, we were assuming that the obstacles are located at right top corner and left top corner of the box and the robotic car would start somewhere in the left half space of the box(Figure 1).

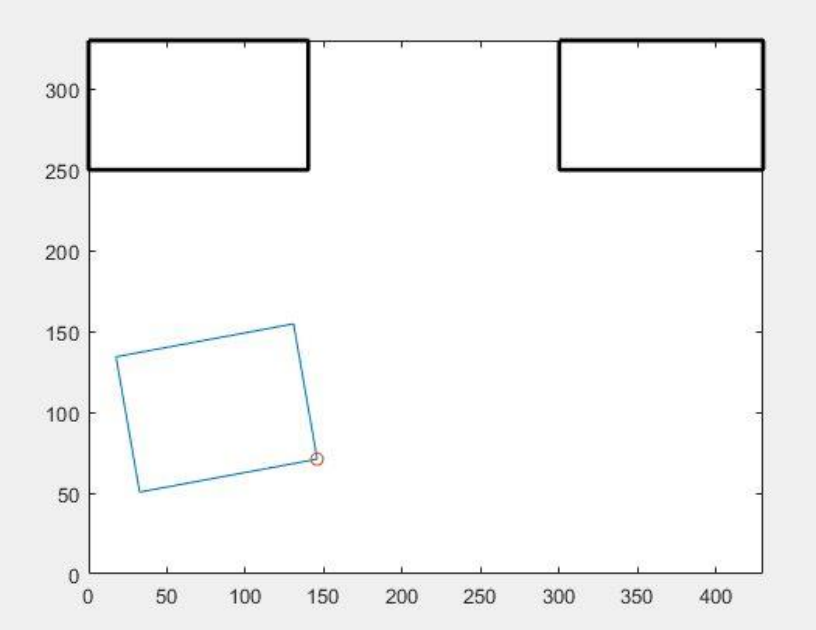

Figure 1

As it can be seen in the figure, we set our origin at the left bottom corner. One unit from each of the axis corresponds to 1mm in the real world. The rectangles with black borders are the obstacles, the one

with blue border is the robotic car. The small red circle at one of the corners of the car indicates the spot where we put the sensors.

In this specific case, our motion planning algorithm is responsible to calculate an optimal path for the car to successfully finish his parallel parking mission without hitting the obstacles or the walls.

2. Method

Our motion planning algorithm is made up of two core functions called “Get_Milestones” ,“Get_Control_Inputs” and several small helper functions. In this section, I’m going introduce the mechanism of our algorithm by explaining the two core functions.

Get_Milestones

This function, as its name suggests, is responsible to find a series of milestone positions that the car must reach on its way the destination position.

In the very beginning of the function, we will prove the dimensions of every object in the box that we want the function to consider when it’s doing the calculation(Figure 2).

```
global Cv; %Translational coefficient of the robot
global Cr; %Rotational coefficient of the robot
global box_length; %The dimension of the box, this is the longer side
global box_width;  %This is the shorter side
global left_obstacle_length; %The distance from left to right of the obstacle located on the left top corner
global left_obstacle_width;  %The distance from up to down of the obstacle located on the left top corner
global right_obstacle_length; %The distance from left to right of the obstacle located on the right top corner
global right_obstacle_width; %The distance from up to down of the obstacle located on the right top corner
global car_length; %The dimensions of the robot
global car_width;
global car_semidiagonal; %The distance from the rotating center to the further corner of the car
global safety_distance; %The distance we want the car to keep from the walls and obstacles
global N; %A constant that is used to control the simulation speed, the smaller N is, the faster the simulation.

% Constants in mm or radians
Cv = 120;
Cr = 70 * pi/180;
box_length=430;
box_width=330;
left_obstacle_length=140;
left_obstacle_width=80;
right_obstacle_length=130;
right_obstacle_width=80;
car_length=115;
car_width=85;
car_semidiagonal=sqrt(car_length^2+(car_width/2)^2);
safety_distance=5;
N=30;
```

Figure 2

Then the function will define a “shield” (the red line in figure 3)around the wall and the obstacles near the parking space according to the given sizes of the obstacles and the safety distance that is defined by us.

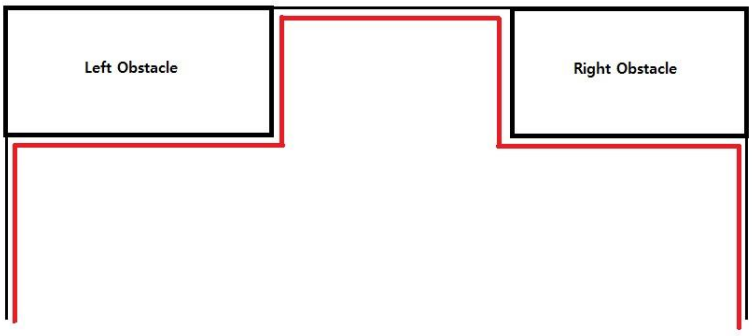

Figure 3

In this case, we defined “safety_distance=5” as it can be seen in figure2, which means that we want the car to always keep 5mm distance from

the obstacles and wall. That corresponds to the distance between the red line and the black lines in figure3.

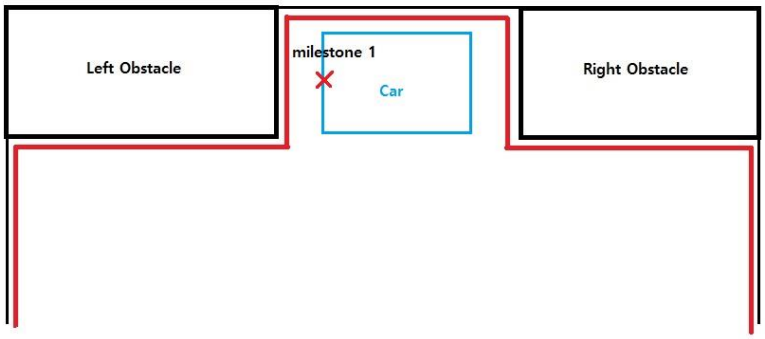

Figure 4

Then the function will find the first milestone position such that when the rotating center(the middle point of the segment that connects the centers of two wheels) of the car reach that position, the car will be the same distance away from both obstacles.(Figure 4)

This milestone 1 is also the destination milestone. In other words, if the car successfully gets to this position, the parallel parking is completed.

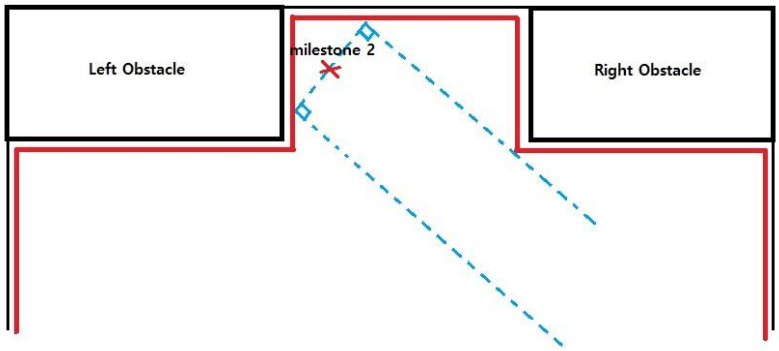

Figure 5

The next step is to find the milestone position that the car must reach when it's parking into the parking space using backward parking from a position near the right obstacle.

Seemingly, the milestone 2 might look very close to the milestone 1 in this specific case when the parking space is pretty narrow. But be ware that they are completely different positions that are found by using completely different calculations.

Milestone 1 can be directly found by looking at the dimensions of the objects, while milestone 2 has to be found by solving an equation.

```
%////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
syms x;
d=box_length-right_obstacle_length-safety_distance;
e=box_width-right_obstacle_width-safety_distance;
c=destination_y;
f=left_obstacle_length+safety_distance;
R=half_car_width;

%This equation expresses the relationships between important parameters on the optimal backward parking path
eqn = ((x-f)/R)*sqrt((x-d)^2+(c-e)^2-R^2)-R*sqrt(1-((x-f)/R)^2) == c-e;

solx = vpa(solve(eqn, x));
%////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Figure 6

Figure 6 shows the equation we used to find the x coordinate of the milestone 2. The equation describes the geometric surrounding of the parking space and the relationships between important constant parameters.

This equation will guarantee that the car won't hit any of the obstacles or the wall while it's moving backward into the parking space and eventually rotate to face the obstacle.

As it can be seen in Figure 6, all of the parameters are defined by the global constants. So if we change the settings for any of the objects, this equation will also change accordingly, and give us the correct answer that works for the new situation.

And this is how we put the "smartness" into our algorithm. No matter how we change the value of the settings, using different box, different

car or different obstacles. It will always automatically figure out an optimal backward parking path for the car.

The milestone 2 is the most important milestone in this function. Once we get this milestone, we can easily find the rest of the milestones.

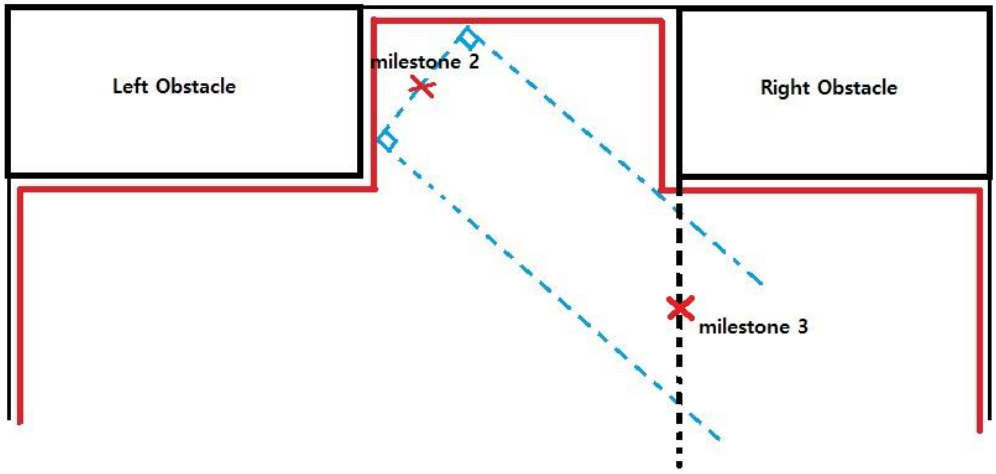

Figure 7

The milestone 3 is the position that the car must reach right before starting to move into the parking space. It can be found based on the coordinates of the milestone 2.

Draw the expected trajectory of the car(blue dash line) and draw a black dash line by extending the left boundary of the right obstacle. The trajectory line will cut the extending line into a segment and the function will define the middle point of the segment as the milestone 3.

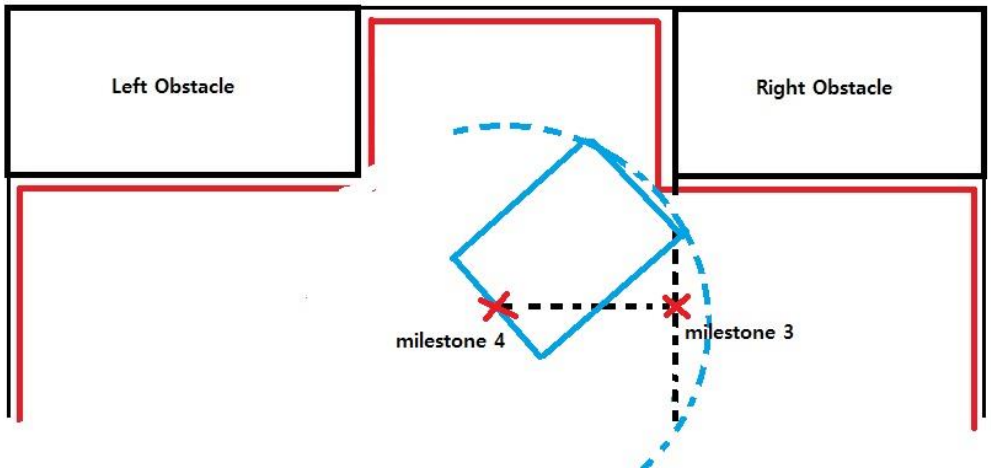

Figure 8

To find the milestone 4, the function will first make a horizontal line that starts from the milestone 3 to the left.

Then it will try to make a circle that has the center on the line with the radius of the “semi-diagonal” of the car and check if the circle has any intersections with the red line.

It will define the center of the circle as the milestone 4 when the circle is tangent with the red line corner of the right obstacle.

The “Get_Milestones” will then, finally, output these four milestone coordinates so that the “Get_Control_Inputs” function can use them as its guidance to generate the proper control inputs for the car.

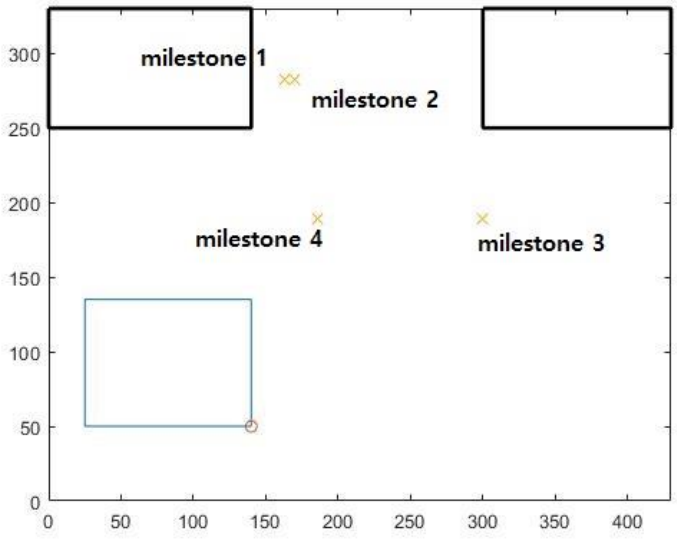

Figure 9

The actual result that is from the “Get_Milestones” looks like this(Figure 9) on the MATLAB screen.

Get_Control_Inputs

After running the “Get_Milestones” function, we know where our robotic car needs to arrive in order to eventually park into the parking space.

Now we need another function to tell us how to actually move the car considering the known milestone positions. And that’s exactly what the “Get_Control_Inputs” function does.

Given the initial state, the “Get_Control_Inputs” function will generate 3 control inputs that can lead the car to the milestone 4 position.

These 3 inputs are organized in an order of “turn, move, turn”, meaning the first input tells the car turn a certain degree and the second input

tells the car to move by a certain distance, and the last input tells the car to turn a certain degree.

After this series of motions, the car is supposed to be the next milestone position in an ideal situation.

Suppose we initially put the car at the position as shown in Figure 9, the "Get_Control_Inputs" function will generate the following inputs.(Figure10)

```
pre_parking_s0_input_1 =

    0.3616


pre_parking_s0_input_2 =

    1.2986


pre_parking_s0_input_3 =

   -0.3616
```

Figure 10

If we check Figure 2, there are two constants "Cv=120" and "Cr=70". Those are the translational constant and the rotational constant of the car, meaning that if the input is "1" the car will move by 120 mm or rotate by about 70 degrees.

Knowing that, if we look at Figure 10, we can see that these 3 inputs will do "turning the car counter clock wise by 70*0.3616 degrees, Moving the car forward by 120*1.2986 mm, turning the car clock wise by 70*0.3616 degrees".

The simulation results of plugging these inputs into the car look like below.

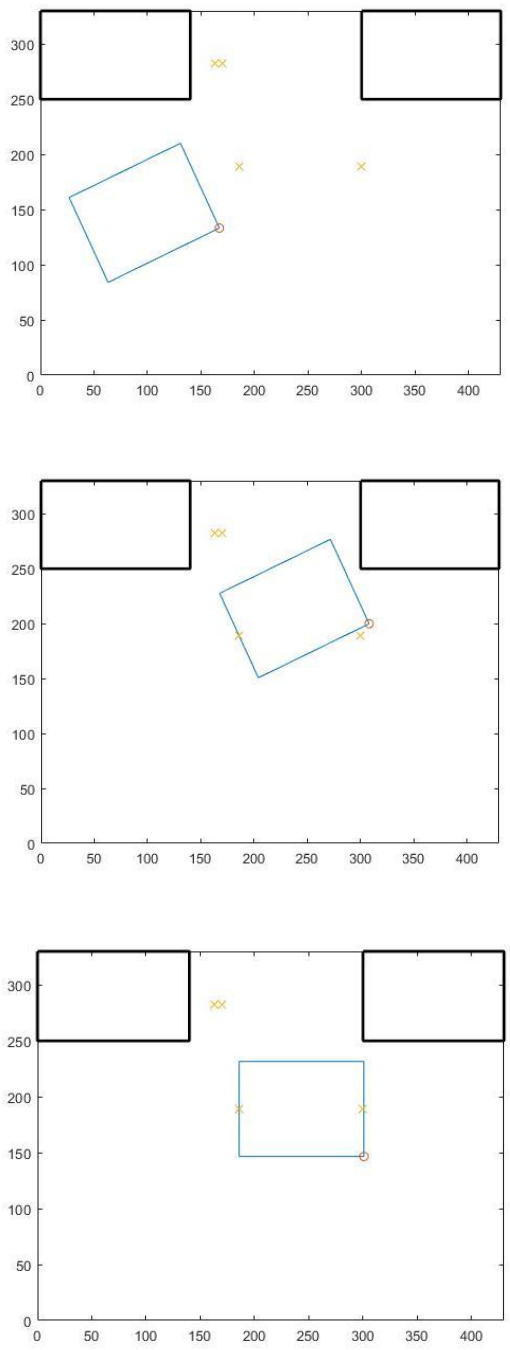

Figure 11

The first picture in Figure 11 shows the result after plugging the first turning input, the second picture shows the result after plugging the moving input, the last picture shows the result after plugging the second turning input.

Similarly, when the function has been noticed that the car has reached the target milestone, it will generate the next series of inputs to lead the car to the next milestone position, and go on and on until the car arrives the destination milestone to complete the parking mission(Figure 12).

To watch the entire dynamic simulation with more details, please check the “Dynamic Simulation” folder on my GitHub.

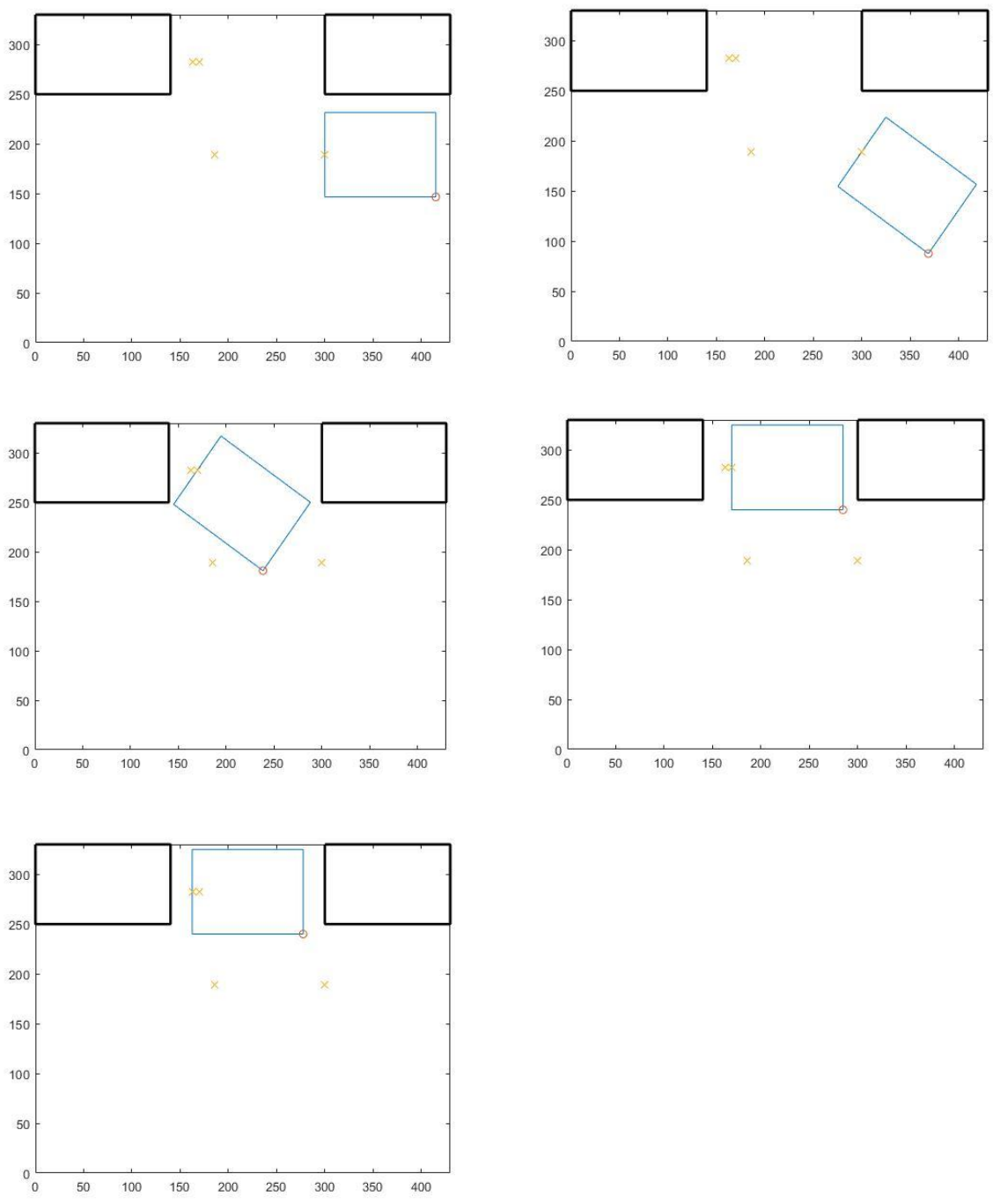

Figure 12

3. Result

Different from the ideal situations we have in the simulation world, in a real world situation, we can't make the car get to the exact milestone point after each series of motions.

Fortunately, we already have a well functioning estimation state calculator with the Karman filter from lab3 to deal with the errors in the real world.

Additionally, we added another insurance to improve the performance of the car, which are namely the "acceptable_radius" and the switch variable "Close_Enough".(Figure 13)

```
%These are the acceptable radius,
%the car will keep moving until it gets to a point within this radius from the target point,
%then move to the next milestone.
global acceptable_radius1;
global acceptable_radius2;
global acceptable_radius3;
global acceptable_radius4;
global Close_Enough;%Determine if it's close enough to the target position.
```

Figure 12

If we, for example, set acceptable_radius1=10. That means, if the car gets to a position that is within 10mm from the first milestone point, we will consider it as a position that is "close enough" to the target or vice versa.

The Close_Enough variable will be set to 1 if it's close enough, otherwise it will be set to 0. This variable is used to determine if the car is close enough to the current target milestone so that it can start moving to the next milestone in the next step.

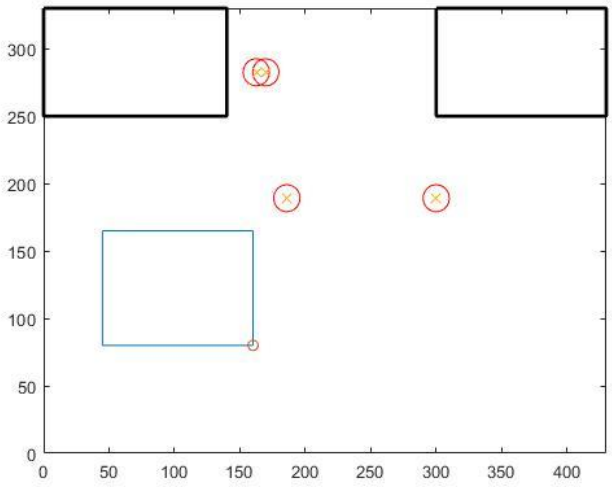

Figure 13

Figure 13 shows the case when we set the acceptable radius as 10mm.

In the main control code, we put a “while” (Figure 14) loop for each of the milestone points. Inside the loop, the “Get_Control_Inputs” function will keep generating the inputs to order the car to get closer and closer to the target spot until the car is within, in this case, 10mm radius from the milestone so that the switch variable “Close_Enough” is set to 1.

```
%Get the milestones according to the current dimension settings
Milestones=Get_Milestones;
N=length(Milestones)/2+1;
for i=1:1:N
Close_Enough=0;

%Estimated_State=[];%Replace this with the real initial reading.

while Close_Enough==0
[u_turn,u_move,u_extra_turn]=Get_Control_Inputs(Estimated_State,i,Milestones);

%Send u_turn,u_move,u_extra_turn to the car.
%Wait util the motion is done.

figure_index=figure_index+1;
if Close_Enough==0

%Estimated_State=[];%Replace this with the real estimation reading.

end
end
end
```

Figure 14

As it can be seen in Figure 14, the new inputs are recalculated in the iteration loops based on the current estimated states.

If the estimated states are reliable and the process errors are decently small, we can expect that this algorithm will give a pretty good result on the real world parallel parking. And it indeed did a good job in our actual tests.

In our real world parking experiments, we have recorded the expected positions and the estimated positions around all of the milestone points and plotted them as figures when the parking is finished. Figure 15 shows the result of one of our experiments.For more details about the result, please check the demo video on my GitHub.

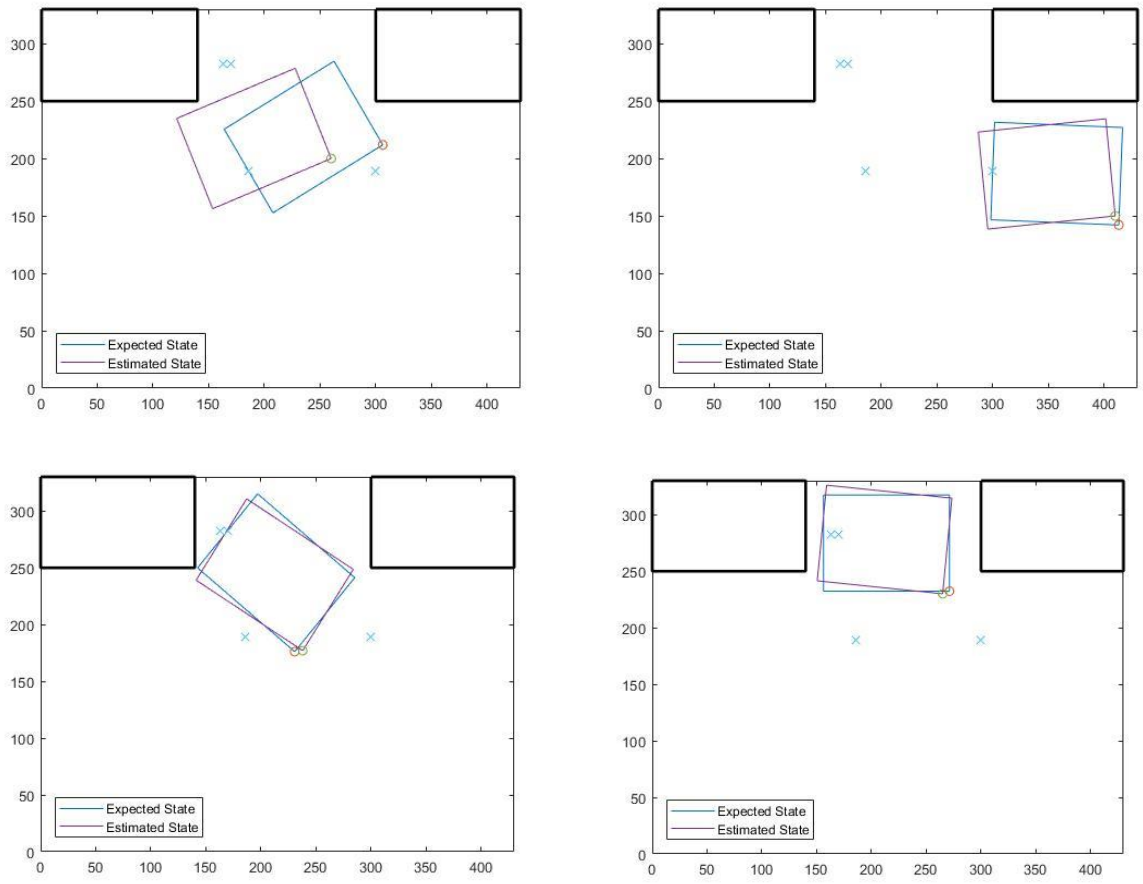

Figure 15