UNIVERSITY OF GOTHENBURG

Autonomous Parallel Parking for Miniature Vehicle in Carolo Cup 2015

Technical Report

Victor Borosean
victor.borosean@gmail.com
Georgi Dungarov
dungarov@gmail.com

March 4th, 2015

IT Department
University of Gothenburg, Sweden

Table of Content

Abstract - Carolo Cup is an international student competition for self-driven miniature vehicles organized annually in TU Braunschweig, Germany. The competition is inspired by the DARPA Grand Challenge. The competition "Carolo Cup" provides student teams from universities the opportunity to develop and implement autonomous model vehicles. The challenge is to realize the best possible vehicle control in different scenarios inspired by a realistic environment. The annual competition consists of presenting the student's skills to a jury of experts from industry and academia, and competing with other student teams. The basic functionalities required for the Carolo Cup competition are lane following, overtaking, intersection handling and parking. This report will be focused on describing the parking concept that was prepared and used for this competition this year. The report covers the theoretical research and the practical implementation.

1. Introduction

This paper describes the parking algorithm that was used for the 2015 Carolo Cup competition. Our team was named MOOSE and was divided into three main teams that were responsible for different scenarios for the competition - Lane Following, Parking and Integration. The parking team was responsible for developing an algorithm for the parking scenario and as a major scope to develop this parking algorithm to suit the competition rules, which will be described in the next section.

Parallel parking is the second dynamic discipline in the competition. According to the competition regulations[1], the ability to park is valued as much as lane detection, because of the interaction of different sensors implemented on the car. The competition regulations assume that the miniature car will be able to detect an empty parking spot on the parking strip on the right side. The car has to be able to park as fast as possible without touching any objects around it.

This document covers the software and the hardware aspect of the parallel parking scenario for Carolo Cup competition. Creating a successful algorithm took several challenged into consideration in order to make a suitable algorithm that meets the regulations[1]. These challenges included: taking the correct measurements of each parking spot, correct evaluation of each spot, detecting if there are objects inside the parking spot and defining a set of maneuvers that allows the car to park without collisions. In order to make the parking sequence dynamic, the sensors had to detect objects in real-time and the car should react to them in a quick and robust manner.

Two people were responsible for the parking algorithm: Victor and Georgi. Their various contributions are summarized below.
Victor was responsible for:

1. Designing the parking algorithm
2. Maintaining the parking algorithm during the entire project.

Georgi was responsible for:

1. Map the infrared sensor values by extracting the average number of the readings in order to have accurate data for distance.
2. Redesign the bash script for both cars in order to automate the starting process of the algorithms
3. Design of the LCD module menus and functionalities on the Wooden car

While designing and maintaining the algorithm we encountered many challenges. Our key challenges were the following:

1. Since the lane following was in development until the competition date, we had to establish a way of keeping the car drive straight at least for the 7 meters.
2. The braking distance was too long. This caused the car to collide with the objects around the parking spot.
3. Another issue we had, was the fact that the object in the parking spot was not always detectable, since the range of the IR was just 25 cm of accurate data.
4. The values from the sensors were delayed by the time it reached the Driver.cpp was reading them.

The solutions to these challenges were:

1. We changed the wheels to ones with softer tires. That also helped achieve a shorter braking distance.
2. We replaced the IR sensor on the *MOOSE* car, which helped detect the object in the parking spot.
3. We cleaned the buffer after each value read to achieve a greater response time while performing the sequence.

In this paper we describe the regulations of the Carolo Cup 2015 competition, followed by an overall description of the parking concept including the sensor layout and an architecture diagram. Later, we describe the parking algorithm in detail for each step along with a code snippet and a picture illustrating the position of the car at that step. This section also describes and illustrates the state machine. Also, included are some of our observations and the reliability of the algorithm. The last section is the conclusions we have gathered from this experience and suggestions for future work.

2. Regulations

In this questions we will mention the regulations that were included in the 2015 competition.

2.1 Parking Strip

According to the regulations the parking strip is 30 cm wide and is situated on the right side of the main road. The obstacles positioned on the parking strip are 100 mm high and can be

placed 2cm to 20 cm from the margins of the parking strip. Besides the possible parking spots, the distance between the obstacles can vary up to 40cm. At the end of the parking strip, on the main road, two big obstacles will be situated to indicate that the parking strip has ended and if the car did not park in any of the spots, it should stop before colliding with the obstacles [1]. A possible layout of a parking strip is displayed in Figure 1.

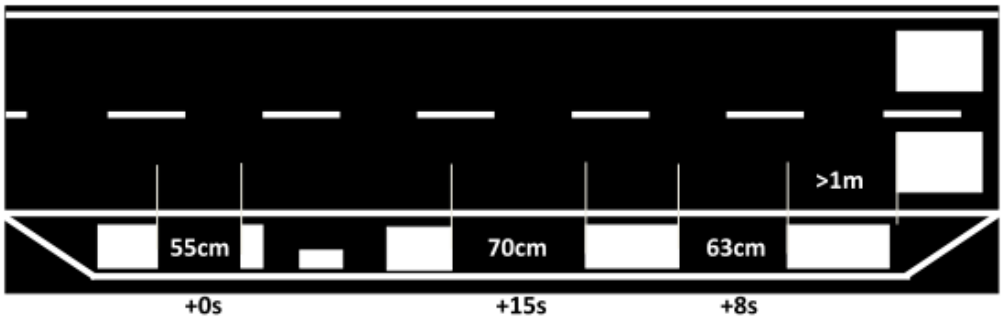

*Figure 1.*Possible parking layout (from Carolo-Cup Organization Team, 2015)

2.2 Parking Spot

The parking spot can be of three different lengths: 55cm, 63cm and 70cm. These parking spots can be arranged in a random order. The obstacles are situated between the white lane marking of the main road and by the solid white lane on the opposite side.

2.3 Sequence

A parking sequence states that the car has to drive on the right lane of the main road and it has to park in the desired parking spot, which is one out of three available. Ideally, the parking sequence has to be performed without colliding with any obstacles. The car is positioned on the right lane before the starting, 4cm-wide, line and it has to start driving when the jury presses the start button. The vehicle has to drive forward and start seeking the desired parking spot situated between stationary obstacles. When such parking spot is found, the right yellow indicators have to start blinking. Once the car is parked, all the directional indicators have to start blinking, indicating that the sequence is over [1].

2.4 Attempts

Each team has a right for two consecutive parking attempts, one after another. The teams will start performing according to order that will be predefined at the beginning of the competition. [1]

2.5 RC Mode

RC mode is not allowed for the parking discipline. [1]

2.6 Time

The time allowed for the parking sequence is stipulated to be the time between the crossing of the start line and the blinking of all directional indicators. [1]

2.7 Scoring

Maximum score will be given to the team that performs fastest. The score will be given to each run individually. The final score will consist of the average of each run. [1]

2.8 Penalties

The attempt will be considered invalid in the following cases:

- when the distance between the car and the obstacles is less than 1 cm;
- when the vehicle is not within the border lines of the parking strip after the parking sequence is over;
- the parking process takes longer than 30 seconds.

Five additional second will be added to the run-time in the following instances:

- of each collision the car will make;
- when the indicators are not working accordingly;
- when crossing with one wheel the white line;
- when the angular deviation is more that 5 degrees compared to the road.

8 addition seconds will be given for teams that will be parking in 63cm gap, and 15 seconds to the 70cm gaps. [1]

3. Concept

Our parking algorithm is based on the algorithm used previously in the Carolo Cup competition of 2014[2]. We have extended the switch statements used in the last year and have changed a few maneuvers. Extending the current switch statements to include more of them has allowed us to test the parking algorithm by assessing each step.

An overall concept of the algorithm starts by driving the car straightforward, while it scans the right side, with the help of IRs, for possible parking gaps. Once the parking gap is found, the

car measures the gap's length and decides if the gap is long enough for the car to park in. Initially, the parking algorithm was designed for the *MOOSE* (wooden) car, aiming for 630 mm gap since the steering angle of the car was not enough to aim for the 550 mm parking spot. Therefore, the *MOOSE* car was looking for the 630 mm gap specifically and not for the other two. If that parking spot was not found the car would continue driving straight until it approached the obstacle in front of it (according to the regulations). If the parking spot with the length of the 630mm was found the car will start parking.

The concept of the parking algorithm is divided in two main parts. These two parts are called ***DRIVING STATE*** and ***PARKING STATE***. The ***DRIVING STATE*** is responsible to drive the car until it finds the right parking spot. The ***PARKING STATE*** will take over after the parking spot is found and park the car successfully in the desired gap. In a more detailed description we will discuss the logic of the concept of those states.

3.1 DRIVING STATE

DRIVING STATE starts with the car driving straight and immediately looks to identify the first box on the right side. Once the box is found it looks for the end of the box or the beginning of the parking spot. Once the beginning of the parking spot is identified, the data from the wheel encoder will be stored in a variable and the sequence will look for the beginning of the second box, which would signal the end of the parking spot. Once the parking spot has ended another variable will store the new data from the wheel encoder. The difference between the values of these two variables will be analysed in order to see if the parking spot has the necessary length for the car to park in it. If the parking spot is not long enough it will continue to drive straight, otherwise it will go into the ***PARKING STATE***.

3.2 PARKING STATE

PARKING STATE starts the *BACKWARDS_RIGHT* maneuver that will get the car in the parking spot. It is the only hard-coded part of this state and it was calculated from multiple testing attempts. The next state, *BACKWARDS_LEFT*, has the role to start aligning the car in the parking spot and stop it when the box behind the car is identified at a certain distance. The next sequence is to drive forward with steering maximum to the right and continue to align the car in the spot. Once the object is identified in the front, the car will go backwards and finish aligning. At this moment the parking sequence will be over.

3.3 SENSORS

The *MOOSE* car can use all sensors in order to detect an object or measure distance and an infrared sensors to align the car in the right position. In this algorithm solution the team used three sensors. One infrared sensor on the right side and two ultrasonic sensors placed respectively on the front and the back of the car. The ultrasonic sensors are used to measure the distance between the car and the objects when the car is inside the parking spot. Depending on the distance the car switches between states in order to align. While in

DRIVING STATE, the front ultrasonic is used as a safety brake in case no suitable parking spot is found. The infrared on the front-side is used only to measure the gap and find which parking spot is suitable. **Figure 2** is an illustration regarding the place where the IRs and USs are positioned on the car:

Figure 2. Sensor Layout

3.4 System architecture

The software architecture consists of Driver and Proxy as high-level components and another component running on the Arduino board as a low-level component [2]. The Driver component runs at 40Hz and Proxy at 60Hz [2]. The program on the Arduino board listens for commands through the serial port every 12ms to ensure the proxy gets correct data. The code is manipulated from the LCD touchscreen display attached on the back of the car. The touchscreen sends data through the Serial port to a bash shell script. The script then starts

the Driver and the Proxy. Simultaneous start of the two components will not start the algorithm due to difference in initialization of the two components. That problem is avoided by starting the Proxy first and then initializing the Driver a few seconds later.

The Arduino board is used as a sensor data collector and as a control module for the servo and the motor. Once the sensor data is collected it is mapped accordingly and sent to the proxy, which will process to the Driver component. The algorithm's behavior is defined based on the sensor data received. The Driver sends back to the Proxy that a change in the speed or the steering angle is needed. Proxy component is the communication tool between Arduino and Driver component. Once the Arduino receives a signal it processes it and adjusts the wheel angle or the speed of the car.

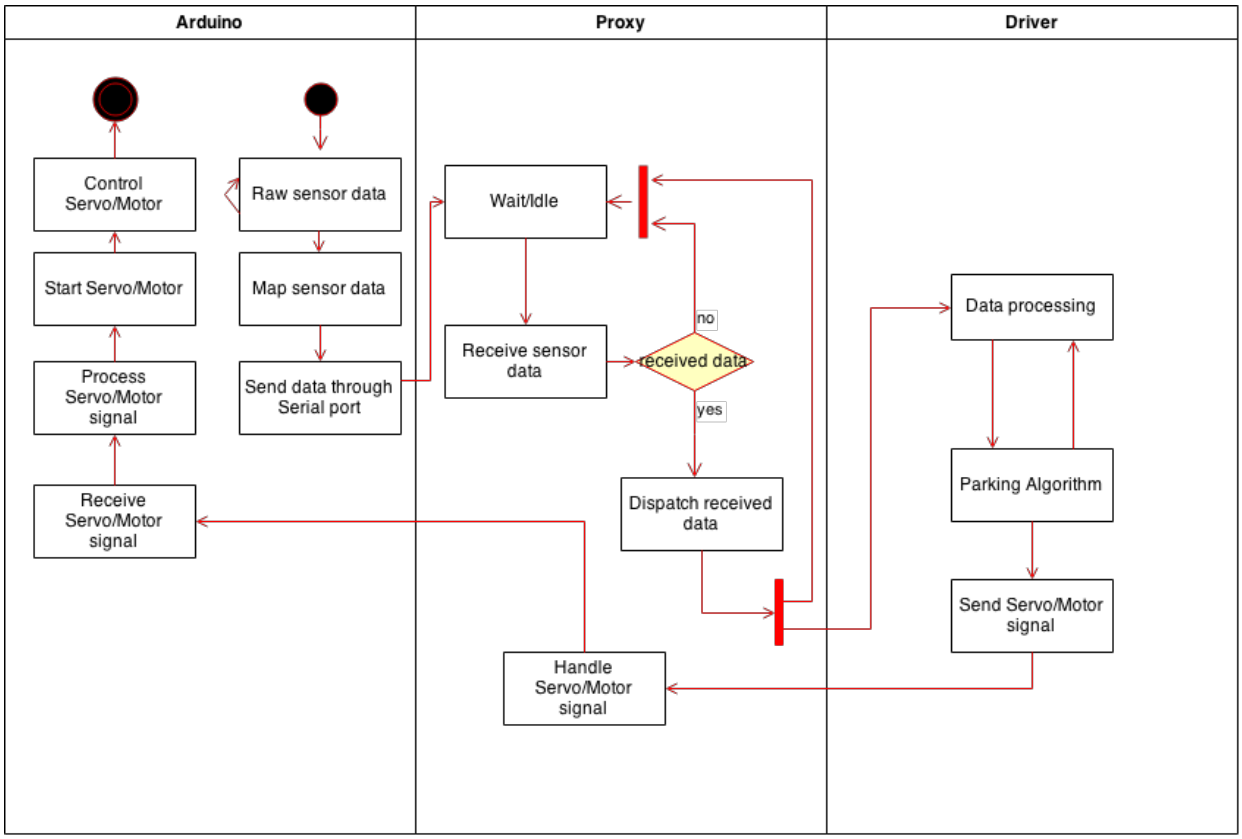

Figure 3. Software architecture

4. Algorithm

4.1 Variables

Before going into the algorithm itself, it is a good idea to introduce the variables that are used together along with their purposes. An improvement from last year was adding the possibility to adjust the variables and their values directly from the configuration file. This would

eliminate the need to recompile the code every time and only the supercomponent needs to be restarted. An image will be shown on each state illustrating the position of the car when the current state is over and the next one is triggered. Also we will include a snipped related to the actual sequence next to the picture. That will help understanding the outcome of the code. In *Figure 4* we have illustrated the state machine where we demonstrate the logic behind the parking algorithm.

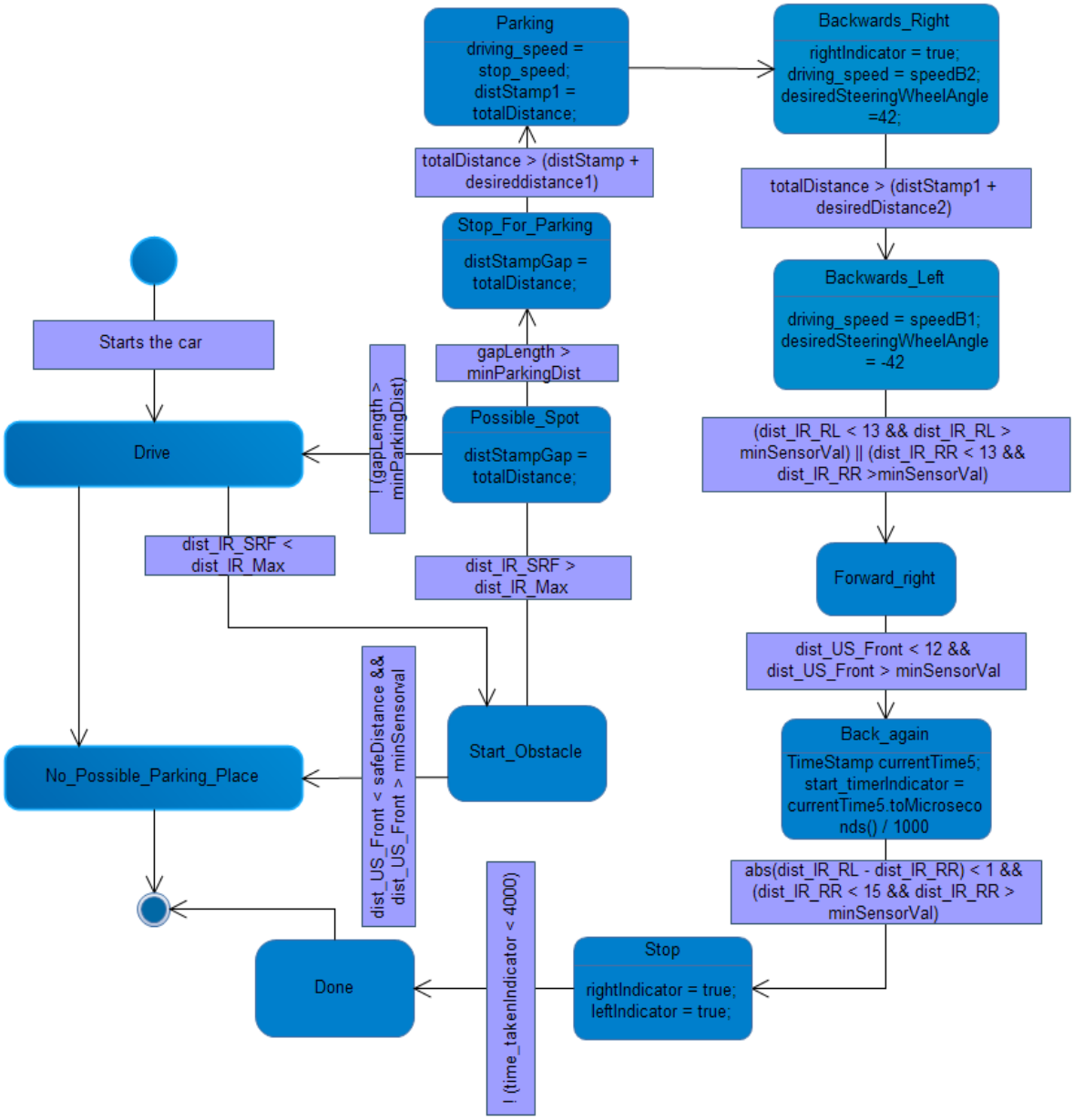

Figure 4. State machine

4.2 DRIVING STATE

DRIVE is the first state; it drives the car forward until the first box is identified. Once the condition is satisfied, it goes to the next state.

```
if ((dist_IR_SRF > dist_IR_Max || dist_IR_SRF < minSensorVal)) {
        driving_state = POSSIBLE_SPOT;
        distStampGap = totalDistance;
        }
```

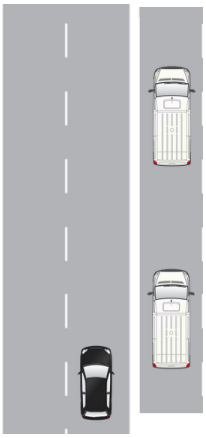

Figure 5. Car in the DRIVE state

START_OBST is the state where the car is traveling alongside the first box. While driving, it scans for the first object to end and once the value from the Side-Right-Front-IR is greater than maximum IR distance (which can be changed in the configuration file) (*dist_IR_SRF > dist_IR_Max*), it will understand that the parking gap has started and it will record the the value from the wheel encoder (*distStampGap = totalDistance*) in order to measure the length of this parking gap. After that it will jump to the next state.

```
if ((dist_IR_SRF > dist_IR_Max || dist_IR_SRF < minSensorVal)) {
        driving_state = POSSIBLE_SPOT;
        distStampGap = totalDistance;
        }
```

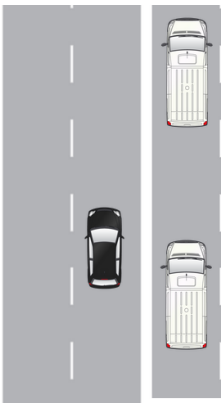

Figure 6. Car in the START_OBST state.

POSSIBLE_SPOT will make the car continue driving straight and measure the parking gap. This means that if Side-Right-Front-IR is less than the maximum value, which is set in the configuration file, the next object will be identified and the gap length will be calculated (*gapLength = totalDistance - distStampGap;*). Since we are aiming for a concrete gap distance we will compare it to see if it is long enough for us to park in it.

```
if(dist_IR_SRF < dist_IR_Max && dist_IR_SRF > minSensorVal){
        gapLength = totalDistance - distStampGap;
        if(gapLength > minParkingDist){
                distStamp = totalDistance;
                driving_state = STOP_FOR_PARKING;
                }else{
                 driving_state = DRIVE;
                }
        }
```

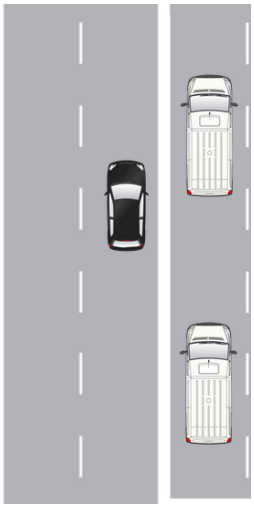

Figure 7. Car in the POSSIBLE_SPOT state.

Therefore if *gapLength* is greater than the minimum possible one (*minParkingDist*), the next distance stamp will be stored and the algorithm will continue to the next state. Otherwise, if the gap is shorter than the minimum value it will start again from the first state.

STOP_FOR_PARKING is the state where the car is preparing its initial position before starting its actual parking sequence. We have to travel straight forward a bit further than the beginning of the second box to be able to get into the gap safely by going backwards. After this distance is reached we will take the next distance stamp, stop and continue to the next state.

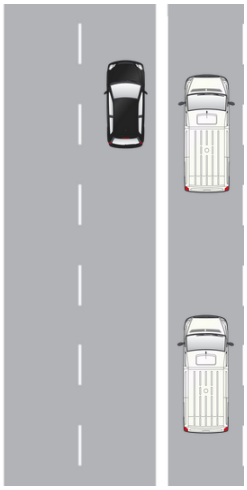

Figure 8. Car position in the STOP_FOR_PARKING state

```
if (totalDistance > (distStamp + desiredDistance1)) {
        distStamp1 = totalDistance;
        driving_state = PARKING;
        driving_speed = stop_Speed;
        }
```

PARKING is the state which calls for the parking function.

NO_POSSIBLE_PARKING_PLACE is one of the states that stops the car if there is a close object identified by the front ultrasonic.

```
if ((dist_US_Front < safeDistance && dist_US_Front > minSensorVal)){
        driving_state = NO_POSSIBLE_PARKING_PLACE;
        }
```

4.3 PARKINGSTATE

BACKWARDS_RIGHT is the first maneuver of the actual parking sequence. According to the regulations, we have to start the right indicator and travel backwards while steering at the maximum angle to the right.

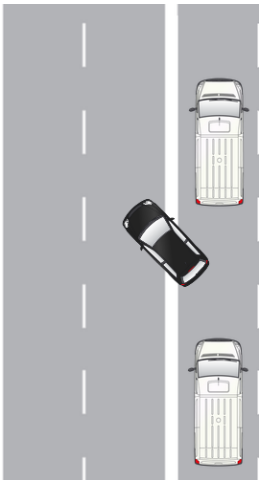

Figure 9. Car position in the BACKWARDS_RIGHT state

```
rightIndicator = true;
driving_speed = speedB2;
desiredSteeringWheelAngle = 42;
```

This state is the only one that is completely hard-coded in terms of reaching the end state. To be more specific, the car will have to travel a specific amount of millimeters before switching to the next step. This amount can also be found and modified in the configuration file. Once this distance is reached, it will jump to the next step and take the next distance stamp.

```
if (totalDistance > (distStamp1 + desiredDistance2)) {
        parking_state = BACKWARDS_LEFT;
        distStamp2 = totalDistance;
        }
```

BACKWARDS_LEFT is the next state. Here the car reduces the backwards speed and steers to the left for alignment.

```
driving_speed = speedB1;
        desiredSteeringWheelAngle = -42;
```

Since the car will be moving already inside the parking gap, it will use all the rear IR sensors to stop before touching the first box.

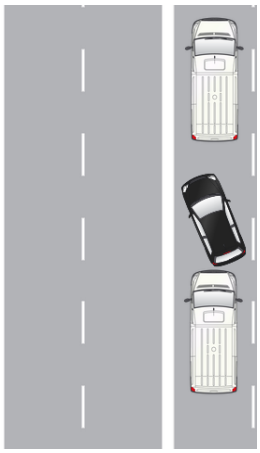

Figure 10. Car position in the BACKWARDS_LEFT state

```
if ((dist_IR_RL < 13 && dist_IR_RL > minSensorVal) || (dist_IR_RR < 13 && dist_IR_RR >
minSensorVal)) {
        parking_state = FORWARD_RIGHT;
        }
```

In this case we are using the rear IRs because they have a faster response time. Once this condition is true we will move to the next state.

FORWARD_RIGHT is the next state where the car will move forward and continue the alignment. The direction is still forward, but at a slower speed, and the steering angle is set to maximum right. If the car gets too close to the second box the front ultrasonic will let it know and the the car will go to the next state.

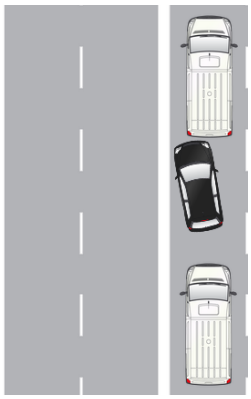

Figure 11. Car position in the FORWARD_RIGHT state

```
if (dist_US_Front < 12 && dist_US_Front > minSensorVal) {
        parking_state = BACK_AGAIN;
        }
```

Also if we want to know at which angle the car is aligned at the moment we can use the formula to identify it. In C language it looks like this:

```
parkAngle = asin((dist_IR_RL - dist_IR_RR)/11.0);
```

BACK_AGAIN is the last maneuver from the parking states. It goes backwards at the slowest possible speed and has the steering angle set to maximum left. Also, it stops the car when it is aligned to the box, or if not, when it is very close to the first box. The alignment is done by calculating the absolute value of the difference of the rear IRs.

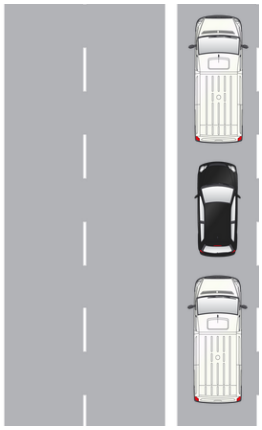

Figure 12. Car position in the BACK_AGAIN state

```
if ((((abs (dist_IR_RL - dist_IR_RR)) < 1) && ((dist_IR_RR < 15 &&
dist_IR_RR > minSensorVal))) || (dist_US_Rear < 10)){
        TimeStamp currentTime5;
        start_timerIndicator = currentTime5.toMicroseconds() / 1000.0;
        parking_state = STOP;
        }
```

When this condition is satisfied, we take a timestamp and we move to the next state. The timestamp is used for later in the next state for the indicators.

STOP is the state where the car stops and does not move anymore, but lights up all the indicators meaning the sequence is over. It is done by using two time stamps (one from the previous state) and an if condition.

```
TimeStamp currentTime6;
time_takenIndicator = (currentTime6.toMicroseconds() / 1000.0)- start_timerIndicator;
if (time_takenIndicator < 4000)  {
        rightIndicator = true;
        leftIndicator = true;
        }else {
        parking_state = DONE;
        }
```

For 4 seconds all the indicators will blink after the sequence and continue to the last state.

DONE is the last state. Here we turn off the indicators and the car remains stopped.

4.4 Observations and encountered problems

- Generally there are some huge issues with this sequence. In the future, it will be good to implement lane following in the ***DRIVING STATE***. This will make the sequence less dependent on the initial position of the car alignment at the starting line. This algorithm requires the car to travel straight and be in the middle of the lane for the entire duration, otherwise the parking will fail! Also, before going to the ***PARKING STATE*** it is necessary to take into consideration the position and angle of the car on the lane before going into the gap. That will also help the algorithm to be more dynamic.

- During the automatization process there was an issue with running the Proxy and Driver component simultaneously. The system registers both processes as initialized, but never starts the Parking sequel. The fix was to add a “sleep” function in the bash script. A setting between 5 to 10 sec gives enough time for the proxy to initialize.

- After each iteration of the algorithm was built, a compilation of tests was done to determine if the implementation of the new iteration or change was successful. During the testing phase, the team noticed a decrease in the performance of the algorithm. The car was improperly evaluating the parking spot’s ending and not performing the parking sequence. In order to find the error we tested each component that was involved in the algorithm. It was determined that the buffer memory was overflowed with values, which caused the IR sensor to work with a crucial delay. That problem was fixed by clearing the buffer.

- The response time of the ultrasonic sensor has a delay that affects the time of execution of the sequence. That sometimes causes collision with objects because the car is too close to an object and the sensors do not have enough time to respond before the car hits the object. Currently, there is no solution to the problem, but a theoretical solution that has not been tested is to set the Maximum Analogue Gain to a lower number. That will reduce the sensor’s response time, which according to its documentation is around 65ms. The lower the number the more rapid the sensor will be. The Gain Register values can be found in the UltraSonic official documentation [3].

- A bash script is used for the automatization of the starting of the parking discipline. The bash script contains not only the components that start the Parking, but the Lane Detection as well. The bash script initializes the supercomponent and the Proxy first and after that, waits for a signal on the serial port before it initialize the next selected component. The selection of the components is done by pressing the physical buttons on the back of the car which sends a predetermined value to the script.

4.5 Algorithm reliability

The parking sequence has not implemented Lanefollowing in its algorithm causing successful parking to depend on a well adjusted position of the car [2]. The reliability of the algorithm was established by performing 10 runs, 8 of which were successful. Samples from the infrared sensor responsible for finding a parking spot are shown in Figure 13-15.

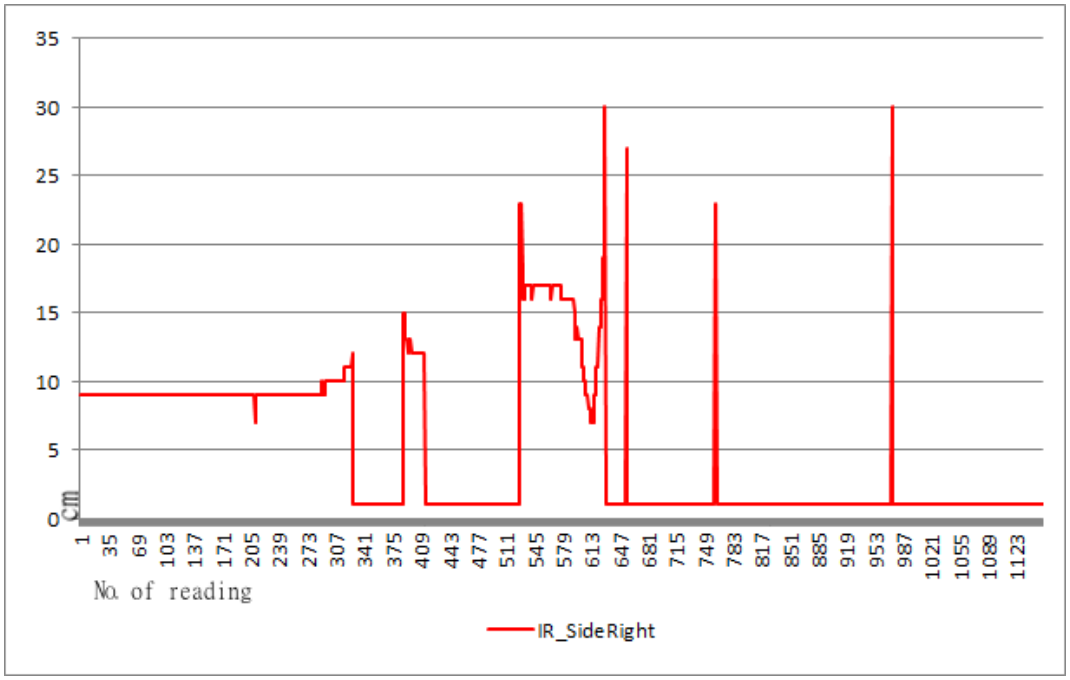

Figure 13. **Parking gap is found with no obstacles detected afterwards**

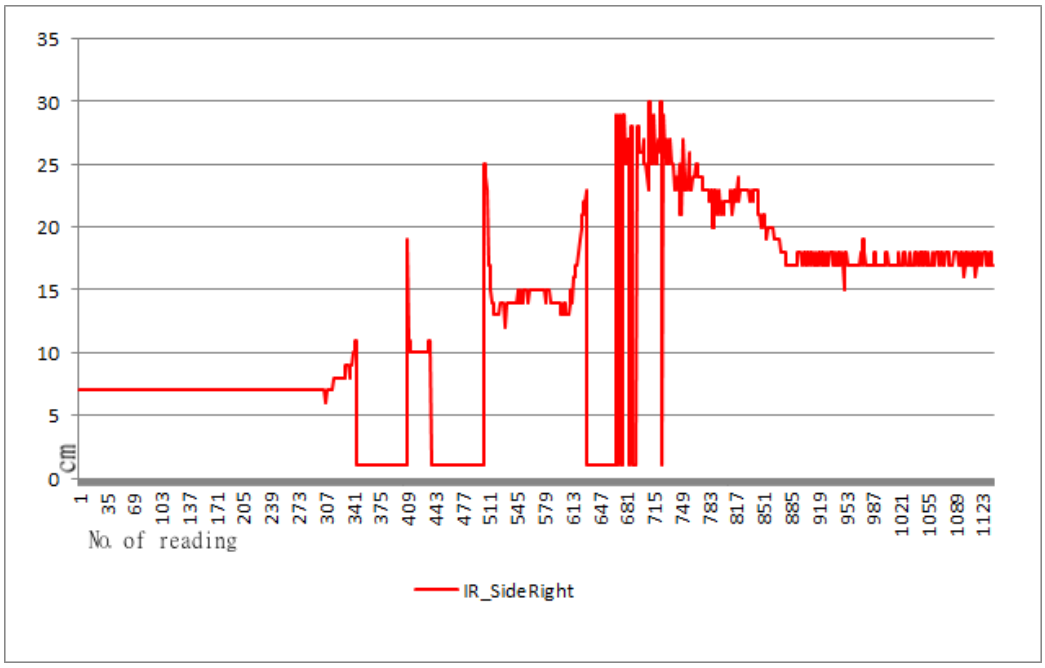

Figure 14. **Parking gap is found and a distant obstacle detected in the end**

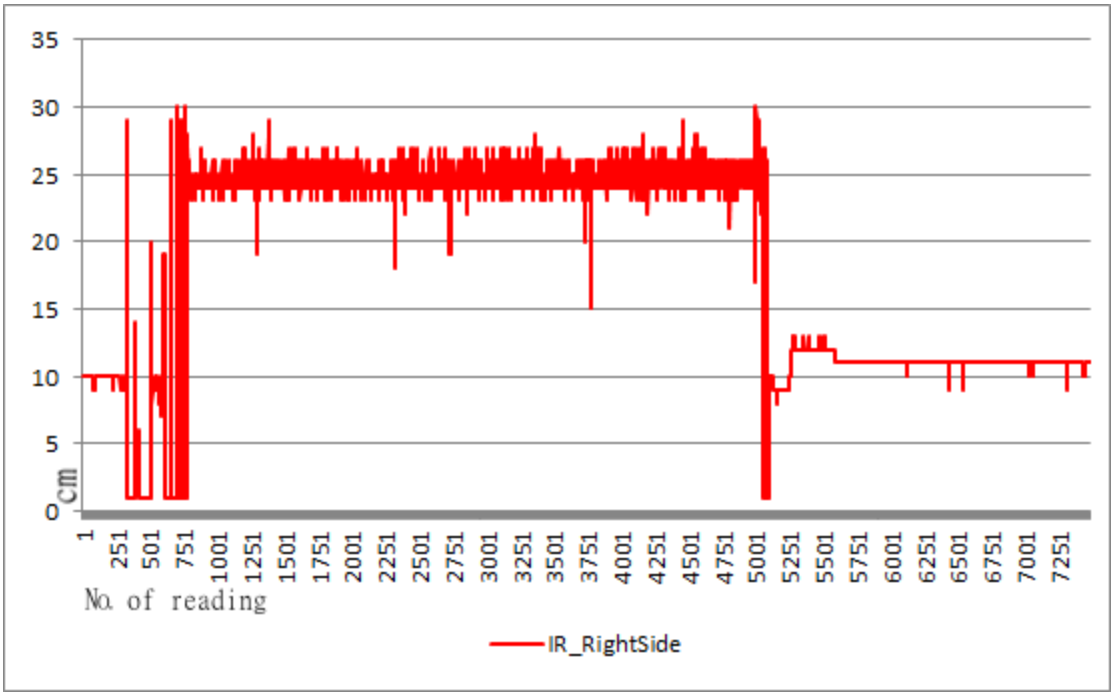

Figure 15. **Parking gap has not been found**

Figure 13 and **Figure 14** have a gap in the range 512-615. In both diagrams the car has parked successfully, even though the end values are different. This contrast is caused by a distant obstacle which in **Figure 14** has been detected. A value of 1 is displayed by the sensor when the reading distance is greater than 25 cm or it is smaller than 5cm. All tests show a similar pattern when the gap is found. **Figure 15** presents an unsuccessful parking. Even though the gap is recognized, the spikes on 2251 and 3751 did not allow the car to see this parking spot as suitable. This caused the car to drive straight ignoring all available spots.

5. Conclusion & Future Suggestions

This document discusses the parking algorithm developed for Carolo Cup 2015. It summarizes the concept of the algorithm together with problems that have occurred and their respective solutions. Furthermore, the report provides suggestion for improvements in the future. The parking algorithm is quite simple and straightforward. It is inspired from the algorithm used last year[1]. The code is well structured and easy to understand and expand. It is also quite easy to modify and test each state in part for different test cases. Based on the previous knowledge we have shared, we can conclude some ideas and suggestions:

- If it is possible to park with high speed, it would be good to aim for a 55cm gap.
- It is required to substitute the steering angle in the DRIVING STATE with lane following, otherwise make sure the car is able to drive straight for at least 5 meters.

- Before starting the designing of the parking, it is necessary to establish a few basic test cases and see if it is possible to rely on the data inputs from the driver.

It is important to know that, even if things look simple and clear, each of these parking steps have to be tested redundantly in order to prove them truly reliable.

According to the regulations that are described and the penalties found in the table, the task of having the car park does not seem very easy. The rules are very strict and with each penalty more time is added to the sequence. The bottom line is that in order to have the maximum possible points for the parking scenario the parking procedure has to: be fast, aim for 55cm gap and avoid any collision with any obstacles. In addition, the car has to be parked parallel to the parking strip margins. Therefore, parking needs a lot of attention during the entire project.

References

[1] Carolo-Cup Organization Team (2015). *Carolo cup rules*. Retrieved from: https://wiki.ifr.ing.tu-bs.de/carolocup/system/files/Hauptwettbewerb2015.pdf

[2] Botev, V. (2014). *Self-driven Miniature Vehicle for Carolo Cup 2014.* Retrieved from: https://github.com/se-research/CaroloCup/blob/master/2014-CaroloCup/Legendary/docs/Victor/cc2014_Viktor_Botev.pdf

[3] Jörg P. (2009). *Ultraschallmodul SRF08*. Retrieved from: https://github.com/se-research/CaroloCup/blob/master/2013-CaroloCup/docs/hardware/UltraSonic/srf08.pdf