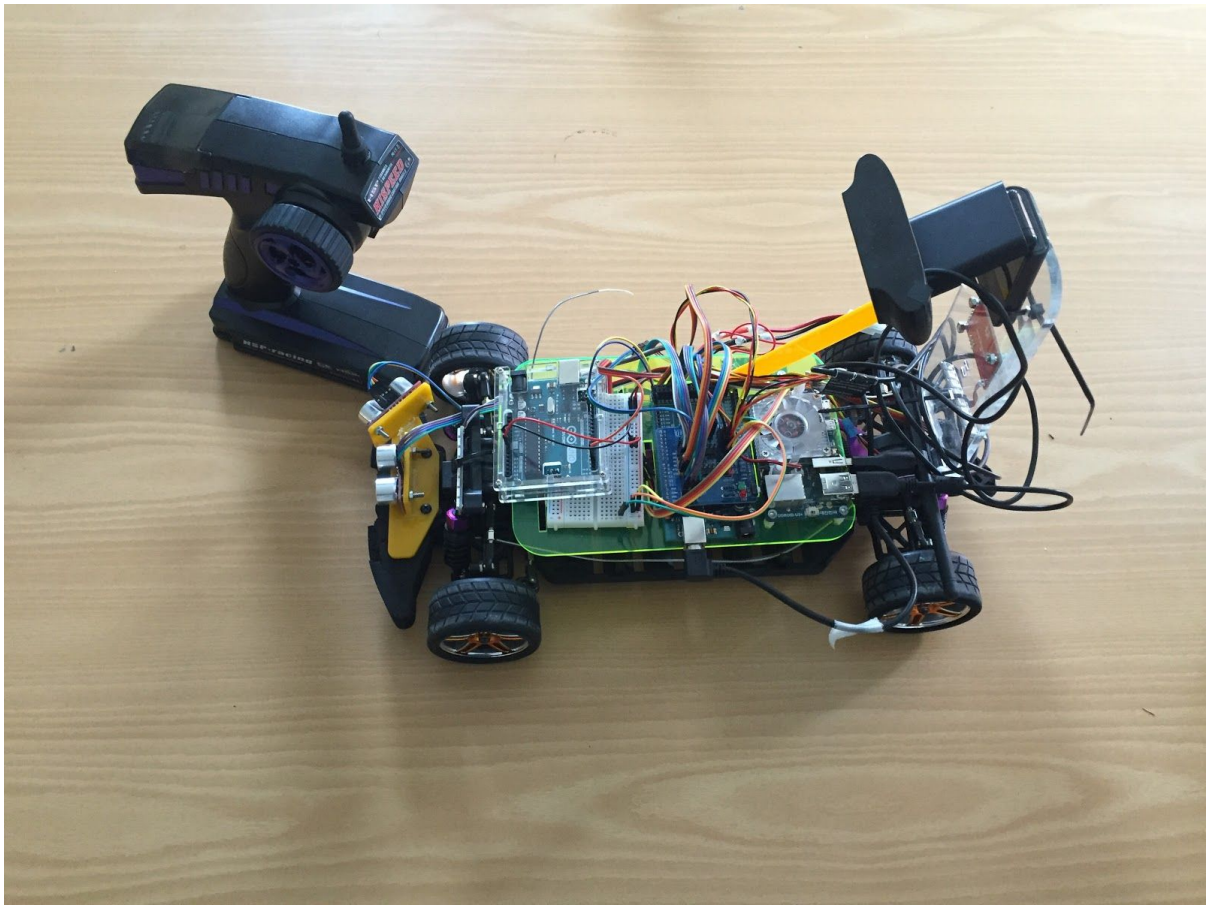
Group 6

Final Report



Oscar Bergström, Sri Hari Thuccani, Viktor Lantz, Zoë Sanderson-Wall, Lena Vartanian, Eythor Atli Einarsson & Martin Kabzimalski

Index

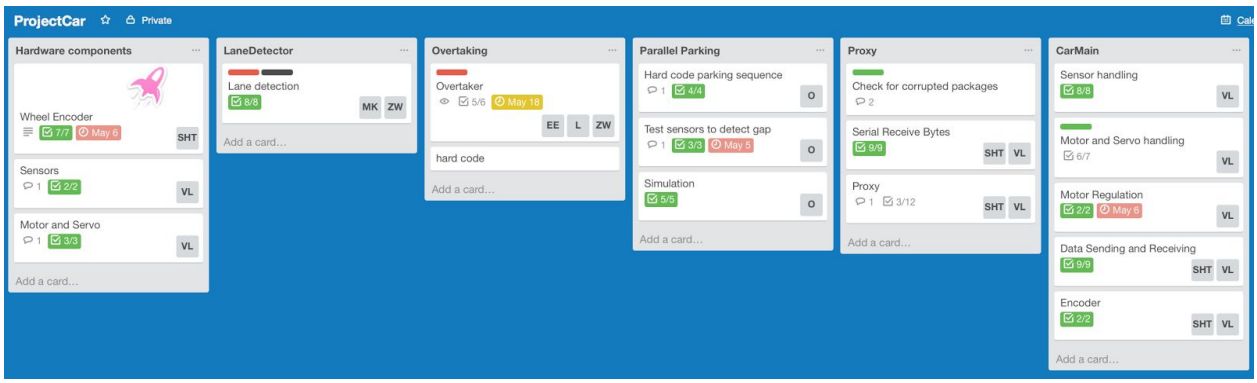
Organization

Eyþór Atli Einarsson

At the start of the project the main focus of the group was on other courses that were running parallel to the project course. Therefore the progress was little as none in the beginning. After the other courses were done we made plans to put all our focus on the project. We set up a plan to use Scrum as our work process. We set up PivotalTracker that should help us with the structure of our work. Unfortunately that did not go as planned and the PivotalTracker never was put to use.

We did divide the work amongst group members and people focused on their tasks and had their eye on the upcoming milestones given by the teacher. We soon figured out that it was not working so well so we made another attempt to use a planning tool and set up a Trello account.

Here is a screenshot of our Trello page:



Since the final deadline was approaching quite fast we had to get work done efficiently and therefore kept our sprints short. Our focus was still at the milestones set by the teacher where we had to present our work every week.

We did fall a little bit behind when it came to presentations. We did not have everything that was asked for in all of them but did manage to get up to speed and get the things done needed for the final demonstration.

Hardware Specification

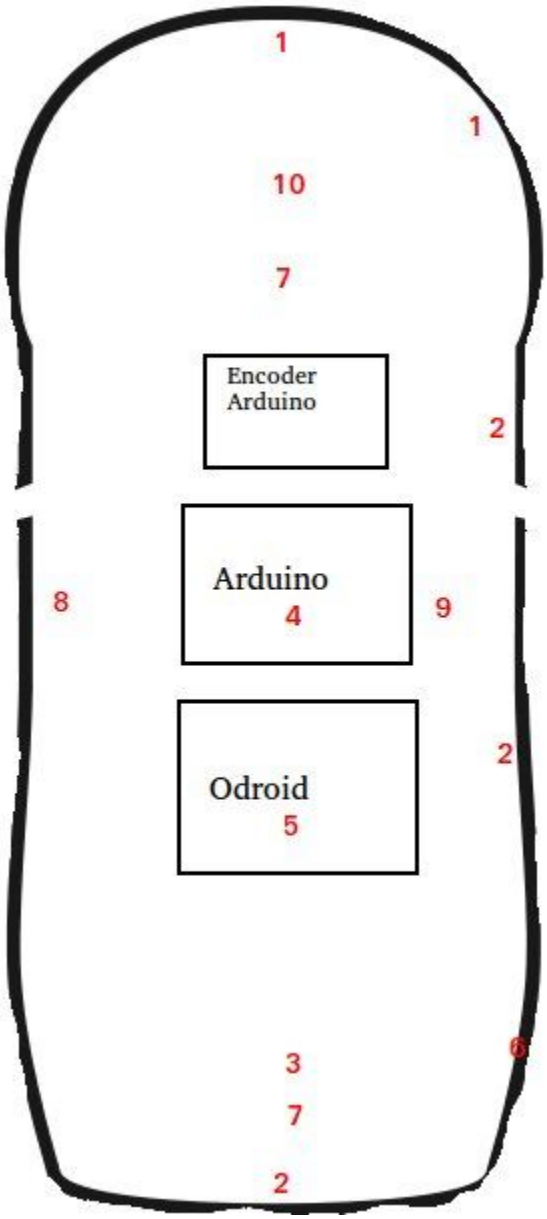
Viktor Lantz

List of hardware used:

(3) Sharp GP2Y0A41SK0F - The infrared sensors which were supplied, they are able to detect objects at a distance of 4 to 30 centimeters.
(2) SRF08 US Rangefinder - The ultrasonic sensors, they have an opening span of 90 degrees. The middle position starts at 0 degrees and read data 45 degrees in each direction. The range of viewing distance is between 3cm to 6m, however this has been reduced to suit our purposes.
(1) Encoder for Pololu Wheel 42x19mm - This object is used to detect the velocity and distance travelled by the car. The encoder is designed specifically for another set of wheels and due to this we were forced to augment the allow the encoder to function.
(1) Lipo Battery Low Voltage Larmsignal - Small LED display which attaches to the battery and allows for reading of the remaining voltage in the battery.
(1) Cheetah 1/10 60A Sensored Combo - The Electronic speed controller (ESC) It instructs the motor at what speed and which direction (back or front) it should drive.
(1) LM2596 spänningsregulator - The voltage regulator, this limits the voltage which is sent to the odroid board to 5volts in order to prevent overloading damage.
(1) Arduino micro (Leonardo) - We added an additional arduino board to the car in order to be able to carry out additional functionality without latency issues.
(1) Arduino Mega 2560 - The low level board which handles output data from the sensors, servo and the motor.
(1) Odroid U3+ - The high level board which is attached to the car, it runs a linux operating system with the openDaVinci framework installed.
(1) Logitech Webcam C930e: a camera overlooking the road.
(1) 9 Degrees of Freedom - Razor IMU - Has not been implemented, it is a gyrometer, accelerometer and magnetometer.

Hardware Architecture

Viktor Lantz

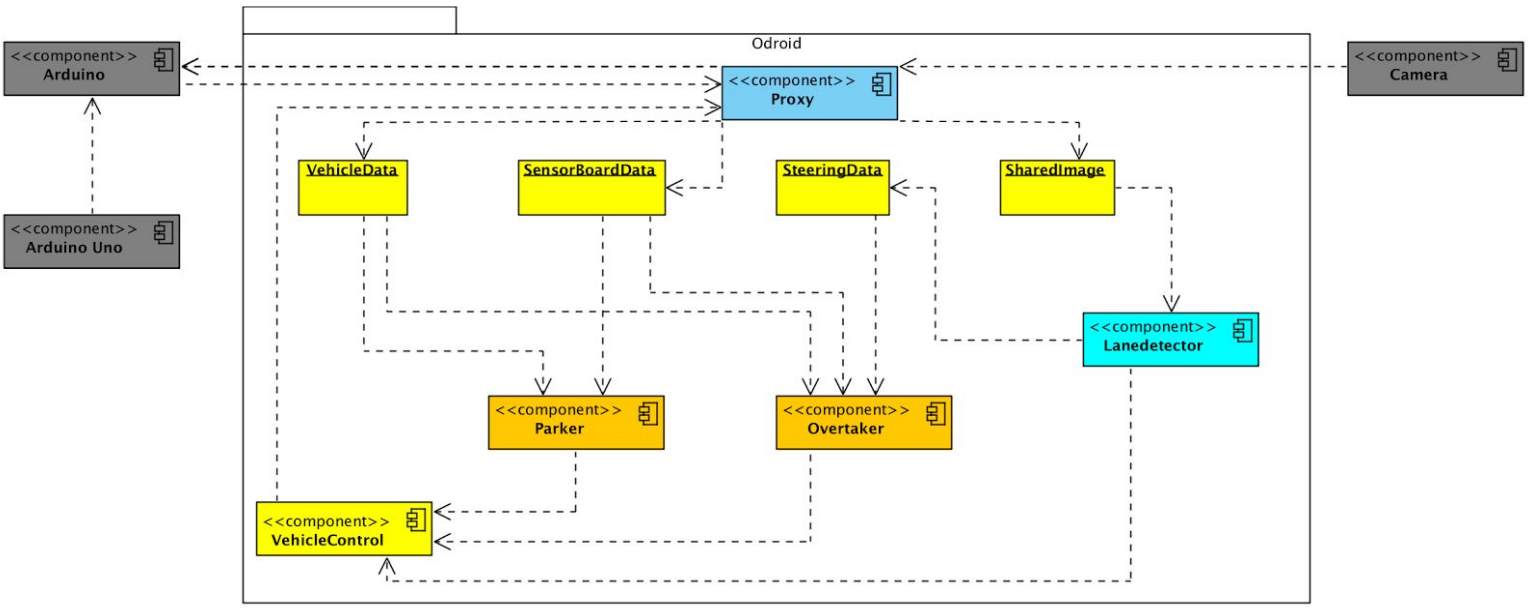


1. Front Center and Front Right ultrasonic sensors
2. Front Right, Back Right and Back infrared sensors
3. Webcam, including the stand it is mounted on
4. Arduino Mega 2560
5. Odroid U3+
6. Wheel encoder
7. Front and Back LED lights
8. Battery
9. Electronic Speed Controller

Architecture

Eythor & Martin

In this part of our documentation we are going to explain the architecture of our system. The component diagram will help us have a clear picture of how every component is connected and how our system is structured.



The Mega Arduino board gathers information from all of the hardware components and send it to the proxy. The gathered data by the proxy is then taken to the required type of data (depending on what scenario we are running). For example, if we initialize our Parker it will listen for sensor and vehicle information (velocity of the car) which will help it go through the different stages of the code (in the Parker). When it enters these stages it will send the output data in the form of packages, which go to VehicleControl and then everything is sent back to the proxy. That way it knows when to tell the Arduino board to steer, go forward, go backwards or stop. The only difference with the other scenarios is that the Overtaker gathers VehicleData, SensorBoardData and SteeringData, which is received from the LaneDetector, for the reason that they work together. Also LaneDetector listens only for the SharedImage information (provided by the camera).

Hardware and Software Integration Progression

Viktor Lantz

Sprint one

Work on the hardware begun prior to the miniature vehicle safety check, on March 15th. This included the implementation of the Electronic Speed Controller (ESC), the servo and the Remote Control (RC) receiver.
The requirements to pass the vehicle safety check was for the car to be able to drive forward and backwards using input commands in the serial monitor of the Arduino IDE. Furthermore, the servo should be controlled in the same way. Finally, the system should have 'Panic Stop' functionality which was implemented using an interrupt. The interrupt function was activated when the RC was switched on, which resulted in a flag variable being set which in turn set the vitals of the car to neutral.

```
// Attach the interupt to pin 3 and run function stopALL if the value rises
// Value will rise if the controller is switched on.
attachInterrupt(digitalPinToInterrupt(3), stopAll, RISING);
```

There were some issues encountered immediately concerning the servo which turned out to be incorrectly calibrated. This meant that the neutral value was significantly lower than it was supposed to be, moreover, the values for steering right and left were reversed.
As the initial sprint, this was a rather large challenge since all aspects of the hardware was new to us. Spending a large amount of time to correctly set up this part was of huge benefit later during the project since the fundamental code structure and functionality has remained largely untouched.

Sprint two

After the vehicle safety check was carried out we had a few weeks which did not require any direct interaction with the hardware. This period was focused on revisiting the previously created software architecture and conceptual idea of the system. However, some time was spent researching and contemplating how the sensors would be connected to the Low Level Arduino board (LLB).

Sprint three

The big picture aim for this sprint was to be able to drive the car inside of the simulation using a lane detection algorithm. The task that was undertook from a hardware perspective was therefore concerning the webcam. The webcam was deemed to be of the highest priority since when the lane following worked perfectly inside of the simulation environment, the camera image would be necessary to begin testing the algorithms in a real life scenario. Additionally, the connection through the proxy to the Open DaVinci framework was undertaken. This was initially rather

problematic due to issues which had previously arisen concerning the setup of the Open DaVinci software. The majority of the sprint was therefore dedicated to creating this connection and was eventually solved.

```
// IR sensor values. Low == close.
IRFrontRight = analogRead(IRSensorFrontRight);
convertedIRFrontRight = IRConversion(IRFrontRight);
IRBackRight = analogRead(IRSensorBackRight);
convertedIRBackRight = IRConversion(IRBackRight);
IRBack = analogRead(IRSensorBack);
convertedIRBack = IRConversion(IRBack);
```

Additionally, when the webcam had been connected, the infrared sensors were connected to the LLB. The values which were received from the sensors were tested to ensure that there was no hardware malfunction issues. Additionally, we added a conversion function which changed the outputted values from voltage to centimeters.

Sprint four

The overall goal for this sprint was to integrate the lane following from simulation with the real life car. In order to do this, the first order of business during this sprint was to get the ultrasonic and the infrared sensors working correctly. The infrared sensors were easily implemented and did not require any specific handling to read data from their respective analog pins. On the other hand, the ultrasonic sensors were a little more complicated and required more setup which was time consuming. We also discovered that the front US sensor was not correctly soldered which resulted in it losing connection and reporting error values.

```
// Main register - The USsensor address.
#define MAIN_08_ADDRESS (0xE0 >> 1) //E0
#define MAIN_08_ADDRESS2 (0xE6 >> 1) //E6
#define GAIN_REGISTER 0x09
#define LOCATION_REGISTER 0x8C
```

```
// USSensors connection and register address location
USFront.connect(MAIN_08_ADDRESS, GAIN_REGISTER, LOCATION_REGISTER);
USFrontRight.connect(MAIN_08_ADDRESS2, GAIN_REGISTER, LOCATION_REGISTER);
```

There were several issues encountered during this sprint which acted as bottlenecks for other aspects of the project. One of these major issues was the fact that the LLB was not being supplied with a sufficient amount of voltage from the battery. This meant that when we connected the ultrasonic and the infrared sensors at the same time as the servo and attempted to drive the car, the process would hang itself and the car would become unresponsive. This was a concern since we were unable to drive the car with all necessary components connected.
This issue was solved by adding a voltage regulator to the arduino board providing it with additional juice.

Sprint five

As the majority of the necessary components had been added to the car with the relevant functionality inside of the INO file, we began handling the data which needed to be sent from the LLB to the Odroid High Level Board (HLB). The boards are connected through a serial connection. To send the sensory data from the LLB,

we simply needed to format the data into a string which contained the appropriate delimiters and print it to the serial monitor.

```
outputString = us + frontSensorReading + ',' + frontRightSensorReading + ']' + ir +
convertedIRFrontRight + ',' + convertedIRBack + ',' + convertedIRBackRight + ']';
```

Writing the proxy was of big help when trying to understand how the data should be sent and received.

In order to receive the data, we created a serialEvent, which is activated once for every execution of the loop. The while loop inside of the serialEvent is only activated if there is available data on the serial. If there is, the data is decoded and the relevant data values are extracted and assigned to global variables.

```
void serialEvent() {
  while (Serial.available()) {
    // get the new byte:
    int firstDelim;
    int lastDelim;
    int middleDelim;
    String speedStr, steeringStr;


    char inChar = (char)Serial.read();
    // add it to the inputString:
    inputString += inChar;
    // so the main loop can do something about it:
    while (inputString.startsWith("{") && inputString.endsWith("}")) {

      firstDelim = inputString.indexOf("{");
      middleDelim = inputString.indexOf(",");
      lastDelim = inputString.indexOf("}");
      speedStr = inputString.substring(firstDelim + 1, middleDelim);
      steeringStr = inputString.substring(middleDelim + 1, lastDelim);
      inputString = "";
    }
    speedVal = (double) speedStr.toInt();
    steeringVal = (double) steeringStr.toInt();
  }
}
```

Sprint six

As the next sprint begun, development of the parking and overtaking scenarios began to progress to a point where they could be tested on the real life car. At this point we discovered that the wheels were slightly touching the axis and this resulted in inconsistencies concerning the speed when turning. We screwed off two of the wheel and added some additional padding which would prevent this unnecessary friction from occurring. Simultaneously we had issues with some unintended acceleration when using the RC, this turned out to be because of fluctuations in the pulse values. This was solved by increasing the parameters at where the car would be considered to be in a ‘neutral state’.

Sprint seven

Sri Hari Thuccani

In this sprint the wheel encoder was implemented into the arduino system. The wheel encoder is a hardware device that is used the measure the velocity of a moving wheel.

To do this the wheel encoder detects black and white colours through two infrared sensors. The two colours are represented as tapes which are attached to the inside of a wheel. As the wheel rotates the IR sensors detect the colours passing by and translates them as ticks. These ticks are presented as binary code as 1 & 0.

On the arduino side an algorithmic code calculates the ticks into rpm then to velocity by time taken of 1000 milliseconds (1 second). The formula used is :

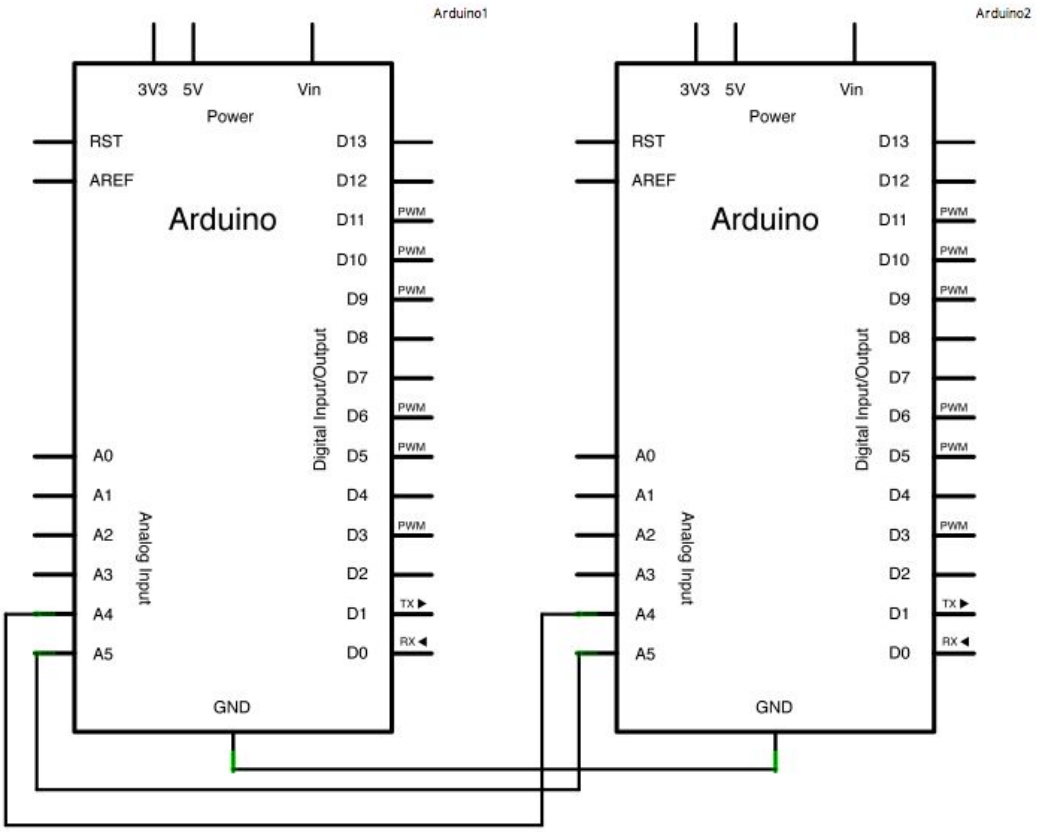
tickCount = number of ticks per second

$$rpm = \frac{tickCount}{wheel\ ticks} \times time\ taken = \frac{tickCount}{4} \times 60$$

$$velocity\ = rpm \times \frac{0.32 \times \pi}{60}$$

Note: tickCount is divided by 4 because the wheel has 4 black tape strips.

Due to the latency issues that the arduino mega had (may be caused by the car's detection sensors), running a millisecond counter did not function as it is intended. Rather than counting up in a consistent manner, it skipped numbers by a huge margin never allowing the calculation code to be executed at 1000ms. To solve this problem an I2C protocol was used to handle the wheel encoder. This is used and imagined as a multithreaded component.



Arduino I2C protocol setup. [https://www.arduino.cc/en/Tutorial/MasterReader]

An I2C protocol requires two or more arduinos connected to each other by two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL). These two lines are used to send and receive data between the two arduinos. SCL pulses from low to high sending a single bit of data that will form in sequence the address of a specific device. Then a command or data is transferred from the master board to the slave device through the SDA line. After the command has been transferred the called upon device executes the request and transmits the data back to the master board over the same line SCL.

To accomplish this a *Wire library* is used to set the functions for communication between the two arduinos.
The main functions used for the communication are :

Slave Sender

Wire.begin() establishes the address (8) for the slave sender and Wire.onRequest(requestEvent) executes requestEvent() method when a command is received from the master reader.

```
Wire.begin(8);
Wire.onRequest(requestEvent);
```

```
This method sends the velocity in bytes back to the master reader
void requestEvent() {
  Wire.write(velocity);
}
```

Master Reader

```
Request the number bytes (1) sent from the specified slave sender by determining the address (8).
Wire.requestFrom(8, 1);

Wire.read() reads the bytes transmitted by the slave sender.
if (Wire.available()) {
  velocity = Wire.read();
}
```

velocity is then assigned to an integer named velocity, which is used to regulate the speed. This ensures that the car doesn't have a constant boost speed.

In the beginning there were issues of the hardware component of the wheel encoder. The issue was that the IR sensor for detecting the colour white was not functioning. However it wasn't a detrimental issue since calculating the rpm requires only one sensor to work.

As mentioned in this wheel encoder content, there were issues regarding to the execution of calculating the velocity due to the latency of the arduino mega. It is believed that the cause of this issue was made by the onboard sensors of the car, specifically the ultrasonic sensors. To solve the issue another arduino was included to act as a multithreaded component to run the wheel encoder functions without hindrance of other components.

Another issue was transmitting more than 1 data from the slave sender to the master reader. The issue was sending an incrementing distance variable so that the overtaking and parking can know the distance traveled from takeoff. This did not work since the I2C protocol can only transmit a maximum of 255 bytes per data. To resolve this issue incremental distance was removed and instead a calculation of distance traveled per Wire request was implemented.

$$circumference = 21cm$$

$$\frac{circumference}{wheel\ ticks} = \frac{21}{4} = 5.12cm$$

$$distance = tickCount \times 5.12$$

The variable distance is then reset to 0 after the the data has been sent to master reader. However this solution created another issue of where the master reader was receiving the value 0 instead of distance values. Due to time constraints the issue was never resolved and instead the parking and overtaking functioned without the distance value.

Sprint eight

Viktor Lantz

The last sprint consisted of mainly small alterations to steering angles and speed values which would be passed to the servo and the ESC respectively. This was due to the fact that the power of the engine greatly exceeded the speed at which we required the car to move. Therefore we decided that a good solution to this would be to implement the wheel encoder in a way that created a ‘gear’ system. If the car was moving to slow then the car would receive a boost and then run at a low speed until the velocity once again indicated that the car needed a boost.

Simultaneously we encountered some last minute issues concerning the values which were being received for the servo. Occasionally the serial would receive zero values which would set the wheels into the neutral position. We eliminated this problem by switching the value which sets the neutral wheel position to five, hence the error values are ignored.

Lastly, we had an unfortunate accident where the LED wire which allowed the turning and brake lights to function tore. We decided to not prioritize this issue and instead focus on functionality which was more vital to the fundamental operations of the car.

Proxy

Viktor Lantz

The proxy was an extremely challenging part of the project. We realised that the proxy may also become a severe bottleneck in the system since it functions as a communication link between the car and the high level logic code. Therefore we decided to tackle it as early as possible so that we would quickly have a functioning version.

We began by instantiating a connection to the LLB through the proxy. This was done using the library SerialPortFactory and the function within this called createSerialPort, passing the correct parameters. We ran into an issue where the defined serial port would change if the serial-usb cable is disconnected from the LLB. This makes the defined serial port changes from e.g 'ACM1' to 'ACM2'. The result is that the proxy needs to be changed and re-made which is a little bit of a hassle, however, it did not affect performance and we did therefore not prioritise finding a solution.

```
//Define Serial port connection for Arduino
const string SERIAL_PORT = "/dev/ttyACM1";
//Define BAUD_Rate for serial connection
const uint32_t BAUD_RATE = 9600;
// Create connection to the serail port
std::shared_ptr <SerialPort> serial(SerialPortFactory::createSerialPort(SERIAL_PORT, BAUD_RATE));
// Link connection to the handler
serial->setStringListener(&receiveHandler);
// Start listening for the data on the serial port.
serial->start();
```

A string listener is attached to the newly created serial port which listens for incoming data. This is created in the body of the proxy, but not inside of the loop since we do not want to create a new connection for each run of the loop. When the loop ends, the serial connection is stopped and the string listener which was previously attached to the port is removed (set to NULL).

```
serial->stop();
serial->setStringListener(NULL);
```

The main loop inside of the body of the proxy carries out two main tasks.

1. Receive data from the SerialReceiveBytes class, insert the data into a map and send the data to the HLB. (LLB to HLB)
2. Receive data from the HLB, encoding the data and sending it to the LLB. (HLB to LLB)

LLB to HLB data transfer

The proxy receives two vectors from the SerialReceiveBytes class which contain the ultrasonic sensor values and the infrared sensor values respectively.

```
vector <int> usVector = receiveHandler.getUSVector();
vector <int> irVector = receiveHandler.getIRVector();
```

We use a map to store an unsigned integer and the values of the vectors in order to be able to pack the data into the SensorBoardData object. The sensorBoard is a library from openDaVinci which is the required way to pack the data in order for it to be read correctly in the ODV framework.

When packing the values into the map we do some checking of the sensory values. As each value is inserted into the map, if it is larger than 50 for the ultrasonic sensor, the value that is stored is set to -1. This was done to assist the overtaking. Similarly with the infrared values, if they read above 60, the value which is stored is set to -1.

```
for(int i = 0; i < 2; i++) {
  if(usVector[i] > 50){
        values[i+3] = -1;
    }else{
        values[i+3] = usVector[i];
    }
}
for(int i = 0; i < 3; i++) {
   if(irVector[i] > 60){
        values[i] = -1;
    }else{
        values[i] = irVector[i];
    }
}
```

As the mapping is complete, the values are inserted into the sensorBoardData object which in turn is packed into a container object. These objects are constructed from the relevant openDaVinci libraries. Finally, we use the distribute function, which takes a container as an argument to send the packed data to the test cases which are requesting the data.

```
SensorBoardData sensorBoard(numOfSensors, values);

Container sensorContainer(sensorBoard);
distribute(sensorContainer);
```

This section of the proxy was not especially problematic, the largest issues were encountered in the SerialReceiveBytes class where all the decoding of packages occurred.

HLB to LLB data transfer

In the proxy we receive two values which are of importance to the operational functionality of the car. These are the speed values and steering wheel angle.

In order to be able to access the speed and steering values we are obliged to use the same method which was done to pack the data sensor board data.

```
Container containerControlData = getKeyValueDataStore().get(automotive::VehicleControl::ID());
VehicleControl vc = containerControlData.getData <VehicleControl> ();
```

We create an object of vehicle control which allows us to use the functions:

```
double speedVal = vc.getSpeed();
double steeringVal = vc.getSteeringWheelAngle();
```

The values which are received from the test cases are doubles, however when we send the data to the LLB, they need to be strings. Therefore we convert the double values into stringStreams and then into regular strings using the string library function .str.

```
stringstream steering_str;
stringstream speed_str;

speed_str << speedVal;
steering_str << steeringVal;

string speedString = speed_str.str();
string steeringString = steering_str.str();
```

Finally, we concatenate the two strings into one whilst also inserting the correct delimiters to denominate the start, middle and end of each package. The package is then sent using the serial library function send.

```
if (sendCounter > 200) {
    serial->send("{" + speedString + "," + steeringString + "}");
}
```

One issue that we encountered which slowed down the development of the proxy dramatically was the function ‘serial -> send’. What we believed to be the issue was that the serial -> send was being carried out before the setup of the serial port was allowed time to finalize. This would result in the proxy crashing without giving us any indication as to why. We spent a considerable amount of time attempting to fix this and we eventually managed to with the assistance of a student supervisor. The solution was to add a counter at the beginning of the proxy loop which waits for 200 runs of the loop to be carried out before Serial-send is called. This means that there is a small wait until data gets sent to the arduino board but it is an insignificant time delay.

SerialReceiveBytes.cpp

Sri Hari Thuccani

The example for SerialReceiveBytes was used to test and understand the code of the receiving bytes from the serial port. It was later modified so that the proxy establishes the connection to the serial port leaving the SerialReceiveBytes.cpp to handle the algorithm for the deconstruction of the packages as a separate thread.

Below is a basic step by step process of how the data received from the serial port gets handled:

1. Receive data package from assigned serial port.
2. Adds the data to a buffer container which stores it until a full package has been received.
3. Evaluates the content of the package during the deconstruction.
4. The data within the package gets pushed to their respective identifier, *u* or *i*, array (the identifiers were determined during evaluation).
5. The array gets sent to the main body of the proxy to be further stored in a container. The container itself can be accessed from the different test cases, them being parker, overtaking and lane detector.

substr() is a function that is capable of creating a substring of the parent string by stating the parameters of its starting and finishing point. This was used in several occasions for the process of deconstructing the package that gets received.

Vectors are containers that represent arrays but are capable of changing its size. They have a higher capacity than normal arrays however they are process intensive. Vectors seem the most appropriate approach of storing the data since the amount of data can vary depending on which sensor the data belongs to.

The packages that get received from the serial port comes in the form:

[u20,34] & [i15,52,32]

Each package has a start delimiter and an end delimiter. Using the first package as an example to be processed through the deconstruction code, a substring gets created starting from *u* and ends at *4*, right before the end delimiter.

```
if (buffer_container.at(0) == '[') {
  std::string sub_str = buffer_container.substr(1,(sq_bracket-1));
  buffer_container.erase(0,sq_bracket+1);
```

Using *u* as an identifier the substring gets sent to a method for further deconstruction.

```
if (sub_str.at(0) == 'u') {
  sub_str.erase(0,1);
  us_vec = vector_compiler(sub_str);
}
else if (sub_str.at(0) == 'i') {
  sub_str.erase(0,1);
  ir_vec = vector_compiler(sub_str);
}
```

The substring go through the same process, this time starting from *2* and ending at *0*, right before the comma. However there is one more substring created starting after the comma at *3* and ending at *4*.

After successfully extracting the values from the string they get pushed into a vector variable which gets returned to vector <int> us_vec().

```
vector <int> vector_compiler(string str) {
  vector <int> vec;
  std::string comma(",");
  std::size_t comma_val = cont_str.find_first_of(comma);

  // Separates the values in the package (str), recursively if needed.
  if (cont_str.find(comma) != string::npos) {
    std::string val_str = str.substr(0,comma_val);
    std::string rest_str = str.substr(comma_val+1);
    std::stringstream vS(val_str);
    int value;
    vS >> value;
    vec.push_back(value);
    vector <int> rest_vec;
    rest_vec = vector_compiler(rest_str);
    vec.insert(vec.end(), rest_vec.begin(), rest_vec.end());
  }

  // Converts the values from string to integers.
  std::stringstream cS(str);
  int final_value;
  cS >> final_value;
  vec.push_back(final_value);
    return vec;
}
```

If a corrupted package gets received it gets caught by another function, which removes everything within the *buffer_container* up until the next start delimiter.

Example of a corrupted package:

0,34] or ,52,32]

Reflection

Initial problems with the SerialReceiveBytes.cpp (SRB) were compilation errors, due to the proxy and the SRB had their own mains. This issue was solved by moving the code contained in the main of SRB to the proxy. Leaving the SRB to handle the data that gets received.

Since the method in SRB runs on recursion it made it a lot easier to complete the unpacking without a long list of *if* functions. This kept the code clean and efficient, however a better approach would have been an algorithm based on a single *for* loop with its loops dependant on the the amount of commas in the package. The only concern would be the amount of latency caused by the for loop.

Overall the proxy ran successfully and was completed on time for running the different test cases.

Lane detection

Martin Kabzimalski

(We used the skeleton LaneDetector.cpp for this project which was provided by Christian Berger)

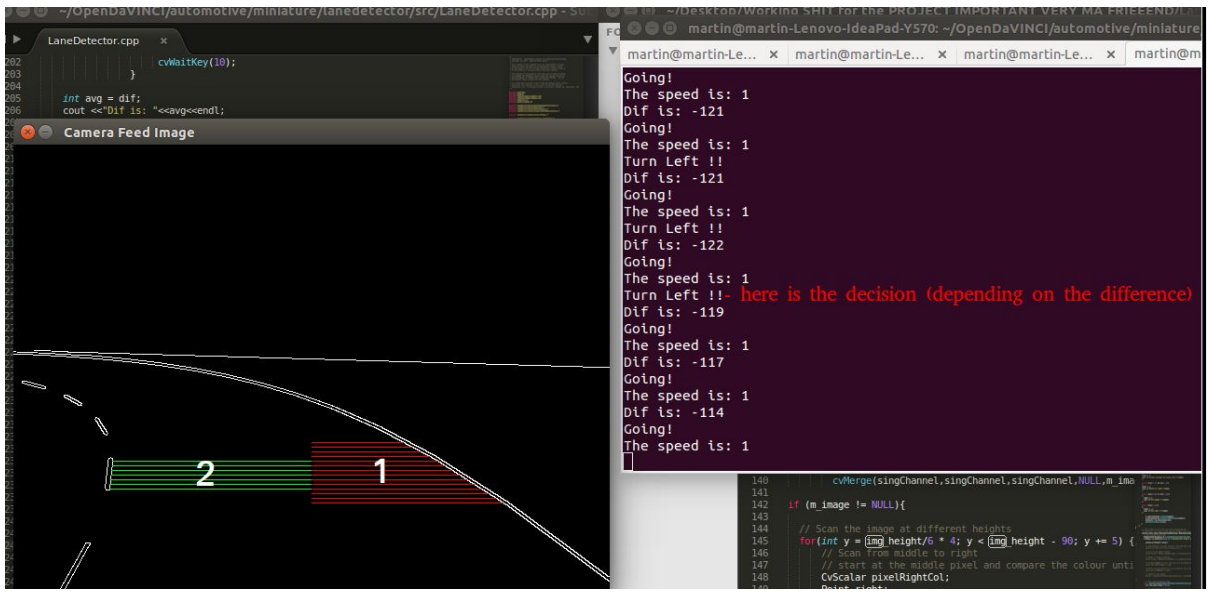
In this section we will describe the lane detection algorithms that are used in our car for determining the position and direction of it on the track.

We use three algorithms in total - the first two are for detecting white pixels: one detects the road from both sides while calculating the difference between the right half and the left half of the road (screen) and the second one is used for listening for the intersection line (if it should stop at an intersection or not).

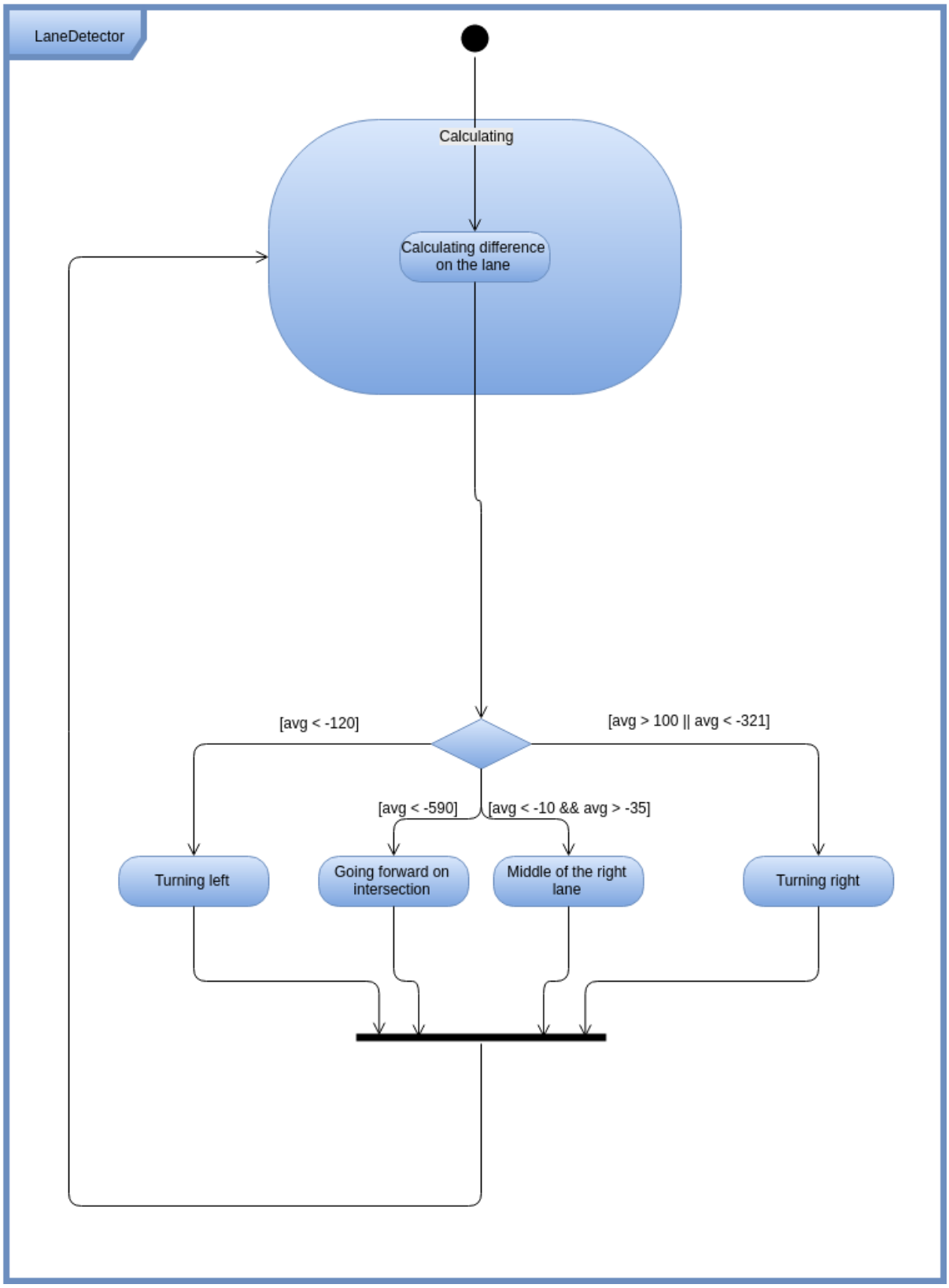
The third algorithm is the one that controls the steering and the speed of the car (whether if it should go forward or stop).

Lane detection with difference

After a long time of brainstorming we decided that the most suitable way of controlling the car is by calculating the difference (distance) between the right line of



the lane and the dotted line on the middle of the road. If we look at the picture we see two numbers that are used for describing the area of calculations: 1 - The car measures the distance from the middle of the screen to the right white line, 2 - this part does the same action but from the middle of the screen to the left (when it detects the dotted white line). When the measurements are done from both sides a subtraction between them follows, thus giving us the exact difference between the two lines, which leads to a better understanding of the direction of the car.



This state machine diagram describes the different states the code goes through: With the initialization of LaneDetector.cpp it begins with calculating the difference between the right lane lines (when they are detected) and depending on that

difference it controls the car. The variable that we use for storing this value is “avg” (presented in the diagram).

[avg < -120] - This means that the car is entering the right side of the right lane so we have to turn left if we want to drive in the middle.

[avg > 100 || avg < -321] - This means that the car is entering the left side of the right lane so we have to turn right if we want to keep the car in the middle.

[avg < -10 && avg > -35] - This means that the car has reached the desired middle area of the road, in order for it to be capable to turn without going out of the track. It is also used for keeping the car from turning for too long.

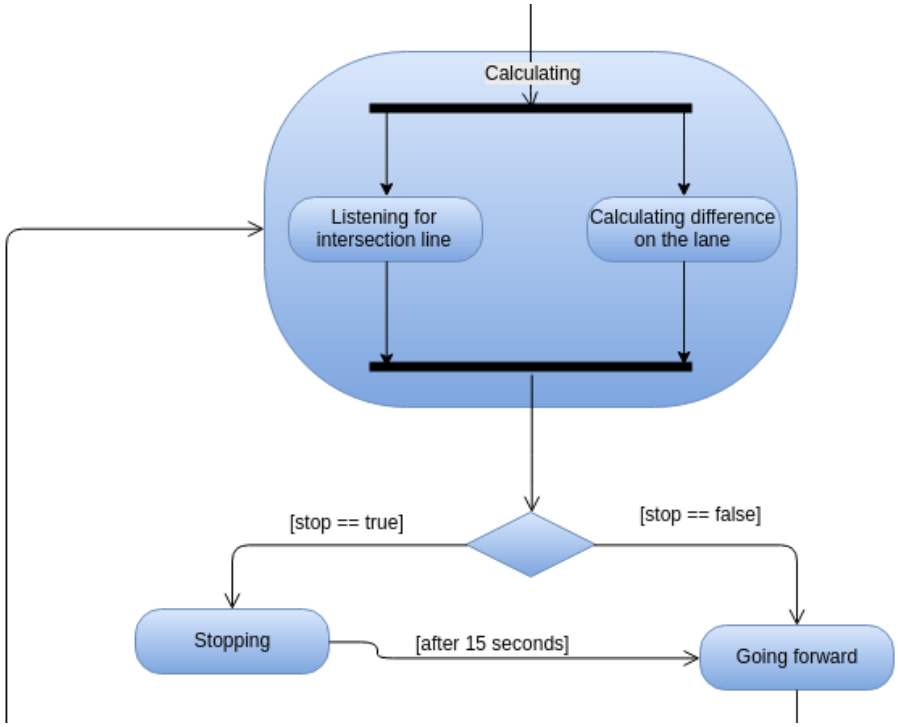
[avg < -590] - This means that there are no lines present, in other words, we are passing through an intersection (driving forward, no turning).

When none of those conditions are present it goes back to calculating the difference.

Stopping at the intersection

What we used to stop the car at the intersection was another algorithm that detects white pixels in an area where the intersection line is expected to be seen by the camera. The way we made the vehicle stop is by counting how many times a white pixel was detected (how thick the line is) at the designated area. When the intersection line is detected it sets the speed so that the car stops instantly.

This algorithm was not used in the presentation, for the reason that there was not enough time for testing, in other words, to make it work correctly with the rest of our algorithms. We were dealing with bigger issues at that moment so we decided that this feature should work only in a simulated environment.



This is how the state machine diagram would have looked if we had more time for testing. With the initialization of LaneDetector.cpp it will start listening for the intersection line and calculating the difference simultaneously. If the intersection is detected it will set a boolean to true, which will stop the car. After 15 seconds it will make the car go forward again. If nothing is present in front of the car it will keep going forward until the desired object is detected by the camera. The rest of the diagram is the same as the first one presented more above.

Image Processing Implementation

Zoe Sanderson-Wall

A number of image processing algorithms were implemented from the open source software library, OpenCV. The original image from the camera is received as the OpenCV data type, IplImage which is then processed in four stages. A smoothing algorithm is used in conjunction with the Gaussian blur implemented by Canny Edge Detection as this alone was not sufficient in removing noise.

cvCvtColor()

The first stage involves converting the image to a single channel grayscale image using the OpenCV function cvtColor(). This function transforms the input array image to grayscale using a transformation algorithm(citation).

cvSmooth()

The OpenCV image smoothing algorithm is employed with a Gaussian type smooth using a 3x3 matrix. This removes unwanted noise from the camera image by allocating a new value to each pixel based on the weighted average of the Gaussian matrix around that pixel.

cvCanny()

The most integral stage involves the OpenCV edge detection function which implements the algorithm developed by John F. Canny. This function locates the edges of the image by analysing the gradients of the pixels and compares these to the threshold arguments. If a gradient value is between the threshold values it is marked as a weak edge pixel, and a value that is higher than the high threshold is considered a strong edge pixel. Any pixels with a gradient lower than the lower threshold are discarded. This process further assists in noise reduction. The final image is comprised only of the identified strong edge pixels and any weak edge pixels that are adjacent to these. The threshold values were determined through a process of trial and error. The derived values deviated slightly from the recommended 2:1 ratio, however the results are satisfactory.

cvMerge()

The final stage involves merging the processed image in order to return to an image format with three channels. This allows for the use of RGB colours.

Lane Detection Implementation

Zoe Sanderson-Wall

The fundamental aspects of the lane detection code are scanning for the lane markers, determining the location of the car with respect to the centre of the image and changing the angle of the steering as a result of this location, and detecting an intersection.

Scanning for Lane Markers

```
// Scan the image at different heights
for(int y = img_height/6 * 4; y < img_height - 90; y += 5) {
    // Scan from middle to right
    // start at the middle pixel and compare the colour
    CvScalar pixelRightCol;
    Point right;
    right.x = -1;
    right.y = y;
    for(int x = img_width/2; x < img_width; x++) {
        pixelRightCol = cvGet2D(m_image, y, x);
        if(pixelRightCol.val[0] >= 200) {
            right.x = x;
            break;
        }
    }

    // Scan from middle to left
    CvScalar pixelLeftCol;
    Point left;
    left.x = -1;
    left.y = y;
    for (int x = img_width/2; x > 0; x--) {
      pixelLeftCol = cvGet2D(m_image, y, x);
      if(pixelLeftCol.val[0] >= 200) {
        left.x = x;
        break;
      }
    }
```

The scanning process involves analysing the pixels along the x-axis from the middle and outwards in both directions at multiple select values on the y-axis. The OpenCV function cvGet2D() retrieves an array of values for the RGB colour at the given point, which is then analysed in order to detect if a white lane marker edge has been found. A white pixel is represented by the array (255, 255, 255) so an edge is determined as found if the first array value is greater than 200.

Determining Location

```
    // If a the lane lines are detected, draw from the middle to that line
    if(left.x > 0) {
      cvLine(m_image, Point(img_width/2, y), left, Scalar(0, 255, 0), 1, 8);

    }
    if(right.x > 0) {
      cvLine(m_image, Point(img_width/2, y), right, Scalar(0, 0, 255), 1, 8);
    }
    // Calculate differences between right and left lines
    dif = (right.x - img_width/2) - (img_width/2 - left.x);
  }
```

The location of the car is determined by calculating the difference between the distances from the middle of the image to the right and left detected lane markers.

Detecting an Intersection

```
//DETETCT THE WHITE LINE BEFORE THE INTERSECTION
for(int yU = img_height/6 * 5; yU < img_height -50; yU += 5) {

    CvScalar pixelRightCol2;
    int countWhite = 0;
    for(int i = img_width/2 - 60; i < img_width/2 + 60; i++) {
      pixelRightCol2 = cvGet2D(m_image, yU, i);
      if(pixelRightCol2.val[0] >= 200) {
        countWhite++;
      }
    }

   if(countWhite > 20 ) {


      stop = true;
  }
```

In order to detect an intersection a segment of the image slightly above the front of the car is scanned for white pixels using the same method as for detecting lane marker edges. If a white pixel is found, the counter increments. An intersection is determined to be found when more than 20 white pixels are detected and the ‘stop’ boolean is then set to true.

Sideways (parallel) Parking

Oscar Bergström

Introduction to Sideways Parking

This section will cover the development and implementation of the algorithm made for the autonomous car to do sideways parking.

Requirements

One of the requirements for the autonomous car is the ability to park itself. The parking should be performed sideways (parallel parking). More specific requirements for the parking was that the parking space should be of maximum 85 cm (assuming that the car is roughly 42 cm). The car should be able to find a big enough gap, indicate that the process of parking is initiated by flashing direction signals with the LEDs and park the car.

Inception

During the inception phase of the project a concept of how the parking should be implemented was developed and a few questions surfaced: How can the car look for a gap? How can the car know the gap is big enough? How is the car going to behave to not crash into obstacles?

The first stage of answering these questions a list of the hardware components was made. What did the hardware do? What kind of constraints did they have? After looking at the hardware the process of making a simulation of the car parking was initiated. The logic concept of how the parking should be executed in the simulation was developed.

Simulation Code for Sideways Parking

1. Initiate the car to a state where it looks for a parking gap to the side. Put the car in a constant velocity forward and use the IR-sensor on the side to look for objects. The IR-sensor detects when there is an object to the side or not. When there is no object a counter starts which measures the traveled distance. As soon as another object to the side is detected the distance counter stops and calculates the distance of the gap. If the gap is big enough (> 85 cm) the car goes on to stage two. If the gap is not big enough, reset the counter and start looking for another gap.
2. Go a certain distance backwards with the steering pointing right. Continue to go backwards a certain distance with the steering pointing left. Put the steering wheels to right and go slightly forward.

This concept was coded, tested and refined in the simulator with good results. After having this done it was time to implement this code for the real life car. As the code for the simulation was built for perfect conditions without noise from sensors, perfect steering angles and constant velocity it was a major difference to the real life scenario.

Problems and Solutions Encountered from Simulation to Real Life Car

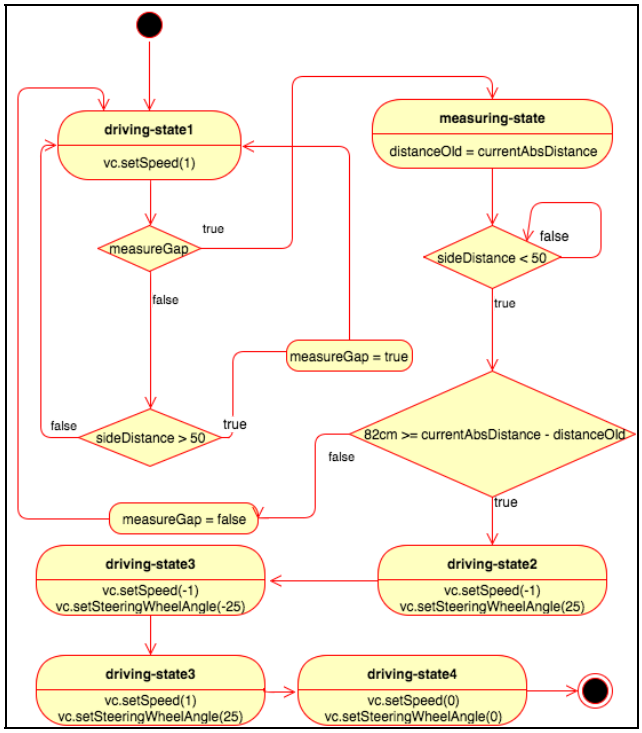
Since the wheel encoder did not work, calculating the distance traveled was virtually impossible which made the gap calculation impossible. After a lot of trial and error a solution was found. When the side sensors didn't detect anything (in gap), a counter was increased. After testing how many times this counter was increased when passing a big enough gap an approximate value was found. When the counter reached this value and both of the side sensors detected a new obstacle to the side the car stopped.

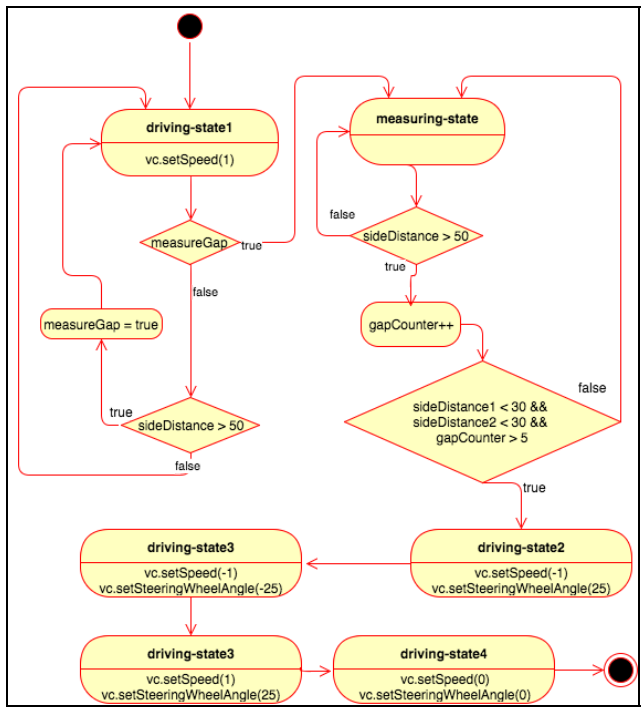
Another problem that was encountered during the same point in time was that the car was going to fast for the lane following. The car didn't have time to calculate and re-adjust the wheels. It went of track and a solution for this was to lower the velocity of the car. This caused another problem. The interchange of the motor was not high enough to make the car go forward from being still with a lower velocity, so two different values for going forward was set. One boost value to get the car going if it was under a certain velocity and one cruising value. The wheel encoder was partially fixed and was able to send an approximate velocity.

Now when the speed was not constant it caused the counter that was made for “measuring” the gap to be different during every run. This in turn had repercussions on the stopping after finding a gap. Sometimes it rolled 10 cm after telling it to stop, sometimes it had such low speed that it stopped right away after passing the gap. To fix this, a stopping value was implemented in the Arduino code to make the car quickly go backwards and then go quickly forwards to make the car go to a full stop. This implementation worked pretty well but the problem of measuring the gap still remained.

Since the speed of the car was inconsistent measuring the gap with a counter was not an option anymore. During discussions with the examinator he allowed us to have a box in front of the parking spot to use the front sensor of the car to “measure” when the parking should be initiated. After some trials we managed to solve it without the box. Instead we used the previous gap-counter to just look for a gap, and stop the car when another obstacle to the side was found.

State Machine Diagrams for Parking





Left diagram: Simulation
Right diagram: Real life Car

Since there were problems with the wheel encoder the state machine diagrams differentiates (for the simulation and real life car) in the logic and decision making. In the code there are more “driving-states”, but for the purpose of displaying the logic and decision making they are redundant.

```
//Measuring state machine.
    switch (stageMeasuring) {
      case 0:
      {
          //Initialize measurement.
          if(parkingSpotFound == false){
            distanceOld = sbd.getValueForKey_MapOfDistances(2);
            stageMeasuring++;
          }
      }
      break;
      case 1:
      {
          if(parkingSpotFound == false){
            if ((distanceOld > 0) && (sbd.getValueForKey_MapOfDistances(2) < 0)) {
                //Found start of gap
              stageMeasuring = 2;
              absPathStart = vd.getAbsTraveledPath();
            }
          }
          distanceOld = sbd.getValueForKey_MapOfDistances(2);
        }
      break;
      case 2:
        {
        if(parkingSpotFound == false){

            if ((distanceOld < 0) && (sbd.getValueForKey_MapOfDistances(2) > 0)) {
                //Found end of gap
                stageMeasuring = 1;
                absPathEnd = vd.getAbsTraveledPath();

                const double GAP_SIZE = (absPathEnd - absPathStart);
                cerr << "\nSize = " <<GAP_SIZE << endl;

                if(GAP_SIZE > 0.1){
                    m_foundGaps.push_back(GAP_SIZE);
                }

                if((stageMoving < 1) && (GAP_SIZE > 3.5)) {
                    //Parking spot found
                    stageMoving = 1;
                    parkingSpotFound = true;
                }
            }
        }
        distanceOld = sbd.getValueForKey_MapOfDistances(2);
        }
      break;
    }
```

Measuring state machine for the simulation

The code for measuring the gap is more complicated than the code for the gap-counter for the real life car. The simulation code is not changed a lot from how the boxparker example was, which was provided by Christian Berger. In the simulation the sensors are giving -1 values when its “not detecting” anything. This means that there are no obstacles closer than 3m.

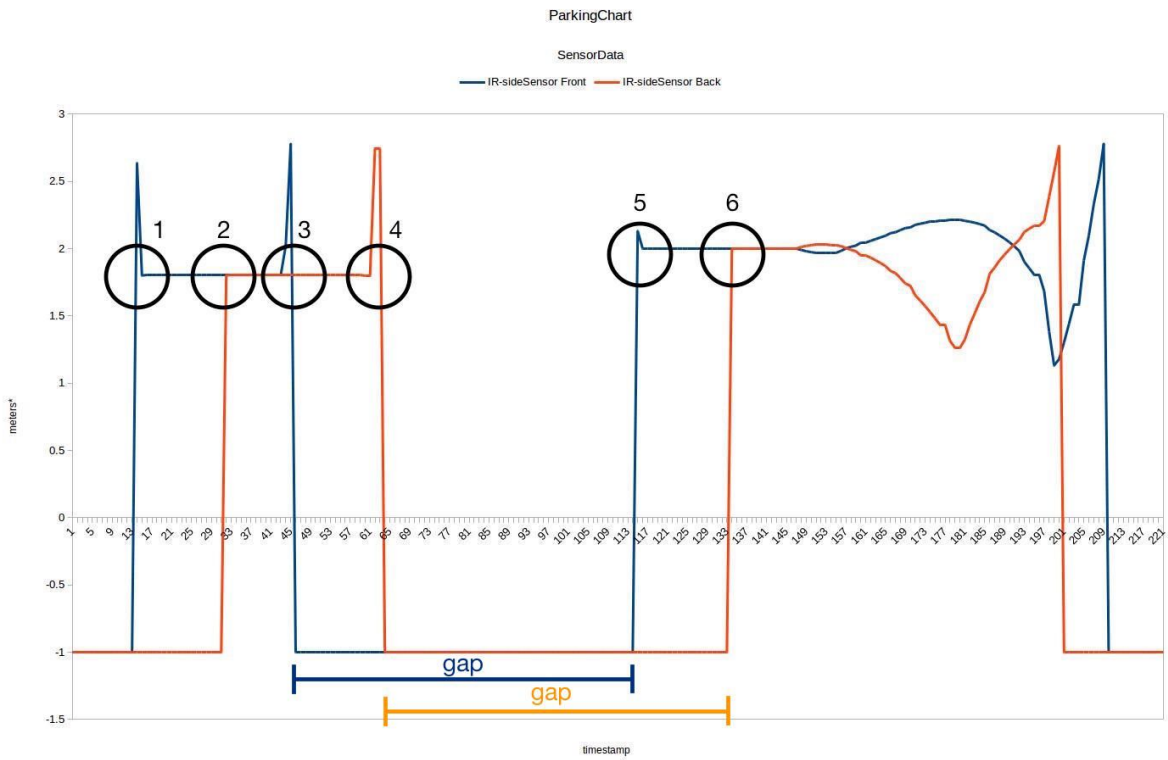
Code for “measuring” the gap on real life car

```
if(parkingSpotFound == false){
    if(sideSensorValueF > 50){      // Adds to counter when the vehicle detects gap on the side
        sideSensorGapCounterF++;
    }
}

if(parkingSpotFound == false){
    if(sideSensorValueB > 50){      // -- || --
        sideSensorGapCounterB++;
    }
}

if(parkingSpotFound == false){
     // Check if the car should start parking
    if ((sideSensorGapCounterF > 5) && (sideSensorGapCounterB > 5) && sideSensorValueF < 30 &&
        sideSensorValueB < 30 && sideSensorValueF > 5 && sideSensorValueB > 5){
        stageMoving++;
        parkingSpotFound = true;
    }
}
```

For the car to find a gap a more simple solution was made. The car is not measuring the gap, rather than just detecting that there is a gap. A notable difference between detecting and not detecting a gap from the simulation code to code for the car is that the sensors in simulations are giving back a value of -1 when “nothing is detected” as mentioned before. This is why the counters are increased when the sensors read more than 50, and not less.



The graph displays how the two side sensors read during the parking sequence in the simulation. The information of importance is rounded and descripted below. The unit of the Y-axis is meters, and the the X-axis is in timestamps from the simulation.

When the sensors doesn’t detect anything (nothing is closer than 3 meters), it sets the sensor value to -1.

1. IR-sensorFront senses start of object
2. IR-sensorBack senses start of object
3. Gap is found by IR-sensorFront
4. Gap is found IR-sensorBack
5. IR-sensorFront senses start of object
6. IR-sensorBack senses start of object

Overtaking

Lena Vartanian and Eyþór Atli Einarsson

Introduction to Overtaking

The following section describes our process of developing our overtaker. It also displays and explains the final logic and implementation. However it is worth mentioning that although the logic works in both the simulator and on the car - without the motor running - it has yet to be implemented with a running motor. We aim to have that completed by the end of this week, by May 21st.

Development Process

When we began working with the *Overtaker* part we looked at the code we were given from the teacher and we played around with that code in OpenDaVINCI simulation and got it working quite well there. We still did not want to use the code given from the teacher so we began to write our own with the teacher's code as a guideline.

In the *Cockpit* we checked the records of which sensors we had to use in our overtaking part and found out that the sensors to use were the *Ultrasonic front center sensor (US center)*, which would decide when to start the overtaking process. Both of the infrared side sensors would be used to make the turning on the left lane as well as decide if we had passed the obstacle. We thought about the *Ultrasonic front right sensor(US right)* to use as well but for the time being we did not think we needed it.

Knowing this, we began hardcoding the overtaker using counters for each step/state we had to take. The only sensor we used was the UltraSonic center sensor to determine when to start the overtaking process. We had difficulties getting it to initiate a left-turn at the right time because of inconsistent sensor values. The results of the overtaking largely varied depending on circumstances then unknown to us. The accuracy of the values seemed to largely increase the closer an object got to the car. The car's velocity slightly complicated while testing, since its boost caused it to get too near the object before sensing it, causing a crash. Trial and error revealed that our errors were due to some minor issues in the Arduino code, which luckily could be adjusted to work fairly simply.

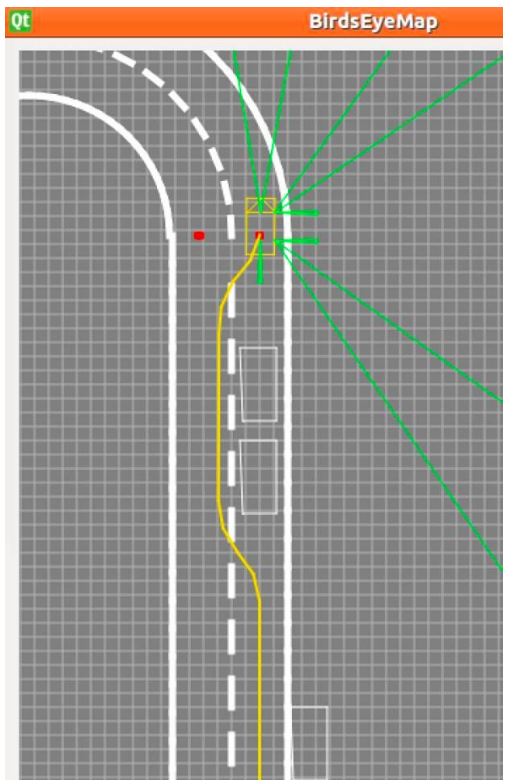
Final Overtake Code

The overtake function is based on a simple series of states. Each state has a trigger which enables it to switch states in order to do a full overtake. The entire overtake goes through the following six steps.

1. m_startingStage
2. m_moveToLeftLane
3. m_turnCarOnLeftLane
4. m_straighten
5. m_turnBackToRightLane
6. m_startingStage

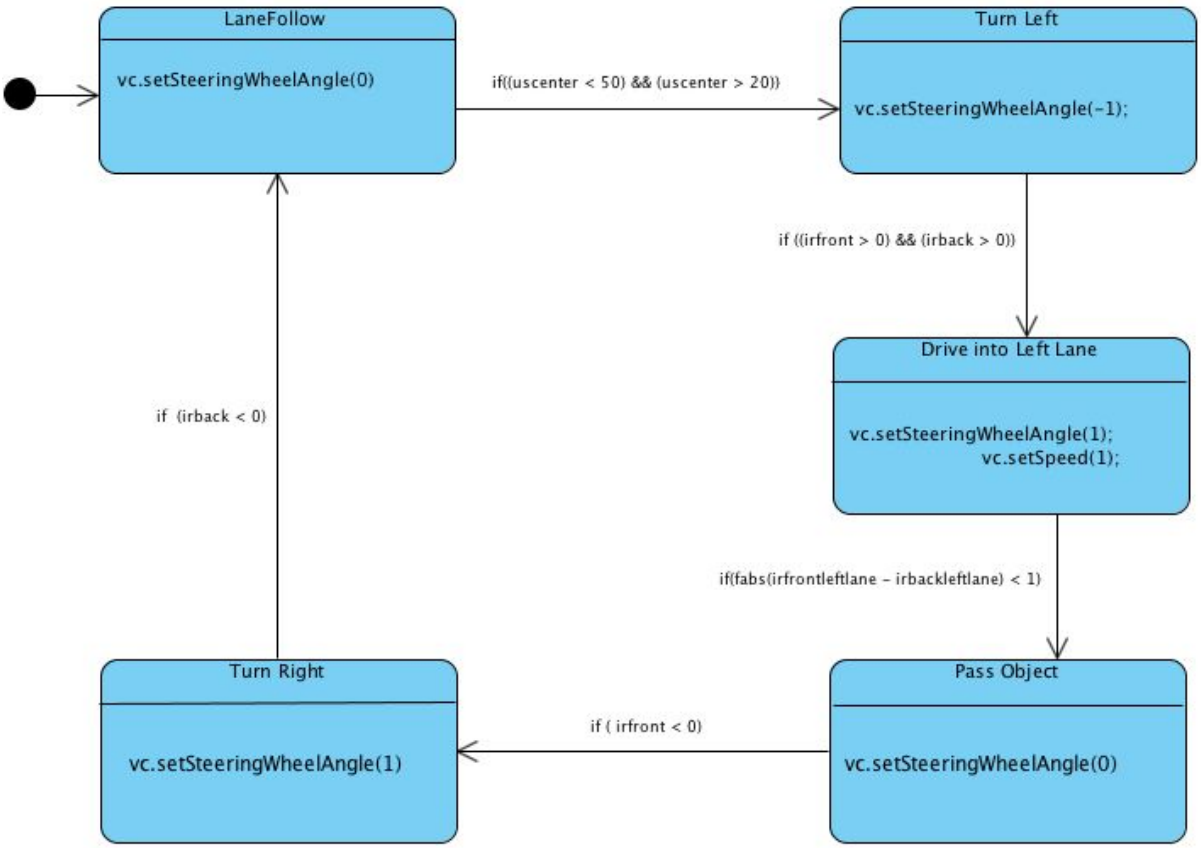
The first state is the regular lane-following in which the car drives according to the lane-follower's rules. It continues to do so until an object is detected within a range of 20-50 centimeters from the vehicle. As soon as an object has been detected within range, the car initiates the second state which is the left turn. The steering wheel angle is set to -1 and the left turn is initiated.

When both infrared sensors start receiving values, the third state is initiated and the car makes a slight right turn until the infrared sensors receive practically identical values. Once they receive similar values, the vehicle is assumed to be aligned with the object it is overtaking and switches states into m_straighten. The steering wheel angle is set to neutral and the car continues driving straight until the infrared front sensor stops receiving values. When the obstacle is no longer detectable, it is assumed that it is safe to turn back into the right field and the fifth state is initiated. The steering wheel angle is set to 1 and the right turn continues until the back infrared sensor no longer receives values.



As soon as the back infrared sensor stops detecting the obstacle, the vehicle returns to its initial m_startingStage.

The following UML state machine diagram displays the states of overtaking and which triggers initiate the vehicle to switch into each of them.



Implementation

While implementing the function of overtaking an obstacle, the first event to trigger the vehicle to commence overtaking is detecting the obstacle with the ultrasonic front sensor. As soon as an obstacle has been detected within range, it will switch states to initiate a left turn.

```
if(uscenter){
    vc.setSpeed(1);
    switch(m_state)
    {
        case m_startingStage :
        vc.setSteeringWheelAngle(0);
            if((uscenter < 50) && (uscenter > 20)){
                //startingcounter++;
                m_state = m_moveToLeftLane;
            }

            break;
        case m_moveToLeftLane :
                turnLeftFromRightLaneCounter++;
                vc.setSteeringWheelAngle(-1);
```

In order to lane-follow in the left lane while overtaking, the infrared sensors were used to ensure a safe passing of the obstacle. As soon as the car has moved into

the left lane, it starts listening to the ir-sensor values. Once it has detected values, it starts to turn right until aligned with the obstacle. The difference between the front and back ir-sensors determine whether or not the vehicle is parallel to the obstacle.

```
            // IF USING IR FRONT SENSOR
            if ((irfront > 0) && (irback > 0)) {
            m_state = m_turnCarOnLeftLane;
            }
        break;
    case m_turnCarOnLeftLane :
        // Turn out to left lane until InfraRed sensor detects the
           object, Keep incrementing counter to know how much to turn
           back later.
        cout << "******In TURNCARONLEFTLANE stage******" << endl;
        cout << "IRFRONT :" << irfrontleftlane << endl;
        cout << "IRBACK :" << irbackleftlane << endl;
        vc.setSteeringWheelAngle(1);
        vc.setSpeed(1);

        if(fabs(irfrontleftlane - irbackleftlane) < 1){
            m_state= m_straighten;
        }
```

Once it is parallel, the vehicle sets the steering wheel angle back to neutral so that it can continue to pass the obstacle. It initiates a turn back into the right lane when the front infrared sensor no longer detects the obstacle. As soon as the back infrared sensor no longer detects an obstacle, the vehicle returns to its initial lane-following state.

```
    case m_turnBackToRightLane :
        // If Front InfraRed and Front Right Ultrasonic is not detecting an object turn back half of
           counter. Go back to lanefollowing.
        cout << "*****In TURN BACK stage*****" << endl;
        cout << "IRFRONT :" << irfront << endl;
        cout << "IRBACK :" << irback << endl;

        vc.setSteeringWheelAngle(1);
          //if using the ultrasonic back right sensor
         if  (irback < 0){
         m_state = m_startingStage;
         }
        break;
```

Retrospective

Eyþór Atli Einarsson

Looking back at the overall progress during this project there are things that we could have done a lot better and there are things that we did quite well.

What went badly

We all agree that our planning should have been done a lot better. There were long periods of time where people were not sure about their tasks due to lack of structure and therefore a lot of time went to waste. Group meetings were not scheduled throughout the whole project and it was not until in the very end that we got some structure on being present at the same place and work together as a group. The lack of communication became an issue as well and more often it took group members a long time to respond to questions directly asked through either Facebook or Slack.

Since the lack of communication was an issue for the group we never really managed to get Github up and running like expected. People were not pushing their code to the repository and at times we had multiple versions of the same file in multiple locations. So not everyone was up to date when changes had been made.

During the most hectic times the room where the track was located was overcrowded and everyone was testing out their work but at the same time interfering with other people's work.

The supervisors were not really helpful since most of the time they were not around. We did not meet the supervisors assigned to us more than three times during the whole project. Some of it was our fault but we do think they could have done better.

We did lose a group member quite early in the project but after that member had a talk with the teacher it was decided the member was back in our team. Unfortunately we did not receive anything from that member and during the whole project we were always uncertain if the member was in or out.

Like mentioned previously in our report we did encounter some problems with the hardware. That slowed down our progress and we could not test out our code on the car until some of those problems were figured out.

What went well

Even though the group work was not at its best for the bigger part of the project we did manage to get the work done and did not let personal differences hinder us at our path towards a common goal. We had pretty good three weeks during the end where efficiency within the group rose to unforeseen heights.

One supervisor in particular was of great help to us and without him walking around we would, at times, have been really lost in the things we were doing. Even though he was not our supervisor he was the one of most help during the our project work.

Improvements

Improvements were made during the project on the planning part but we do all know that it could have been improved a lot earlier. With all of us having clear task and clear deadlines the overall productivity would have been a lot better. As well if we could have worked together in the same place we could have tackled two obstacles at the same time, the communication problem and the planning problem.

Having daily stand ups or at least some sort meeting where people shared what they had done and what they were having problems with would have been helpful. If doing that with the presence of a supervisor it would even have been more helpful.

Links

Trello:

https://trello.com/b/tQ7bH8r0/projectcar

Github:

https://github.com/ViktorLantz/Group6CarProject