Introduction:

This paper is intended as a documentation for my participation in the Carolo Cup competition, it includes all main contributions during the development of the project, as well as personal reflection regarding some of the problem we have had.

- **Carolo Cup:**
  Carolo Cup is the name of a competition in autonomous vehicles that is held every year on February in the city of Braunschweig in Germany *(1)*. The goal is to bring teams from different universities at bachelor and master level, in order to compete with their autonomously driving miniature vehicles in different objectives which are:
    - Follow their own lane.
    - Handle right-of-way at intersections.
    - Park automatically on a sideway.
    - Overtake obstacles.

- **Project Goal:**
  The goal of our project was to build a car that can compete in the Carolo Cup competition, this is however not as simple as just building a car that can follow the lane, overtakes object and park itself. As this competition is also based on the car's performance in achieving these tasks at a high speed, without making any mistakes. Our car was going be evaluated on the algorithms we implemented, the energy consumption, the way we present it. So there were a lot of details we had to take care of in order to be able to compete against the other universities.

- **Main Tasks:**
  I worked on many different tasks during the development of the project, I tried to contribute to the various parts of the project in order to expand my knowledge and experience in such a field as well as benefiting the group from my efforts, the main parts of the project that I was involved in include:
    - Image Processing.
    - Object Detection.
    - Mapping.
    - Software Architecture.
    - Communication.
    - Parking.

Object Detection (Lidar):

One of the tasks that I worked on during the development of our car was object detection, this was done using the **Lidar** which is a laser based sensor with 360 degrees of scanning view, it is mainly used for detecting objects; however it can be used for other purposes such as establishing the current position of an object on a global map.

- **Specification** *(2)***:**
  - **Voltage:** 3.3 V.
  - **Rotation Speed:** 315 RPM.
  - **Range:** 4.5 Meters of accurate data.
  - **Weight:** ~195.3g
  - **Full Cycle Reading:** 200 Millisec.
  - **Data Communication:** Serial USB.
  - **Baudrate:** 115200.

- **Method of Data Extraction:**
  The Lidar works in the form of flooding, there is no communication in the form of request and answer, and instead, as soon as the Lidar is turned on it will start to send data regularly the system that's handling it *(3)*.

  - **Lidar Data:**
    A full cycle will result in **90 packets** of data, the length of each packet is **22 bytes** and contains **4 consecutive readings.**
    Simple math can show that this will result in data that covers **360 readings**, each reading covers **one degree**. The amount of data would be exactly **1980 bytes**.

  - **Packet Format:**
    Each packet is divided into a number of bytes that
    - **Start Byte:** Indicator of the start of a packet, it's always "**0xFA**"
    - **Index Byte:** Indicating the index, going from 0xA0 (packet 0, readings 0 to 3) to 0xF9 (packet 89, readings 356 to 359).
    - **Speed Byte:** Two bytes (It's in the form of Little-Endian), it indicates the speed of the motor, in a **64th of a 1 RPM**.
    - **Data 0, Data 1, Data 2, Data 3:**
      Each is 4 bytes and is organized as follow:
      Byte 0: <distance 7:0>`
      Byte 1: <"invalid data" flag> <"strength warning" flag> <distance 13:8>`
      Byte 2: <signal strength 7:0>`
      Byte 3: <signal strength 15:8>`
    - **Checksum Byte:** Two bytes, checksum to insure the quality of data.

- **Creating Points for Objects:**
  The data we extract from the Lidar for each degree around it, will contain the distance of a possible object to the Lidar's position. Using simple mathematical calculations we utilize the distance between the Lidar and the object as well as the angle (in degrees) of the object to the Lidar, that way we will be able to map the object's position in the form of (X, Y).

- **Objects in Global Position:**
  We convert the points we calculated locally, into a global positions in order to establish the object's location on our map. The Lidar's current position is calculated based on the car's current position (This is done somewhere else), we pass the current position to the Lidar's programme and we use it to convert the detected object's points into global position with a mathematical formula (See Jose Estevez's paper).

- **Data Storage:**
  The data that was extracted from the Lidar and converted into global positions, will be passed from the C programme into Erlang using Nif-functions. The data we retrieve will be stored into an ETS table (a form of Erlang data structure), the ETS is a global table that could be accessed by other processes and modules of the programme in order.

- **Data Quality:**
  We've experimented with the Lidar for some time before we decided to implement it in order to establish how reliable the data readings were, we've came up with some simple but accurate sensor reading which are as follow:
    - When the Lidar wasn't moving at all, the data readings were quite accurate at a range of 4.5 meters.
    - When the Lidar was on the move (we tested it with a speed of around 1.5 meters per second), we noticed a drop in accuracy for objects that were over 3 meters, so we decided to ignore any data that we receive that are over 3 meters in range.

- **Issues:**
  There are a number of issues that we've encountered with the Lidar which can be summed up into:

  1. **Slow Data Update:**
     The Lidar's update for data is pretty slow because it finishes a full cycle in around 200 millisecond, and that is pretty slow if you wanted to overtake an object at a high speed. One way to avoid this issue is to programmatically limit the Lidar to read only 90 degrees (In front of the car) instead of 360 degrees around the car.

  2. **Slow Readings:**
     Even though we wrote the programme in a very low level way with C, the reading of the data was still very slow. This has nothing with our approach, but more to the fact

that a very large number of the data that was provided by the Lidar was failing the Checksum or were simply unreadable, and since we tend to ignore most of those, the resulting proper data are very slow.

3. **Shut Downs:**
   At some points during our programmes run time, we noticed that the Lidar simply stops transmitting data out of nowhere, we weren't exactly sure why, we simply knew that when this happened, we had to reset the device's power. During the final weeks Zlatan applied a hardware fix that reduced the problem, but it still occurred sometimes.

4. **The Need for Open Space:**
   The Lidar can get easily effected by objects around it, it can't see through even plastic material; this means that in order to get clear readings for 360 degree, the Lidar would have to be put on top of the car, somewhere where it can't be effected by anything close by around it. Even if we only wanted parts of the reading, we'd still need to have a free open space for it in the car, and this would leave the car's shell looking very ugly.

I.M.U 3-Space:

We used the I.M.U 3Space in order to calculate our car's heading (Y axis of the car).

- **Specification** *(4)***:**
  - **Weight:** 17 g.
  - **Orientation Range:** 360 degrees about all axes.
  - **Orientation Resolution:** < 0.08 degrees.
  - **Accelerometer Resolution:** 14 bit.
  - **Gyro Resolution:** 16 bit.
  - **Compass Resolution:** 12 bit.

- **Method of Data Extraction:**
  The I.M.U data communication works in a very simple form of (Request/Reply), where we write into the USB's serial port a specific value indicating the type of data we want, and the I.M.U calculates and sends back that data.
  A list of all values for the request, and their replies can be seen in references (5), at page 15.

- **Method of Implementation:**
  Since we were mainly looking for the car's heading, all we needed to find is the I.M.U's **Y Rotation**. We tried many alternatives, we started with **compass orientation** and took the **Y axis** from it, but it was unreliable and kept on changing randomly, we figured later on that this is a problem related to interference from other devices. So then we went for the **gyroscope** and requested its **Y rotation**, it worked fine, however sometimes we

would lose readings completely due to a phenomena known in physics as **Gimbal Lock**; we tried to solve it mathematically for a while, however we didn't manage very well. Finally we found our solution with quaternion orientations, however this one required some mathematical conversion from the four axis's we were receiving from thee I.M.U, into the dimensional space where we can use the Y axis as the heading. This worked fine as well, but after a while we started to notice that the readings aren't very accurate due to electronic interference.

- **Issues:**
    - **Electronic Interference:**
      One of the biggest problem in any 3space device that is dependent on gyro or accelerometer, is the existence of electronic interference from other devices that are close by. This caused our reading for the heading to change in small degrees regularly. The I.M.U is said to be taking care of such problems, and it does seem to reduce the interference, but not completely exterminate it, and since our heading has to be extremely accurate, the I.M.U needed up being a pretty bad solution for our car. We first noticed these interferences when we were using the gyroscope and the quaternion orientations.

    - **Vibrational Interference:**
      Another form of interference we were getting is the vibrational type, this type is very limited and small, but it was still enough to ruin our calculations for the heading. This can be seen during the movement of the car and mostly when using the compass orientations, since any shaking of the device can slightly change the readings.
    - **Calibration Issues:**
      One solution for taking care of the interferences were to calibrate the device's filters to take care of those interferences; while this seemed like a good idea for a while, we had to do the calibration regularly; and it would only work for some circumstances, if there are any new circumstances we'd have to recalibrate. For example if we calibrate the device's filters for when the car is standing still, then move the car, the filter's calibration wont' work, so we'd have to calibrate the device on the move.

Project Structure:

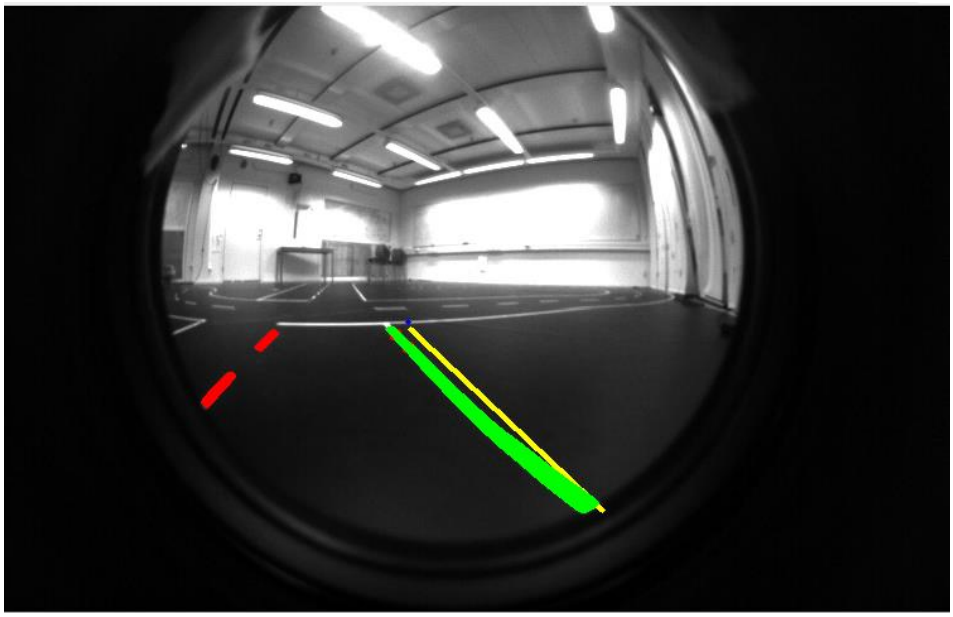
Another task that I've worked on during the beginning of our project was setting up the project's file structure. That meant setting up file structure for Erlang, C and C++ files, as well as the necessary libraries we needed to include, and final taking care of compiling and linking the binary files together.

- **Rebar:**
  We used an Erlang based file structuring system known as rebar, the reason behind that is because a very large portion of our programme is written in Erlang and might require the support of some special Eralng libraries. Rebar can take care of not only structuring the Erlang files into source, include, libraries, and binary files, but also contains a separate file structure for external languages, which is where we put our C and C++ source and include files.
- **Scripts for Compiling & Running:**
  Compiling the Erlang files was taking care of by Rebar manually, all we had to do is link the external libraries that might be needed, and let Rebar does the rest for us. However the compilation of the C and C++ file was trickier, we created make files in order to compile most of them. We then created Bash and Python scripts in order to take care of including all the external libraries that the compilation phase might need, as well as linking and connecting all the binaries together as a final step.
  The bash scripts included compiling the Erlang modules through Rebar, and C and C++ files through Make scripts; we had different command for compiling both of them together, or quick separate compilation of specific files and modules. And finally we had scripts for running the entire project with special keywords in order to specify the running phase, which could later on include running lane following mode, parking mode, overtaking mode, testing and debugging mode.

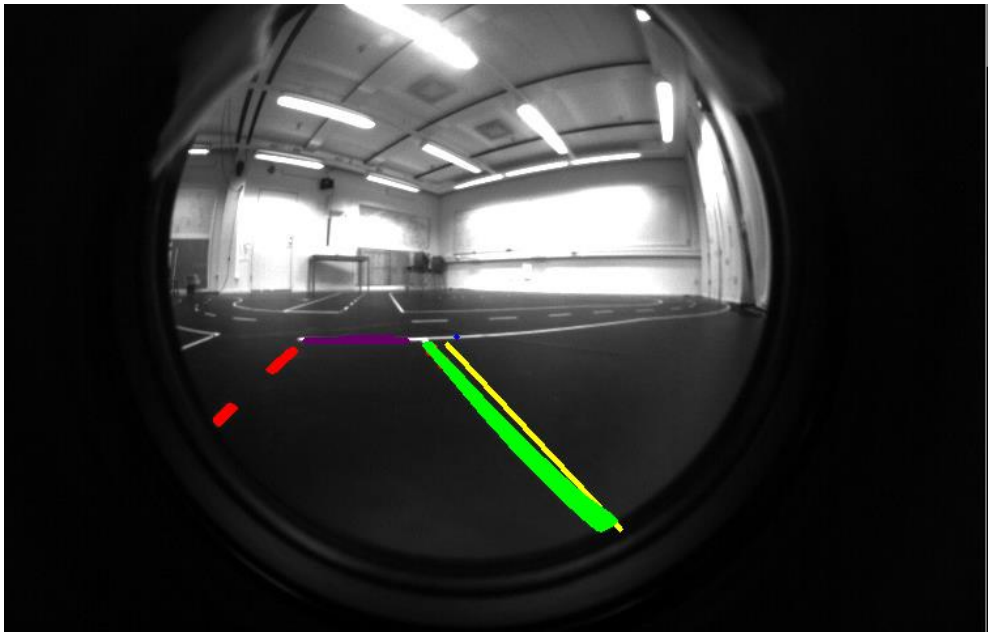
Image Processing:

I worked on special parts of Image Processing during the final days of the project, my work was mostly related to the following:

- **Finding Intersection:**
  There has already been some ideas for algorithms that we could use in order to find the intersection, one of which was looking for a (70-110) degree angle between two connected lines in front of the car. However this was a really bad solution as there were a lot of lines that were being detected as an intersection which shouldn't be.
  Robin and I came up with a solution which was much more accurate however it still had some problems, our solution was an enhancements for the previous idea we had before, the idea is to use a number of rules matched with a region of interest in front of the car in order to find the intersection; the rules are:
    - The right line must be detected.
    - A horizontal line must be found on top of the right line.
    - Three points are taken from those two lines.
    - The three points must be at the lower part of the image.
    - Find an angle that is between 80 and 100 degree.
    - Confirm that the angle is within a specific range from the right line.
    - The three points that create an angle between 80 and 100 degree, must be at a horizontal level that matches a dash line, this is for confirmation.



- **Stop Line:**
  We've set up a similar algorithm for finding the Stop line, which is dependent on finding the intersection at the road, the algorithm includes the following rules:
    - Identify a valid intersection with the method mentioned above.
    - Confirm the intersection against a neighboring dashed line.
    - Identify two points that form a horizontal line in front of the car.
    - The two points must be at the lower part of the image.
    - The two points must be to the left side of the right lane.
    - Match the intersection points with the stop line points for confirmation.



- **Issues with the Algorithms:**
  One very bad issue with both our algorithms is that, the stop line is dependent on the existence of an intersection, which in most cases is alright. However the intersection itself is dependent on the existent of a dash line to the left for confirmation, which during Carolo Cup didn't work out because that dash line was missing.
  The reason we did it this way is because we wanted a very accurate method, and we were told that the dash line will never be missing at the intersection.

Recording Map:

Another task that I worked on during the final days of the project is figuring out a way to hard code the map. This was a last resort to be used in case our plan for recording the map during the first run of the car would fail.

- **I.M.U 3-Space:**
  The first way of recording that we came up with was the implementation of I.M.U as a tool for recording, since the I.M.U contains both a gyroscope and an accelerometer, we could use it to create global positions of the I.M.U, that way we can just move the I.M.U around the track and record those positions. However this method didn't work well because the I.M.U was not very accurate, and we were getting too many interferences from external sources. The map we recorded with it looked okay, but there were a lot of points that didn't match on the track, we could fix this with some sort of a filter, but we didn't have time to look into it.

- **Laser Distance Measuring Tool:**
  The second way we came up with is using a laser based tool that can give us the distance from our current location to another object. We used the Wall as a reference point, and created a graph of X and Y point; naturally those points were in a global perspective, and so we were able to create a pretty accurate map. However this process was very tedious and time consuming, and required us to be very precise in taking the measurements.

C – Erlang Communication:

A rather important task that I worked on was the communication between the Erlang modules with the C/C++ programmes, we tried different ways to achieve this, which are as follow:

- **Nif Functions** *(6)***:**
  This is perhaps the best library that we could use between Erlang and C, it provides us with a very large number of functions that helps us in sending information from Erlang to C. However the main problem with this library is that it's a one way communication, we could send data, or call functions from Erlang to C, but not the other way around.
- **Erlang Interface** *(7)***:**
  Another Erlang library that takes care of a large portion of the communication between the two languages, however it had the same problem as Nif Functions in terms of making calls from C to Erlang. Also the library is very complicated and hard to get used to, that's why I recommend using Nif Functions instead.
- **ALE:**
  This is an Erlang library that is still being developed by Erlang Solutions, it takes care of setting up direct communication between Erlang and the hardware; it's a pretty good way to skip the middle man (in this case C), when interfacing with hardware. However as I mentioned it's still experimental and there aren't a lot of documentation or support for it, we were able to make some simple hardware control with Erlang, however we couldn't do complication things.

Parking:

During the last two days of the project I had to work on setting up a hard coded scenario for Parking along with Zlatan.

- **Procedure:**
  We created a finite state machine that is based on time which will allow the car to move forward until it finds a parking spot, then starts the parking phase by moving backward with wheels turned to the right, and then more back with wheels turned to the left and then go a little forward with straight wheels. The scenario works nicely with a large open space for parking, as well as slow speed.
  However later on we decided that speed might be important for parking to score more points, so we decided to make the process dependent on the ticks of the HAL sensor instead of basing it on time. Unfortunately this didn't work out too well because the HAL sensor was malfunctioning a lot and producing a random number of ticks that we simply couldn't depend on for parking, so we went back with basing the algorithm on time.

- **Problems:**
  There were so many problems with creating and implementing the algorithm with parking, some of which are:
  - Time constraints, we were literally working against time, completely stressed out and to make it even worse, there weren't a lot of communication with the rest of the team because they were too busy working on other parts that had troubles as well; that is why I strongly believe that parking must be done during an early phase of the project, just to ensure that you have it to score points and move on into more important tasks.
  - The car was completely busy most of the time, so we couldn't run our test on it, instead we had to use our other car, which had many problems by itself, not to mention that our algorithm might work on this car, but maybe it won't work on the real car, hardcoded solutions always never work during integration phase.
  - During the last day, our side car's PCB board was burned so we had nothing to test with since the real car was with the rest of the team testing lane following.

References:

1. https://wiki.ifr.ing.tu-bs.de/carolocup/wettbewerb/2014/termine
2. http://profmason.com/?p=13246
3. http://xv11hacking.wikispaces.com/LIDAR+Sensor
4. http://www.yeitechnology.com/productdisplay/3-space-usbrs232
5. http://www.yeitechnology.com/sites/default/files/YEI_3-Space_Sensor_Users_Manual_Wireless_2.0_r13_28Aug2013.pdf
6. http://www.erlang.org/doc/man/erl_nif.html
7. http://www.erlang.org/documentation/doc-5.2/pdf/erl_interface-3.3.2.pdf