DAT295: ROS Migration to Gulliver

Viktor Nilsson, vikni@student.chalmers.se

Herman Fransson, hermanf@student.chalmers.se

Chalmers University of Technology

Gothenburg, Sweden

Version no: 1.0

February 18, 2015

This page has been left blank intentionally

Contents

This page has been left blank intentionally

Abstract

This report describes our work behind migrating the Gulliver platform for autonomous self-driving cars to use the Robot Operating System (ROS). The purpose of the project is to evaluate if ROS is a suitable platform for a small-scale car used to test ideas and algorithms related to autonomous driving. The report goes into detail regarding the hardware that were selected and theory behind techniques used. This includes the various sensors and controllers that were used and how drivers for these were implemented, along with descriptions of the Kalman filter, PID-controller and the kinematic bicycle model.
The migration is considered to be successful. ROS modules for all hardware were implemented and the car were eventually able to drive several laps autonomously along an existing test track. However the error rate from sensors is relatively high, which for example makes the interpreted absolute heading of the car drift by several degrees in just a few minutes or rotations.

Acknowledgement

We would like to thank our supervisor Thomas Petig, to whom we are very grateful for his support in answering all our questions, pointing us in the right direction and fixing all our broken hardware.

1 Introduction

Autonomous self-driving cars may be seen as an extension to the active safety features and driver assistance systems seen in present cars. Removing the human factor may decrease the risk of many accidents. This would also open up a lot of other possibilities such as car-trains to increase the efficiency of traffic. However, this field of research still has a lot of research and testing left to be done. Many research projects are using platforms developed in-house, which may give efficient and elegant solutions but does not foster code reuse.

Our project is to migrate the Gulliver platform to use ROS. This would make it possible to reuse code and algorithms developed by others, which would decrease the time until a useful test-bed is done. It would also ease a migration to a different platform, which will most certainly be the case if an actual product is to be developed.

1.1 Related work

Similar projects to this one has been done, although their focus in the implementation is a bit different. In [1], the authors tried and succeeded to port the AUTOSAR platform to a Raspberry Pi. The main goal was to give an opportunity for students and researchers to work with the AUTOSAR platform which is in most cases closed source. The main goal in this project is also to migrate a platform but instead of AUTOSAR, ROS will be used. The authors of [1] did also develop a similar platform as Gulliver which utilized the ported version of AUTOSAR, called Mobile Open Platform for Experimental Design of Cyber-Physical Systems (MOPED) [2].

Another related project is [3], were Ford is developing a robot, capable of steering one of their cars. The reason behind the project was that Ford wanted to be able to perform consistent test runs when testing the stability of their vehicles and the only way to do this was with the help from a robot. This means that they also had to, for example, develop some sort of path following control algorithm.

1.2 Our contribution

In this project we have successfully migrated the Gulliver platform to use the Robot Operating System. This includes ROS nodes for the inertial measurement unit, ultra-sonic sensors, motor controller and parsing of joystick commands. The software that runs on the actual motor controller has also been extended to accept various extra commands related to the servo steering [4]. A relatively simple autopilot has been implemented as a ROS node. It follows waypoints created in a self-developed application, that also has the functionality to plot position updates received from the Gulliver car. Our position estimation is able to incorporate data from odometry, gyroscope as well as from an external reference system such as RCMs (See Subsection 2.2.5), with the fusion being made by a Kalman filter (See Subsection 2.1.1).

We have evaluated the use various sensors. In particular, the relatively cheap gyroscope has been found to be infeasible for this kind of project. The report also includes summaries of various techniques used during the project. We regard our project to be a good starting point for others that wish to use ROS as the operating system of choice for various configurations of the Gulliver car.

2 Background

In the following chapter, the theory behind the project is presented and the hardware that were used realize the project.

2.1 Theory

The theory section covers information about the different techniques that were used in this project.

2.1.1 Kalman filtering

Kalman filtering is an algorithm that combines measurements from multiple sources. The goal with combining measurements is to achieve higher accuracy than it would be possible with individual measurements. The Kalman filtering algorithm is also able to handle noise that is likely to be present when dealing with different sensors in a real world application.

To be able to fuse two different measurements together, Kalman uses probability theory as an aid. Every state is described as a Gaussian distribution, also called normal distribution where the variance and mean values corresponds to the states value and its error rate. Kalman also keeps track of the previous state which is used in the next iteration.

The Kalman algorithm consists of two steps, the predict step and the update step. In the first step, the algorithm predicts where the new state should be with the help from a model that represents the system. When it does this step, the uncertainty grows as the model can not be an exact representation of the system as in the real world due to approximations or noise. In the second step, the update step, it combines the predicted state with the measured state and gets a combined state which is hopefully more accurate depending on the measurements [5].

One drawback with the standard Kalman filter is that it is only designed to work with linear systems. This drawback limits the use of Kalman filtering in many real world application, due to many models in the real world is nonlinear. One way to circumvent this limitations with standard Kalman is to linearize the model in every iteration of the filter to be able to apply standard Kalman filtering. This approach is called extended Kalman filtering (EKF) [6] [7].

2.1.2 PID-controller

In control theory there exist several different types of regulators that has different properties and approaches about how to regulate a system. One of the most usual controllers is the PID-controller (proportional-integral-derivative controller).

The PID-controller controls a system based on an error value which is calculated by the input value r subtracted by the output value u. The input value is the value the user want the system to be at and it is also called the set point. The output value is the control signal that is sent to the system that should be controlled. An example of what the variables r and u can stand for is the desired speed and the throttle response when implementing a cruise controller[8].

A PID-regulator consists of three different components; a proportional component, an integral and a derivative component and these three components affects the controller differently. These three components are controlled through three different parameters K_p, K_i and K_d and they affects five different properties in the system. These five properties are rise time, overshoot, settling time, steady-state error and stability[8].

How the three parameters K_p, K_i and K_d affects the different properties when they are increased are shown below:

Parameter	**rise time**	**overshoot**	**settling time**	**steady-state error**	**stability**
K_p	decrease	increase	small impact	decrease	worse
K_i	decrease	increase	increase	removes	worse
K_d	small impact	decrease	decrease	none	increase

Figure 1: The influence for the different parameters when they are increased

2.1.3 Bicycle model

To be able to estimate the position of the robot, a model of how the robot move is needed. There exists a lot of different models depending on what type of robots that is used. The robot in this project is designed with Ackermann steering which means that the model has to take this into account. There exists models that are able to simulate robots with this type of steering but most often is it possible to simplify these models into an easier one called the bicycle model.

The bicycle model simulates a basic bicycle which is fairly comparable to a robot with Ackermann steering. Instead of having two wheels at the front that are responsible for the steering, it has been simplified to one wheel in the bicycle model. There exists two different types of the bicycle model. One of them is the dynamic bicycle model which takes into account the dynamic forces that can occur and the other one is the kinematic bicycle model. The kinematic model is a common approximation for robots that is designed with Ackermann steering. In figure 2, a graphical representation of the kinematic bicycle model is shown [9] [6].

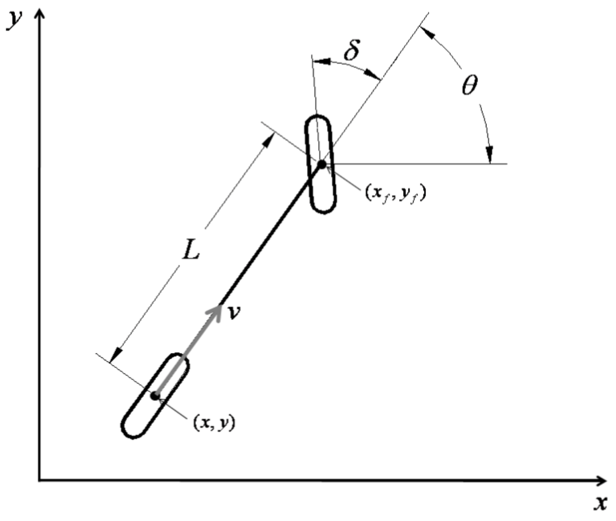

Figure 2: Kinematic bicycle model [9]

The input to the mathematical derivation for the kinematic bicycle model is the steering angle for the front wheels and the distance the robot has traveled from the starting point. The mathematical derivation model is shown below:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + \Delta d \cdot cos(\theta_k) \\ y_k + \Delta d \cdot sin(\theta_k) \\ \theta_k + \frac{\Delta d \cdot tan(\Phi)}{L} \end{bmatrix}$$

Figure 3: Mathematical derivation for the kinematic bicycle model

2.1.4 ROS: Robot Operating System

ROS is a collection of packages that are installed by the user above the host operating system and provides several functions/features that simplifies the development of the software targeting a robot. ROS also provides several already implemented nodes that can easily be reused.

One of the main features of ROS is the communication layer that can be utilized by different so called ROS nodes. These nodes are independent programs that runs in the ROS environment and offers some sort of functionality, for example navigation. They are usually written in either C/C++ or Python but ROS also supports other programming languages such as LISP by default. Other languages ports are in different states of completion. With the help from the communication layer, two nodes can communication with each other through two different kinds of communication paradigms. The first one is the publisher/subscriber scheme and the other one is the client/server scheme. With these two communication schemes the nodes are able to communicate with each other and together perform tasks such as autonomous driving [10].

Another benefit by using this communication layer is that it is simple to add more peers running in the same environment. This is really beneficial when the robot do not have much computational power by itself. Then it is possible to connect another computer to the same environment and launch the computational heavy nodes on that one instead of the robot[10].

2.2 Hardware

The hardware that has been used in this project is presented in detail in the following sections.

2.2.1 Gulliver

Gulliver "Gulliver 8" is the name of the robot that has been used in this project and it has been developed by students at the computer science department at Chalmers. The frame of the robot is originally from a 1:8 scale RC-car which has been modified to be able to fit the different sensors, motor controller and a small computer. The robot has many similarities with a regular car as it is based on Ackermann steering. This type of steering entails that it is the front axis that are responsible for the steering of the robot. The rear axis is fixed and are not able to turn at all. The drivetrain is composed of one main electric engine responsible for the driving force and one servo responsible for the steering of the front wheel. The main engine is hooked up to two differential gears making the robot four-wheel drive (4WD). The main engine is also responsible to apply braking force on the robot [11].

The power supply is handed by two separate Li-Po batteries, one for the motor controller, servo and the engine and one for the onboard computer. The reason behind having two separate batteries is to eliminate the electric noise made by the engine that might be bad for the computer. The onboard computer consists of a mini-ITX board containing an x86 Intel quad core processor, 16 GB of RAM and a 60GB SSD.

2.2.2 IMU - Inertial measurement unit

An inertial measurement unit (IMU) is a collection of typically two different sensors, a gyroscope and an accelerometer [12]. Today, some manufacturers also include a magnetic field sensor because these are often used in combination with the gyroscope and accelerometer in different implementations.

In this project, the IMU was provided by a STM32F3DISCOVERY board made by ST-Microelectronics. This board comes equipped with a L3GD20 gyroscope chip and a LSM303DLHC chip which contains both an accelerometer and a magnetometer.

Gyroscope: This devices enables the user to measure if the devices has been turned in the three different direction(roll, pitch and yaw). It has no frame of reference, every measurements are relative to its own sense of the world. Because of this, it is not possible to know at what direction the gyroscope has been turned, just that it has been turned for example 90 degrees on the x-axis from its starting position.

The output is given in θ/s which is the angular velocity of the gyroscope. The angular velocity can be converted to degrees by integrating the angular velocity $\int_o^t \omega$ where ω is the angular velocity.

Accelerometer: The accelerometer works quite in the same way as the gyroscope does but instead of measuring the angular velocity, it measures the acceleration in three different direction(x, y, z) which are orthogonal to each other. It is possible to get reference to the real world by using a accelerometer because there is always an acceleration facing downwards, towards the earth which is equal to one G ($9.82 m/s^2$). With this information it is possible to fix the orientation of one of the axes of the accelerometer in the real world frame.

The output is given in m/s^2.

Magnetic field sensor: This type of sensors measure the magnetic field around the sensor on three different directions (x, y, z). As in the case with accelerometer, the axes are orthogonal to each other. By knowing the strength of the magnetic field in different directions, it is possible to calculate which direction for example north lies at. With this knowledge and in combination with a accelerometer, it is possible to fix the orientation of all axes(x, y, z) in the real world frame [13].

The output of a magnetic field sensor is given in Tesla with an appropriate prefix.

2.2.3 Ultra-sonic sensor

An ultra-sonic sensor uses sound to measure distance to a objects ahead of of the sensor by measuring the time difference between the given pulse and the echo. This is a cheap way to estimate the distance to objects compared to other solutions, as for example a laser scanner. The main problem with this type of sensors is the uncertainty of the data given by the sensors. The uncertainty is mainly due to the sound pulse which is sent out by the sensor spreads out like a wave and not in a straight line but also due to multiple echo for the same pulse. Due to these uncertainty parameters, an ultra-sonic sensor is not suited for precise measurements but it is suited to be a last resort sensor. [14]

The ultra-sonic sensor already fitted to the robot is the SRF08 Ultra-sonic range finder made by Devantech.

2.2.4 Motor controller

A motor controller is responsible for running and controlling the electric engine. This means that the controller is responsible for giving the correct current and voltage for the engine. An example of this is if the controller is given the task to run the engine at 1000 rpm, the motor controller will try to control the rpm by adjusting the current that is given to the engine. This can be achieved with an PID-controller. By adjusting the current, the controller is not only able to adjust the rpm but it is also able to adjust the torque given by the engine or the direction of the rotation.

In this project, a motor controller made by Benjamin Vedder has been chosen. This is a custom made motor controller for a brushless DC engine (BLDC) [15]. It is equipped with a STM32F4 microprocessor running a real-time operating system called ChibiOS [16]. Besides from controlling the main engine, it is also able to control the servo that is used for the steering of the car and also provide information about the engine such as tachometer values. A tachometer is a counter that keeps track of how many revolutions the engine has done so far.

The motor controller comes together also with a GUI called BLDC Tool. With this program, it is possible to configure the controller to match a specific engine. The more fine-tuned the controller is for the engine, the more efficient it can run. BLDC Tool comes equipped with a feature called Auto Detection. It is also possible to test the engine through this program by controlling the engine directly from the keyboard or setting the rpm to a specific value [15].

Some of the specifications for the motor controller are shown below:

- Regenerative braking.
- Current: Up to 240A for a couple of seconds or about 50A continuous depending on the temperature and air circulation around the PCB.
- Voltage input: 8V – 60V (Safe for 3S to 12S LiPo).
- Interface to control the motor: PPM signal (RC servo), analog, UART, I2C, USB or CAN-bus
- PCB size: slightly less than 40mm x 60mm.
- Duty-cycle control, speed control or current control.

2.2.5 RCM

To be able to get an absolute position, some sort of external reference system is needed. Examples of this type of system is the American system GPS or the European version called Galileo. These system works well outdoors with reasonably good precision (a few meters at best) but the precision gets a lot worse or non-existent indoors. To get around the precision problem indoors another solution is needed. One solution is to use multiple anchors and and a beacon that make use of ultra-wideband (UWB) spectrum to measure the time of flight between the bacon and the anchors. With this information it is possible to estimate the position with the help of trilateration [17]. These beacons (same type of hardware in anchors and beacon) are called Range and Communications Modules (RCM).

A beacon was mounted on the robot with help from another group working solely on localization, to provide position data for testing and verification. This positioning data were also fused with the other data from odometry and the IMU to provide a more accurate and stable positioning.

3 Implementation

3.1 ROS modules

This subsection describes in some detail the purpose and functionality of all the implemented ROS software modules.

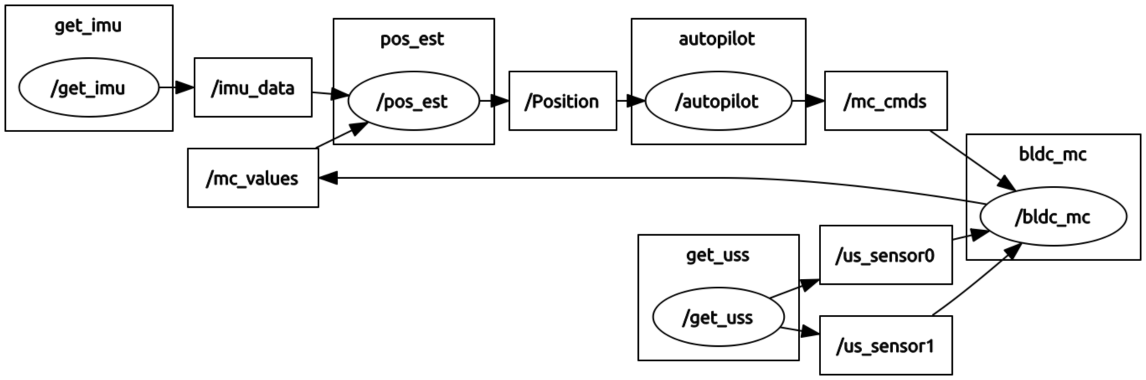

Figure 4: Interaction between ROS modules, without the RCM-localization.

3.1.1 bldc_mc

This node is the only one that talks to the motor controller. It subscribes to the 'mc_cmds'-topic of which messages it parses and sends the resulting commands to the motor controller over the serial bus. It also subscribes to the ultra-sonic sensor data topics (one for each sensor). In case of a range less than a specified margin and a speed greater than 0, it immediately stops the car. When all sensors report that the obstacle is gone, the previous speed is restored.
It periodically requests data from the motor controller which it parses and sends out to the 'mc_values'-topic as an MCValues-message. It contains several fields regarding temperature, rpm, voltage and tachometer (to name a few), but of which the tachometer value is the only one currently in use by other nodes. Somewhat redundantly, it also periodically sends 'I'm alive' to the motor controller. This is due to the design of the motor to automatically stop if it has not received any command the last 500ms.

3.1.2 estimate_position

This node tries to estimate the position of the robot by fusing data from the available sensors. When run solely on information from the motor controller and the gyroscope, the node receives odometry and steering angle on the 'mc_values'-topic and gyroscope data on the 'imu_data'-topic. From this a position is estimated by use of the bicycle model (see Subsection 2.1.3). This is done every time new odometry data is available. We get an absolute heading at time $t = k$ by integrating the gyroscope data from time $t = 0$. Unfortunately, this means that we need to know the absolute heading when the car is started.

When also using the RCM, data is received on the 'Position_rcm'-topic. These messages include, among other fields, a position and its estimated error. When combining this data with the odometry, a Kalman filter (see Subsection 2.1.1) is used to get a fused position.
The position that is finally published includes (x,y)-position and the absolute heading, along with a standard ROS header.

3.1.3 get_imu

This node periodically requests readings from the IMU (by sending 'r' over the serial port). It gets 9 values in return (x,y,z for each of the sensors). These are parsed and sent out as a message of type 'IMUData' to the topic 'imu_data', of which the estimate_position node is the only subscriber.

3.1.4 get_uss

This node periodically requests readings from the ultra-sonic range sensors. These values are published as a standard ROS message of type Range, of which the fields radiation type (set to ULTRASOUND) and range (range in meter) is used.
The only subscriber is bldc_mc.

3.1.5 joy_to_mc_cmds

The car may be driven by a generic off-the-shelf joystick. The standard ROS package 'joy' is used, which supports several different joysticks (Playstation 3 Dual-shock, Xbox360 among others). The node joy published messages of type 'Joy', which contains the current state of all buttons and axes. The joy_to_mc_cmds node subscribes to the 'joy'-topic and in turn publishes messages of type AckermannDrive which contains several fields including speed and steering angle (which are the only ones currently in use), under the topic 'mc_cmds'. In our case, using a knock-off xbox360 controller, speed is determined by the button 'R2' (which gives values in the range [0,255]). Reverse direction is achieved by holding 'A' while accelerating. Steering angle is determined by the left analog stick. The published message has a speed in the range [-1.0,1.0] and a steering angle in the range $[-\theta,\theta]$, where θ is the maximum steering angle.

3.1.6 autopilot

The autopilot subscribes to the 'Position'-topic and for each update, if the final target is not reached, adjusts the steering angle and speed and send the new command out by publishing an AckermannDrive message to the 'mc_cmds' topic. It takes as input a list of coordinates (waypoints) that it needs to pass by (with a specified margin). Angle error is calculated as the difference between the current heading of the car and the angle between the car's back axis and the next waypoint. This is used by the PID-controller which calculates a new steering angle. This is sent to the motor controller in degrees. Speed sent is either zero or a certain (static) percentage of the maximum speed of the car.

3.1.7 Waypoint creator

A relatively simple program for creating a list of waypoints for the car to follow was implemented in Python along with the pygame [18] library. The program features a map of the track (obtained by measuring the track with a laser ranger, since we did not get the blueprints). By clicking this area waypoints are created/removed. It got functionality to save the track as a text file (a list of (x,y)-coordinates) and as an image (the waypoints superimposed over the map). It may also subscribe to the position estimation ROS node, to be able to plot the where the car think it has driven. This was seen as a more suitable solution in contrast to using standard ROS tools like rviz or rqt_plot.

3.2 Motor controller

The software running on the motor controller board, originally written by Benjamin Vedder (See subsection 2.2.4) were modified to suit our needs. At first the servo was not initialized in the code, also it were only able to receive the servo command COMM_SERVO_OFFSET. Since zero offset meant straight steering and the command only accepted positive number, it could only steer right. Our modifications added servo initialization and acceptance of several servo move command, which included COMM_SERVO_MOVE, COMM_SERVO_MOVE_WITHIN_TIME and COMM_SERVO_RESET_POS. Our fork can be found at github [4].

4 Testing

All test driving were carried out on the test track in our designated lab room (See Figure 5). While developing drivers for the sensors and actuators, testing and measurements where made. An example of this is when we measured the number of ticks the motor controller reported from the tachometer. This were made by driving various fixed, straight distances (10-20 times each) and checking the tachometer value. Some obviously bad readings where discarded. Another example is when we estimated the maximum steering angle. The measurements were made by putting the wheels into full right and left and measuring the angle.

When driving the car along the track manually with joystick while plotting the estimated position, we were able to visually see how far off the positioning were. However, it was not deemed possible to get any comparable data on this, in part due to the human factor, it was impossible to drive exactly the same in multiple iterations. A plot with the position estimated by RCM in comparison with the position estimated by odometry and gyroscope is shown in Figure 6. For the autopilot, we initially tried simple paths as in "drive forward 3 m, turn left, drive 1 m, stop" or circles. In later stages of the project we measured key points of the test track with a laser ranger, to get an accurate map. This had to be done due to the that we did not get hold of the blueprints of the track. The autopilot, while estimating position based on odometry and gyroscope, succeeded in driving one lap around the track, although the positioning after this was off by around half a meter, which sometimes had made it hit a wall if not for the ultra-sonic range sensors.

The PID-controller was calibrated by simply adjusting the K_p, K_i and K_d parameters until a suitable configuration was found. This included that curves were taken smoothly, no overshoot was made. Initially there was problems with a "snake-like" driving, but this was eventually fixed.

Testing with the RCM (See Subsection 2.2.5) was at the very last stages of the project. This was first used to make comparisons between our previously estimated position, and the position estimated by RCM which was without error accumulation. By later fusing the data from the RCM with the data from other sensors, error accumulation from the odometry could be minimized by step by step pushing the estimation in the direction of the RCM position. This was done by using a Kalman filter (See Subsection 2.1.1). A plot of two successful laps while using data from all sensors can be seen in 7. Driving further than this were hardly possible due to the gyroscope drift.

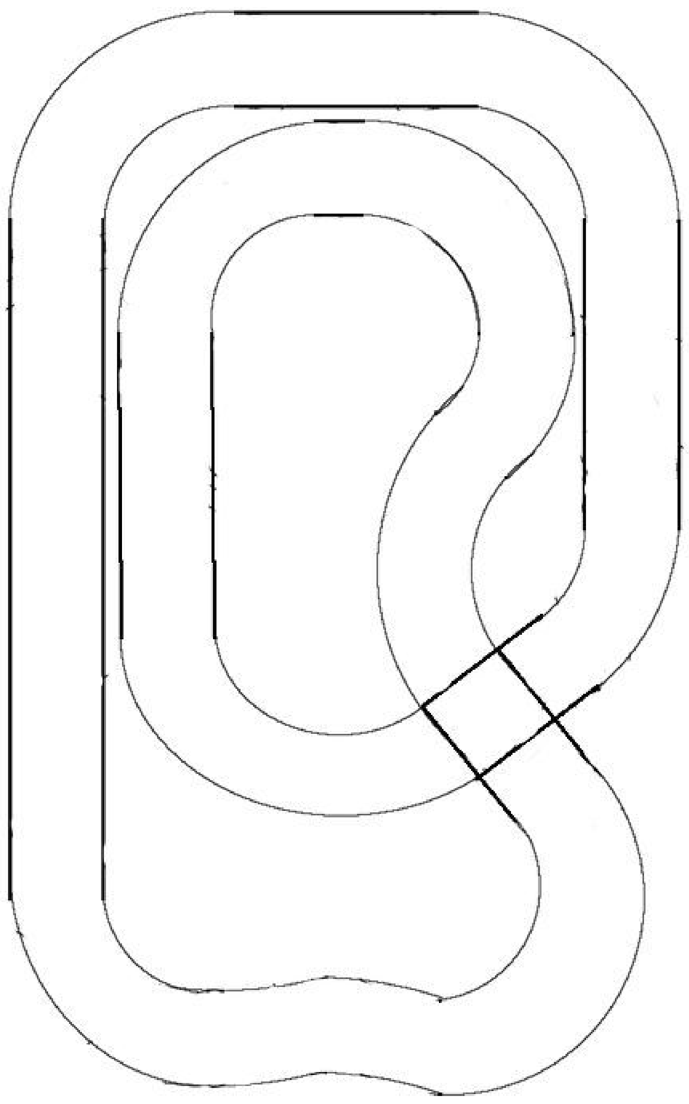

Figure 5: The test track with a length of around 45m.

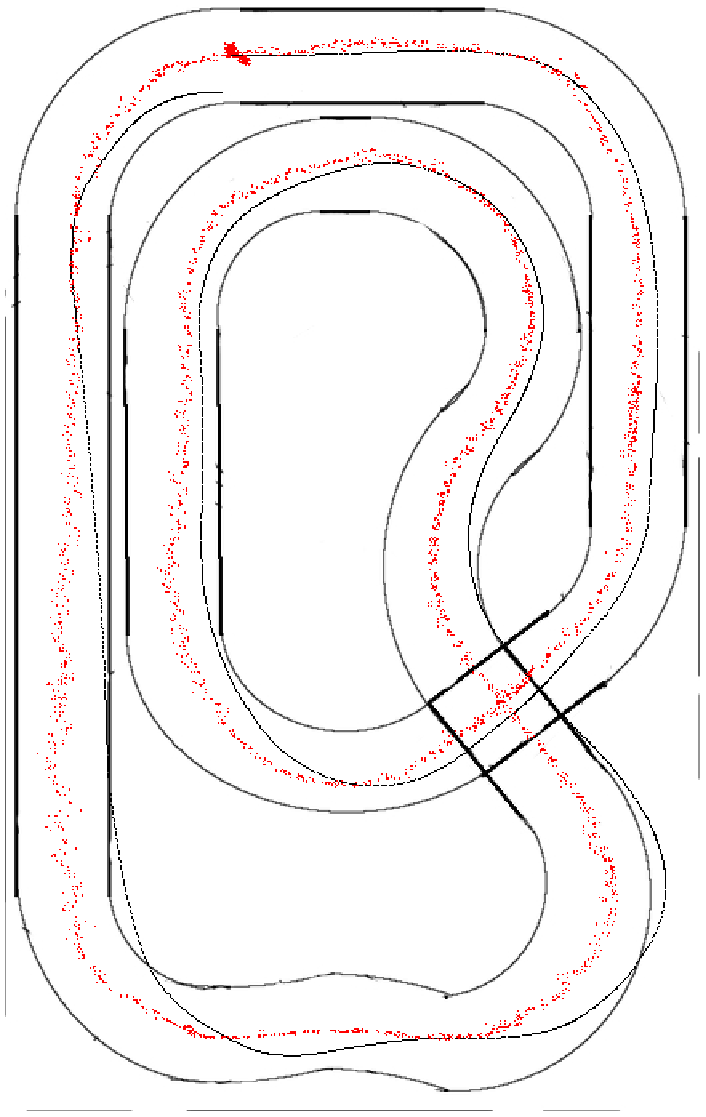

Figure 6: Estimated position with RCM (red dots) and odometry (black line), while driving manually with joystick.

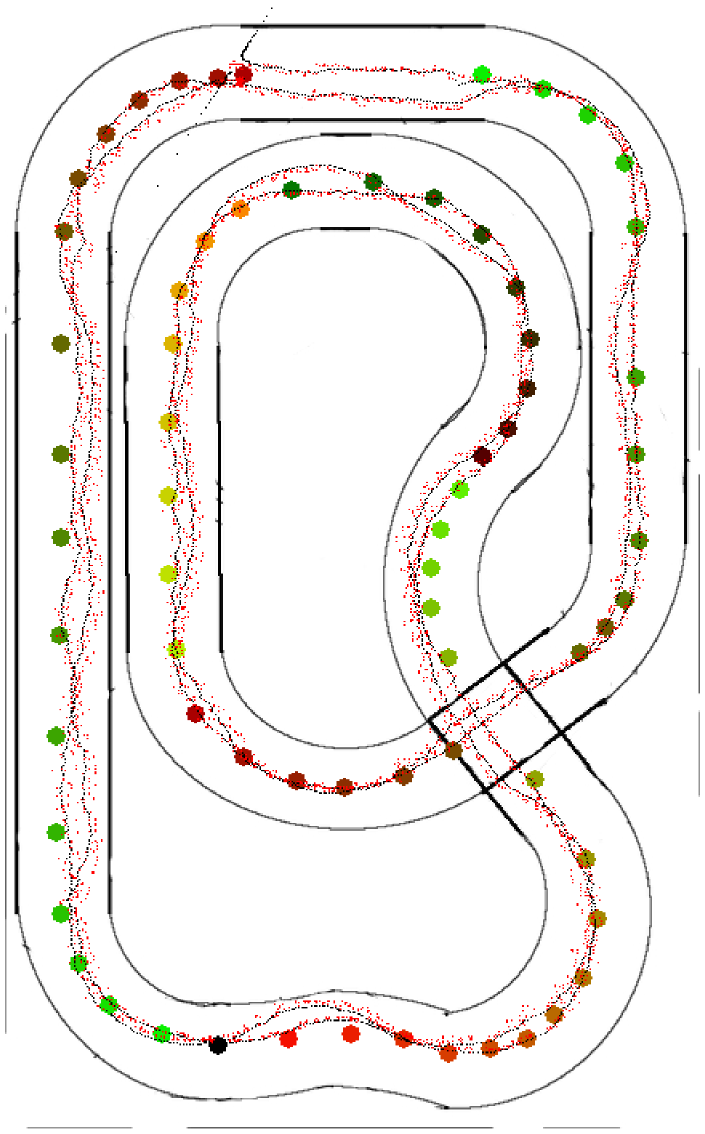

Figure 7: Position estimated by fusing odometry and RCM position (black line). Raw data from RCM is shown in red.

5 Discussion

The end result of the project was satisfying, the robot was able to drive two laps around the test track autonomously without detours, or manually with an Xbox controller and the robot used ROS as the underlying abstraction layer.

We had some problems with our initial trials with the autonomous driving because we only used the yaw axis of the gyroscope and the odometry which consists of the steering angle and the total travelled distance. If these sensors and estimations had been perfect, we would have been able to estimate our position very well without any external references but due to the quite large error rates from both the odometry and the gyroscope, this was impossible. The gyroscope drifted up to 10 degrees after two laps which is quite significant. Also the estimation of steering angle could have been more exact. We even had to re-tune the servo several times during the project since it actually drifted, possibly due to vibrations. With more expensive and more accurate sensors, some errors could have been minimized but not completely removed. This was the main reason why we also added the sensor for the external positioning system, to try to compensate for the error in the odometry and gyroscope.

To be able to fuse the sensor data from the gyroscope, odometry and the external positioning system, a standard Kalman filter were used. This gave a good result for the positioning estimation despite the erroneous sensors. To make the position estimation even better we could have added accelerometer data and the magnetometer from the IMU but for the purpose of this project, it did not seem necessary to do for achieving the goal. Another improvement that could have been done is to replace the standard Kalman filter with the extended version which is able to handle nonlinear system. The non-linearity in our system comes form the bicycle model which contains both sine and cosine functions which is nonlinear.

The bicycle model itself is another source of error. This is due to our choice of approximating the Ackermann steering with the bicycle model. If we had used the proper Ackermann model, it could have improved the position estimation. However, at the same time, it also makes the model much more complex and due to our time constraints, this was a good trade off. The model does not either take into account other things like wheel slip when turning or wheel spin which affects the position estimation.

A lot of time were spent debugging the hardware and its existing software. The documentation were many times insufficient. We could probably have gotten better results if a more mature hardware platform had been selected. The code for the motor controller board were hardly commented at all, and it was time consuming to figure out how things worked and what functionality was implemented, done or ready to use.

We found ROS to be overall convenient and we found no major hurdles in using it. An expectation was that a lot of functionality in modules were usable in our setup, but that was in part false. We were planning to use the ROS navigation stack for the autopilot part. If everything would have worked out, that would have reduced our job to only writing drivers for the sensors and the motor controller. However the ROS navigation stack requires a planar laser (a LIDAR) for map building and localization. This dependency might have been possible to remove but the software stack is relatively big and complex and we assumed it to be too time-consuming to research what eventually would have lead to a dead end. It was very convenient when functionality were in fact usable, as in the case with the ROS module for the joystick. Communication in between modules was done asynchronously with the publish/subscribe scheme as we deemed it unnecessary to use the synchronous communication, in ROS named services. This worked well and we had no problems with communication between Python or C++ modules, or similar situations that could have been an issue.

6 Conclusion

We have found ROS to be well suited for this type of projects, it can help by greatly speed up the development time for new functionalities. It also allows and encourage code reuse from other projects thanks to the encouragement to make well defined interfaces and message types.

Another insight in this project is that it is hard to parse and filter the sensor data into useful values without too much noise. This is much due to the error rate for the different sensors that are used. To be able to get something that makes sense from the sensors, some sort of filtering is needed. In some simple or approximately cases a standard Kalman can be sufficient. If a more accurate result is needed or if the system that is model is more complex, some other filtering technique is required, for example extended Kalman.

We found that an external reference system was needed to get a continuous accurate position estimation. It was not possible to use only data from the IMU and odometry for position estimation due to the error accumulation. However, this error accumulation can be minimized with the help of more precise and more expensive hardware but it can not be completely removed. An example of an external reference system has been used in this project and combined with the others sensors, it resulted in a good position estimation in the global frame.

References

[1] S. Zhang, A. Kobetski, E. Johansson, J. Axelsson, and H. Wang. “Porting an autosar-compliant operating system to a high performance embedded platform”. In: *ACM SIGBED Review* 11.1 (2014), pp. 62–67.

[2] J. Axelsson, A. Kobetski, Z. Ni, S. Zhang, and E. Johansson. “MOPED: A Mobile Open Platform for Experimental Design of Cyber-Physical Systems”. In: *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE. 2014, pp. 423–430.

[3] H. Tseng, J Asgari, D Hrovat, P van der Jagt, A Cherry, and S Neads. “Evasive manoeuvres with a steering robot”. In: *Vehicle System Dynamics* 43.3 (2005), pp. 199–216.

[4] V. Nilsson and H. Fransson. *finkultur/bldc*. 2015. URL: `https://github.com/finkultur/bldc` (visited on 02/14/2015).

[5] R. E. Kalman. “A new approach to linear filtering and prediction problems”. In: *Journal of Fluids Engineering* 82.1 (1960), pp. 35–45.

[6] C. Suliman, C. Cruceru, and F. Moldoveanu. “Mobile robot position estimation using the Kalman filter”. In: *Department of Automation, Transilvania University of Brasov, Brasov* (2009), pp. 1–3.

[7] B. Friedland and I. Bernstein. “Estimation of the state of a nonlinear process in the presence of nongaussian noise and disturbances”. In: *Journal of the Franklin Institute* 281.6 (1966), pp. 455–480.

[8] K. H. Ang, G. Chong, and Y. Li. “PID control system analysis, design, and technology”. In: *Control Systems Technology, IEEE Transactions on* 13.4 (2005), pp. 559–576.

[9] J. M. Snider. “Automatic steering methods for autonomous automobile path tracking”. In: *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08* (2009).

[10] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. “ROS: an open-source Robot Operating System”. In: *ICRA Workshop on Open Source Software*. 2009.

[11] B. Vedder. “Gulliver: Design and implementation of a miniature vehicular system”. MA thesis. Chalmers University of Technology.

[12] N. Yazdi, F. Ayazi, and K. Najafi. “Micromachined inertial sensors”. In: *Proceedings of the IEEE* 86.8 (1998), pp. 1640–1659.

[13] J. Lenz and A. S. Edelstein. “Magnetic sensors and their applications”. In: *Sensors Journal, IEEE* 6.3 (2006), pp. 631–649.

[14] Devantech. *SRF08 Ultra sonic range finder*. 2014. URL: `http://www.robot-electronics.co.uk/htm/srf08tech.shtml` (visited on 02/02/2015).

[15] B. vedder. *VESC – Open Source ESC*. 2015. URL: `http://vedder.se/2015/01/vesc-open-source-esc/` (visited on 02/02/2015).

[16] G. Sirio. *ChibiOS/RT real time operating system.*

[17] F Hidstrand and C Tadesse. *Robot Localization Using ROS.*

[18] P. Shinners. *PyGame*. `http://pygame.org/`. 2014.