SOFTWARE ARCHITECTURE FOR AN AUTONOMOUS CAR SIMULATION USING ROS, MORSE & A QT BASED SOFTWARE FOR CONTROL AND MONITORING

Tibério Ferreira*, Olmer García †, Janito Vaqueiro*

*Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas
Rua Mendelejeu, 200
Campinas, SP, Brasil
†Ph.D Scholarship PEC/PG CAPES/CNPq-Brazil.

Emails: tiberiusferreira@gmail.com, olmerg@fem.unicamp.br, janito@fem.unicamp.br

Abstract— Cars capable of driving in urban environments autonomously are today one of the most challenging topics in the intelligent transportation systems (ITS) field. This paper presents the architecture and software used in the autonomous car simulation used in the VILMA (Intelligent Vehicle of the Autonomous Mobility Laboratory) project at UNICAMP. It focuses on describing each program used in the project and the architecture which integrates them. The programs used are MORSE, ROS and a custom program based on the QT framework. Finally, the article presents how some autonomous functions like the control of orientation and velocity of the vehicle are implemented inside the architecture.

Keywords— Autonomous robotics, MORSE, ROS, Vehicle.

1 INTRODUCTION

Autonomous cars play an important role in current robotics and artificial intelligence research. The development of driverless cars started in the late ’70s and ’80s. Ernst Dickmann’s Mercedes Benz achieved a travel velocity of 100 km/h on restricted highways without traffic (Dickmanns et al., 1994). In the DARPA Grand Challenge 2005, autonomous cars drove off-road on desert terrain, several of them reaching the finish line (Thrun et al., 2006). DARPA’s Urban Challenge of 2007 demonstrated that intelligent cars are able to handle urban scenarios and situations with simulated traffic (Urmson et al., 2007). Lately, autonomous cars have been driving through real world traffic for testing purposes in urban and rural areas alike (Thrun, 2010).

In mid-2008, interested in robotics and vehicular safety, the Laboratory for Autonomous Mobility (LMA) was formed in the DMC-FEM at UNICAMP. Taking into account the model of (Siegwart et al., 2011), the three main lines of study are: Perception, Navigation and Control. According to (Thrun et al., 2006), these systems can be organized into six major functional groups: Interface, Sensors, Perception, Control, Vehicle Interface and User Interface. The figure 1 shows how these blocks interact and for this article the environment and the vehicle will be replaced by a model in a simulator. The algorithm of the blocks of speed and path controllers are tackled in this article. It is important to note that even though this article focuses on MORSE as it’s simulator, there are other simulators worth noting for robotic applications. Two of simulators tried by the project were V-REP (Freese et al., 2010) and Gazebo (Koenig and Howard, 2004).

Figure 1: Architecture of the autonomous car of LMA

The paper is presented in four sections. The first one presents the implementation of the MORSE simulator and the Robot Operating System (ROS) as well as the data exchange ways between them to the necessities of the project. The second one explains the back end and front end design of the software developed. The third
one explains the algorithms and theory of the autonomous function of the software and presents results obtained. Finally, some conclusions and futures work are presented.

2 Simulator integration

The software architecture of the simulator is based on two softwares: ROS (Robotics Operating System) (Quigley et al., 2009) and the MORSE simulator (Écheverria et al., 2011) (Lemaignan et al., 2012). Both will be introduced below as well as the way they exchange information with the supervisory software.

2.1 MORSE - The Simulator

The project architecture uses MORSE as the simulator framework in which the car is developed. MORSE is derived from Blender’s Game Engine and so Blender can be used to create vehicle models to be used inside MORSE. It allows for various parameters to be set in order to have a realistic simulation, such as mass, gravity center, suspension damping, suspension stiffness and roll influence. As Blender uses python as it’s scripting language, scripting is easy and fast. It’s based on a game engine and because of that it has various tools for debugging, optimization and real time realistic physics based on the widely known Bullet physics library (Coumans et al., 2013). MORSE and Blender are both open source, which makes it possible to add any missing critical feature for the project. MORSE also has many common sensors built into it. Most sensors used in UNICAMP’s VILMA car were readily available from MORSE’s library, such as GPS, stereo cameras and inertial measuring unit. MORSE also provides data from the Blender Game Engine itself which can be used to build any sensor necessary by getting the original accurate data and adding some kind of noise, such as Gaussian noise.

The model of the car in the Blender Game Engine (BGE) is based on scripts. There is a car setup script which sets the car wheels’ and suspension parameters which are: stiffness, damping, compression, roll influence, tire grip and suspension height. Once the setup is complete, the BGE constantly executes scripts which change the car steering, engine power and brake according to values stored in variables at runtime. These variables are the ones which allow external programs to communicate with the car and control him because they are accessible to external application through MORSE’s socket interface. This way, if an external application sends a new steering angle, all MORSE does is rewrite the variable corresponding to it at runtime and the scripts take care of actually changing the vehicle physical state. In addition the BGE also listens to keyboard events. This way the vehicle can also be controlled using the keyboard.

2.2 ROS - The Middle-ware and Toolbox

ROS acts as a middle-ware and toolbox. ROS provides multiple libraries which are commonly used in robotic problems. Some of those used in the VILMA project include PID implementations, communication interfaces between programs with it’s own publisher-subscriber system, callback queues, clocks and timers. In addition it also provides multiple graphical user interfaces tools to aid development. Some of those tools are used in the VILMA project to view real-time details of messages being published to other programs, to interpret sensor information such as a cloud of points (typical for LIDAR sensors) and to process video stream.

2.3 Data exchange between the softwares

The interaction with MORSE is done through ROS Topics and sockets. ROS collects data from the simulation and makes it available to external programs using it’s topic-subscriber interface. Sockets are used to send commands to the car inside the simulator. Sockets are used instead of ROS topics because the interface to do so using sockets was already developed by USP/SC and used in their CaRINA project (Fernandes et al., 2014). Also, using sockets demonstrates how different middle-wares can be used together without problems as seen in the Figure 2.

![Figure 2: Simplified simulation architecture.](image)

3 Software used for Control and Monitoring

A software to serve as base to the development of the control and monitoring algorithms was developed using the QT framework (Ward, 2000).

3.1 front-end Design

The front-end shown in Figure 3 makes use of the QT framework, because of that it is portable.
across multiple platforms, mainly Windows, Linux and OSX. Being open source it could be ported to virtually any platform.

The tools available in the front-end are displaying of data from the sensors in real time, displaying of the current value being sent to the car actuators and plotting data in real time from values received by the sensors in a separate window as show in Figure 12. This plotting tool allows for different plots to be on a single window at the same time. In the VILMA project this is used by first plotting a desired trajectory and then plotting the car position every 250ms. This way the algorithms controlling the vehicle can be tested and its results validated graphically.

The frequency of update of the values displayed on the screen such as position, orientation and velocity, are controlled internally by a timer which when triggered queries each sensor module, gets an updated value and updates the displayed image.

3.2 back-end

The current architecture of the back-end is highly modular and is by design easy to adapt to any project using ROS or any other middleware. For each sensor there is a module which receives data from ROS, but since each module has its own class other middlewares can be used without any modification of the program architecture. The modules communicate with each other using functions of the type get and set. Therefore the internals of each class are not important to the other modules as long as the API is kept the same.

The current modules in use in the project are:

- **GPS** Collects data from the car GPS.
- **IMU** Collects data from the car IMU.
- **MainWindow** Responsible for the main application window. Generates requests to each sensor module and updates the screen accordingly. Also receives user input and sends messages to the appropriate module.
- **MorseReceiver** Module responsible for receiving messages from the MORSE simulator using ROS as middleware. Among some of the messages are the car exact speed and position.
- **MorseTransmiter** Module responsible for sending messages to the MORSE simulator using sockets.
- **VilmaSelfDriver** Module responsible for the autonomous functions of the vehicle.

![Figure 4: Modules communication scheme.](image)

4 Control of the vehicle

This section explain the implementation of the class VilmaSelfDriver. This class controls the vehicle according to the instructions given in the user interface. It supports controlling the car speed by giving a new value or taking the current vehicle speed as input. It also allows the vehicle to follow pre-determined trajectories described by a series of waypoints. These waypoints can be entered manually in a table in the user interface or loaded from a pre-recorded text file.

4.1 Trajectory

Trajectories can be taken by recording the car position at regular time intervals. The one shown in Figure 12 was taken at a 250ms interval. Even though the user interface does not yet support an easy way to record the vehicle position, the back-end support is already in place and in a future patch there will be support for it in the user interface. However, trajectories could also be generated using maps and could contain hard turns or a small number of points.

In order to solve these problems two functions which can be accessed by the user interface were created. The first one, accessed by the
Smooth Trajectory button, takes the trajectory points from the table and tries to make the hard turns smoother. For that, the algorithm utilizes two criteria, minimizing the quadratic difference from a new point’s position to it’s older position and minimizing the distance between consecutive points. While the first condition ensures fidelity to the original points, the second one turns the trajectory into a smoother one. How strongly one condition is obeyed in detriment of the other can be changed, this way the algorithm is very versatile. The results can be compared in Figures 5 and 6. The algorithm used to smooth the trajectory

\begin{algorithm}
\textbf{Data: } X, \epsilon, \alpha, \beta
\textbf{Result: } Y
\begin{align*}
Y_0 &= X;
\text{while } EQM > \epsilon \text{ do}
\quad \text{EQM} \leftarrow 0;
\quad \text{for } i := 2 \text{ to } n - 1 \text{ do}
\quad \quad \text{tmp} \leftarrow Y^k_i
\quad \quad Y^k_i \leftarrow Y^k_{i-1} + \alpha (X - Y^k_{i-1});
\quad \quad Y^k_i \leftarrow Y^k_i + \beta (Y^k_{i+1} + Y^k_{i-1} - 2Y^k_i);\\
\quad \quad \text{EQM} \leftarrow \text{EQM} + |\text{tmp} - Y^k_i|;
\end{align*}
\text{end}
\end{algorithm}

\textbf{Algorithm 1:} Gradient algorithm to smooth the path

Where \( \epsilon \) is the tolerance used as a stopping point in the iteration, the value used in the project is 0.000001. X are the points corresponding to the path before being ran by algorithm, Y the points after being ran by the algorithm. \( \alpha \) is the term which controls the fidelity to the original points, the higher the closer the resulting points are to the original points. In the project it is set to 0.5. \( \beta \) is the term which controls the smoothness of the new trajectory, the higher the closer the points will be. In the project the value of \( \beta \) is 0.1. Also, \( n \) is the number of points of a trajectory.

Another problem that occurs is lack of points to create a proper trajectory. In order to solve this problem there is an algorithm, accessible by the user interface, which interpolates linearly between points in order to create a better defined trajectory. The result can be viewed in Figures 7, and 8.

Figure 5: Before Smoothing Algorithm.

Figure 6: After Smoothing Algorithm

is the following:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Before Smoothing Algorithm.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure6.png}
\caption{After Smoothing Algorithm}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{In red the trajectory before Increase Point Count Algorithm and in purple after the Increase Point Count Algorithm}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure8.png}
\caption{After Increase Point Count Algorithm and Smoothing Algorithm}
\end{figure}

\subsection{Path Controller}

The path controller is implemented as a function which takes as input the first point \((X,Y)\) of the trajectory and calculates the required turn angle which the front wheels require to get to that point. In order to do so, the algorithm subtracts the car coordinates from the desired coordinates, resulting in coordinates in respect to the car center, a coordinate system with the car at the origin. After this, desired car angle is obtained by calculating the tangent of the angle formed by the two new coordinates in respect to the car center. Finally, the resulting wheel angle is obtained by taking account the current car orientation and subtracting it from the desired car angle. When a point is reached it is behind the vehicle. The algorithm detects it and proceeds to the next point of the trajectory. The algorithm gets the vehicle current angle from the IMU or MORSE’s report of the vehicle’s exact orientation. The vehicle’s position is obtained from the GPS sensor or MORSE’s exact \((X,Y,Z)\) report. Checking if the point is behind the car or not is done by evaluating if the absolute value of the angle required to turn is greater than 0.6 rad. If it is, the point is behind the ve-
A proportional controller is used to control the wheel’s angle. The algorithm used is the following:

1: procedure TURN-WHEELS(X, Y, CarX, CarY, CarZAngle)
2: newX ← X − CarX
3: newY ← Y − CarY
4: ang ← atan(newX/newY)
5: ang ← ModAng(ang)
6: if ang ≥ 0.6 or ang ≤ −0.6 then
    start over with next point
end
7: return ang
8: end procedure

Figure 9: Algorithm to determine the wheel angle in order to get to a given point.

4.3 Speed Controller

The speed controller is implemented as a function which takes as input the desired speed, the car current speed and the amount of power being sent to the wheels. It utilizes a PID (Proportional, Derivative, Integrator) controller in order to control the amount of power sent to the wheels. The PID gains are dynamically adjusted based on the difference between the desired speed and the current speed (Delta) and the current wheel deflection.

<table>
<thead>
<tr>
<th>PID for less than 1 m/s of difference between target and current speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportional Gain</td>
</tr>
<tr>
<td>200[w*300]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PID for more than 1 m/s of difference between target and current speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportional Gain</td>
</tr>
<tr>
<td>200[w*300]</td>
</tr>
</tbody>
</table>

Figure 10: PID parameters

There is also a limit on the maximum allowed change of power sent to the wheels per second. This limit is imposed in order to prevent sudden accelerations and decelerations. The PID controller is predominantly proportional if the Delta is bigger than one meter per second and predominantly integral if the Delta is smaller than one. There is also a limit on the power sent to the wheels at any given time based on the desired speed. This is used to prevent large overshooting.

This approach allows a steady acceleration until the speed is not close to the target speed. When the speed is close to the target, the integral controller takes over and allows for a smoother convergence to the target speed. If at any given time the Delta switches from lower than one to bigger than one, the controller has its parameters reset and is set to mostly proportional again. This usually happens when there is disturbance such as the vehicle taking a hard turn. Resetting the parameters allows for a reliable response from the controller because it cleans the residual integral value stored from before the disturbance. The algorithm also tries to compensate for the variations of velocity when the car is turning by adding a term proportional to the current front wheel angle in the PID’s parameters.

4.4 Results

The speed controller performance can be seen in Figure 11. The vehicle currently takes about 15 seconds to get to the desired speed, but the acceleration can be increased or lowered by changing the proportional PID parameter. The three graphs are very close in form to one another, but it can be noticed that as the desired speed increases, the lower the overshoot is. This happens because the maximum power sent to the wheels is implemented to scale linearly with the desired speed. However, the resistance faced by the vehicle to accelerate scales quadratically. In a future development this issue will be addressed and when the algorithm is implemented in the real car, experimental data will be used to determine the maximum output to a given desired speed.

Figure 11: PID control performance for target speed of 5[m/s], 10[m/s] and 15[m/s] in a straight line.

The results using the path controller can be seen in Figure 12 which shows the planned path and the actual path.

The results of the path controller are shown in Figure 12. For a reasonably high density of points the algorithm provides very good results with error in the order of centimeters even when complicated trajectories previously recorded are used. Although the algorithm provides good results, it has its drawbacks, such as not taking into account the wheel turn speed and modifying the angle of the wheels at each point used in the trajectory leading to continuous small changes which can cause discomfort and mechanical components can suffer fatigue. One solution is to increase the number of points of the trajectory until the changes in wheel angle become so small that it
seems as it is a continuous transition. The problem with this solution is that the algorithm must be run at a high frequency in order to keep up with the large number of points and also the car position information must also be updated at the same rate or higher. Which means the position estimation algorithms would also need to run at a high frequency.

5 CONCLUSION

Because of the modular architecture used in this project it can be easily adapted to other projects. Having MORSE as a module makes it possible to exchange the simulator for the real car without big changes. Since the path controller makes no assumptions about the car it is expected to work properly with all ground vehicles moving in a plane. The speed controller is based on PID controls with variable parameters, so other projects can reuse the code only changing the parameters in order to match the different vehicle dynamics. MORSE supports many vehicles, including flying vehicles and aquatic. This way, MORSE and all the work reported here can be a starting point to other projects.

Perception using vision to implement a local path planning is the next step in the project. This way the vehicle will be able to plan its own paths knowing what happening in the environment. Also planned are refinements to the current path following and speed control algorithms. All the work done so far is hosted publicly at www.github.com/tiberiusferreira/VilmaProject.

ACKNOWLEDGMENT

This work has been cofounded by CAPES and by CNPq Brazil. Part of the code used in MORSE simulator is based in CaRINA PROJECT of the Laboratory Mobile Robotics from USP/SC.

References


