Remote Controlled Robot using ROS over LTE

Robert Hedman* , Samuel Karlsson†, Oscar Sandström‡, and Olov Sehlin§
Luleå University of Technology
*Automatic control and embedded systems
Email: robert.hedman@mac.com
†Software Engineer
Email: samkar-4@student.ltu.se
‡Software Engineer
Email: oscsan-4@student.ltu.se
§Automatic Control and Simulations
Email: oloseh-4@student.ltu.se

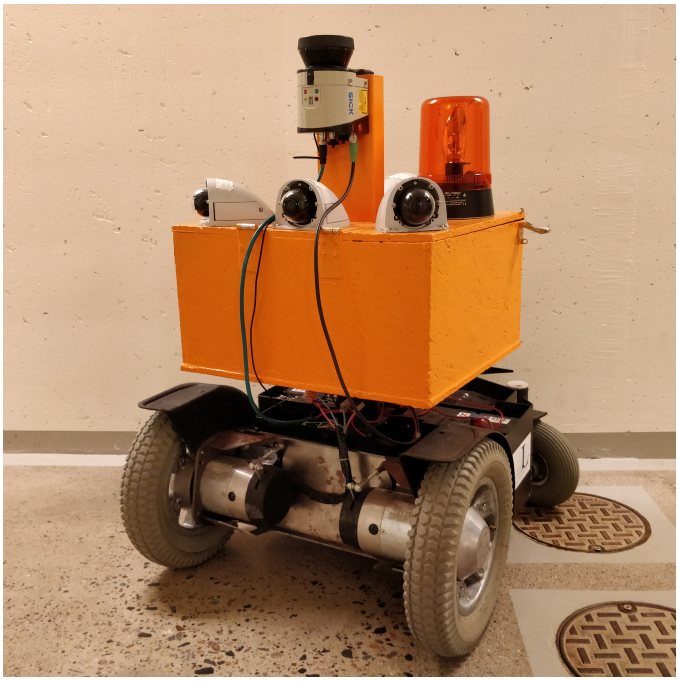

Fig. 1: A photo of the finished Permocar.

***Abstract*— Limiting human presence in non-human friendly environments is the primary objective of this work. By developing technology for real time remote control is it possible to quickly remove humans from many dangerous sites, such as mines.**

The Permocar, depicted in figure (1), is based on a wheelchair and can be remotely operated over LTE using ROS. Video feedback is provided from four cameras mounted on top of the vehicle and a scan of the surroundings is supplied from a LIDAR. The video data is rate adapted to limit latency using SCReAM.

Slight autonomy has been implemented; for example collision detection using the LIDAR data. A theoretical physical model of The Permocar has been derived, including models of the motors. An extended Kalman filter was used to estimate the immeasurable states providing a full state estimate to the feedback linearization controller. The control system derived from the model did not work due to model errors/parameter insecurities. System identification methods were used to model the motors. Due to the nature of the motors neither parameters nor output could be measured so blind identification was used. Probable ranges for the motor parameters were found but an accurate model was never found causing the control system to not work as expected.

ROS has made implementation, and structuring, of code immensely simplified. The publish/subscribe framework makes expanding the platforms functionality very straightforward and modular.

Hector_slam was implemented for Simultaneous Localization and Mapping using only the LIDAR data. While working well further tuning might improve performance, especially in low feature areas such as the basement of the university corridors.

Adding an Inertial Measurment Unit for positioning together with sensor fusion of the LIDAR data and GPS might allow for fully autonomous navigation with a Model Predictive Control for Path-Planning. MPCs are very easy to implement for linear systems, hence our approach with feedback linearization.

Adding custom ROS message types for some of the messages our nodes use would increase readability of our code.

***Keywords*— Remote control, ROS, Differential drive robot, SCReAM, SLAM, Feedback linearization**

I. Introduction

In harsh environments such as mine shafts and other similar environments where it is dangerous for humans to be, it is preferable if human could be removed. Peoples security is of great concern, the risks of working under ground is severe, so the improvement of remote controlled mining equipment is of great importance. A big problem with remote controlled vehicles are visual feedback in real time and delay in the system [3].

The Permocar is a remote controlled vehicle built on an electric wheelchair. It is a small platform to test systems on before installing it on big machines (that are expensive in comparison). The Permocar utilizes LIDAR, GPS, encoders and four cameras, for navigational feedback which helps with driving. It is driven using a gamepad (under development, a wired Xbox 360 controller is currently in use) over internet and a LTE through a VPN server.

All code for this project can be found on our git [1].

II. Background

This project started about one year ago when Robert was working at Ericsson. Robert and Olov started to develop a mobile remote controlled platform as a fun project without any real connection to their course plan at LTU. Now it will be remade in a more focused way with more structure and documentation. The main goal is to create a well engineered test platform, where it is easy to add new component for testing. To complete this task we decided to use ROS.

A. Permocar

The foundation of this project is the Permocar. It is the vehicle upon which all testing and development is performed.

1) Component Breakdown: The Permocar is based on a wheelchair base with two separately driven electric motors with wheels in front and two freely rotating and spinning wheels in the back: a differential drive robot. On top of the chassis is a weatherproof orange box containing all electronics. On-board computers include a Raspberry Pi, which is the central on-board computer, along with three Arduino circuit boards; one for motor drive control, one for encoder communication, and one for relay operation. The Permocar has several sensors and cameras. On top of the orange box four cameras are mounted, three facing the front and one facing the rear. A lidar of model SICK LMS111-10100 is mounted on the top along with a GPS antenna. On the rear wall of the box an LTE antenna is mounted. A warning light is also mounted on top of the box along with an emergency stop button. A switch (with POE capabilities) along with an LTE router handling all the networking on-board and externally. All parts connected to the network and camers were lent for Ericson, the permocar chassis from the CEG at LTU and the rest from XP-R. All hardware components are listed in the hardware diagram in figure (10). More information such as brand, model and quantity are listed there as well.

2) Communication: Connecting to the internet through an LTE modem means connecting through Network Address Translation, NAT. A remote user connection generally does not offer a static IP address. Having to change parameters in code to adapt every installation/setup is impractical. Therefore a layer three virtual private network, VPN server is set up in a server hall with a known static IP, providing a means of common communication ground for all project components. The subnet is 10.9.0.0/24 where the server has 10.9.0.1.

B. Robotic Operating System

Robotic Operating System (ROS) is a framework design to make code for robots simple to develop and reuse. It is achieved by standardized internal communication.

1) Messages: The communication is done with a message system. There are many types of messages in ROS and it is possible to create custom message types with the desired content. Messages are sent between nodes using TCP protocol. As a programmer there is no need to know how messages are passed or how they find their receivers. The messages are published on a topic which is what the programmer uses to connect nodes. The only thing a programmer needs to know about the communication in ROS is the message type and the topic, where the topic is an arbitrary name chosen by the programmer.

2) Node: ROS is built up with nodes: a module of code that can run by itself. ROS, as any other system, has its own naming convention. When it comes to nodes they consist of two words that are separated with a underline: _. A node communicates with other nodes by sending and receiving messages through topics but it does not have any knowledge of the other nodes or how many there are. The nodes connect to roscore, the main organizer of the ROS system, where they receive the communication handles to other nodes. Roscore is the only node that keep track of all the nodes in the system. Nodes can be located on any computer, anywhere in the world as long as it has a connection to roscore (on the same sub-network).

There are four main types of nodes: publisher, subscriber, server and client. A node can be one or many types at once (a node can both be a publisher and subscriber). A publisher node is publishing a message to its advertised topic. The subscribers subscribe to a topic, when a message is published on that topic the corresponding callback function is triggered for the subscriber. A client sends a request to a server, that triggers the server to return a message synchronous [10].

C. SCReAM

Varying latency and bandwidth in any network, especially over cellular networks, is a major issue for real-time video transfer. In time critical systems using large buffers to even out varying bandwidth is not an option since it will introduce delay. Keeping the video quality, specifically the bit rate, low may work well but drastically limits the maximum possible quality attainable.

SCReAM (Self Clocked RatE Adaption for Multimedia) is a software developed by Ericsson to solve these issues [2]. By monitoring network latency the bit rate is constantly adjusted to allow for maximum quality while never exceeding a set transmission latency. If the video data stream contains more data per time than the network can handle there will be buildup in buffers along the network path; introducing delays.

The code was provided for this project to be implemented on the top mounted cameras.

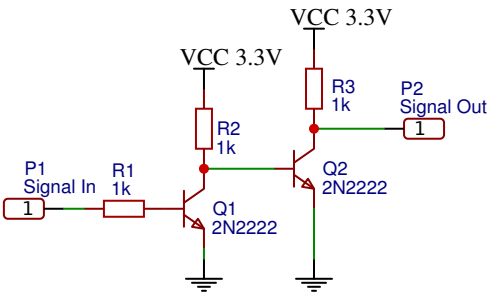

Fig. 2: A one channel logic level conversion circuit for the encoders.

III. Sensors

Several sensors are installed and used for smoother driving and future autonomous driving.

A. Encoders

Encoders are mounted in the motors on the Permocar. There is no documentation or other information on them. The lack of documentation meant that we had to reverse engineer the system. Each motor has two thick cables and five small cables shown in figure (11). The thick cables very likely provide power to the motors. The other five are then probably for the encoders. Encoders in general have two cables for power supply to the circuitry and two cables carrying some form of signal indicating speed and direction of the motor. By trial, error, and measurements the following is true:

- The blue cable carries a PWM signal with a frequency proportional to the speed of the wheel, ~3-17KHz.
- The white cable also carries a PWM signal, identical to the blue but phase shifted ± 90 degrees depending on the direction of the motor.
- The green cable is the relative ground for the signals.
- The black cable is for ground.
- The red cable is for voltage supply. The supplied voltage directly scales the max and min of the PWM signals within the range ~3-4V to 24V (higher seems possible but we did not dare test it).
- At 5V supply the current draw is close to 20mA at all times.

We connected the black and green cable to common ground for simplicity and decided to feed the circuit 5V since 5V is available from the Arduino interfacing the encoders. The encoder frequency is so high that a standard Arduino UNO is too slow so we used an Arduino M0 pro, equipped with an arm processor, instead. The M0 pro, however, cannot handle logic signals of 5V so we built a small logic level conversion circuit for the 4 channels with discrete components, which can be seen in figure (2).

B. GPS

The on-board GPS is an Adafruit Ultimate GPS Breakout v3. To connect the GPS board to the Raspberry Pi a standard USB-USART converter was used.

C. Lidar

The lidar is a SICK LMS111-10100 [17]. It primarily uses an ethernet IP/TCP interface to provide scan data to any requester.

1) Lidar issues: During development our lidar had a setting get stuck in memory. The n-mean-filter, averaging n scans to provide a smoother result when standing still, got stuck permanently turned on. The only way to remove this setting was a complete factory reset. The scripts factory re-setting and re-setting the lidar can be found on our git.

D. Cameras

Four IP cameras, Panasonic, are mounted on top of the orange box. One facing forward, one backwards and two facing mostly forward but angled to the sides. These provide real time HD video streams over IP with RTP (Real-time Transport Protocol).

IV. Mathematical model

When modeling a system, different methods can be used. Two commonly used and known methods are the Newton-Euler and Lagrange method. The Newton-Euler method derives the equation of motion using forces, while the Lagrange method using energy instead. [5] [6] Here, the Lagrange method was used to get more practice in this method.

To express the position and movement of the robot, two different coordinate systems have to be defined. First the inertial coordinate system: The position and movement from a fixed frame, viewing the robot from a distance. It will be called the inertial frame. The other coordinate system is the robot coordinate system: The position and movement of the robot from its own point of view and it is fixed to the robot. It will be called the robot frame. These two coordinate systems are shown in figure (3).

The transformation between the two coordinate systems can be described by the orthogonal rotation matrix:

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1)$$

Such that:

$$X^I = R(\theta)X^r \quad (2)$$

Where X^I and X^r are the coordinates of the inertial frame and robot frame, respectively.

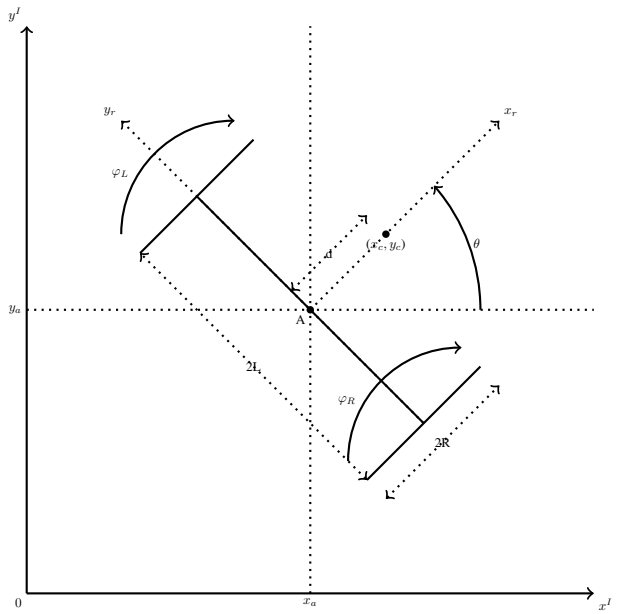

Fig. 3: The differential drive robot and the both coordinate frames.

A. Generalized coordinates

The velocity and angular velocity in the robot frame can both be expressed as a function of the individual angular velocity of the wheels as follows:

$$\begin{aligned} v &= \frac{v_R + v_L}{2} = \frac{R(\dot{\varphi}_R + \dot{\varphi}_L)}{2} \\ \omega &= \dot{\theta} = \frac{v_R - v_L}{2L} = \frac{R(\dot{\varphi}_R - \dot{\varphi}_L)}{2L} \end{aligned} \tag{3}$$

and from the inertial frame:

$$\begin{aligned} \dot{x}_a &= v sin(\theta) \\ \dot{y}_a &= v cos(\theta) \\ \dot{\theta} &= \dot{\theta} \end{aligned} \tag{4}$$

By combining equation (3) and (4), and a coordinate extension. The velocities in the inertial frame can be expressed as follows:

$$\dot{q} = \begin{bmatrix} \dot{x}_a \\ \dot{y}_a \\ \dot{\theta} \\ \dot{\varphi}_R \\ \dot{\varphi}_L \end{bmatrix} = \frac{1}{2} \begin{bmatrix} Rcos(\theta) & Rcos(\theta) \\ Rsin(\theta) & Rsin(\theta) \\ \frac{R}{L} & -\frac{R}{L} \\ 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} \dot{\varphi}_R \\ \dot{\varphi}_L \end{bmatrix} = S(q)\eta \tag{5}$$

where q will be used as the generalized coordinates. These are the minimum independent coordinates that describes the system and simplifies the derivation of the equation of motion using Lagrange method. [7]

B. Kinematic constraints

In order to derive a model for the differential drive robot, some assumptions has to be made. First, no sideways slip motion. This means that the robot can move only forward and backwards, not sideways. From the robot frame it can be described by:

$$\dot{y_a^r} = 0 \tag{6}$$

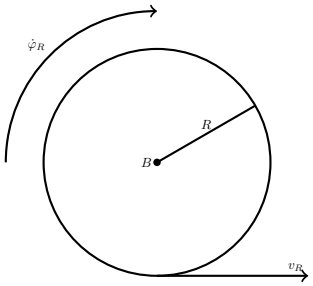

Fig. 4: The right wheel and the pure rolling constraint.

From the inertial frame, transformed using the transformation matrix $R(\theta)$:

$$-sin(\theta)\dot{x}_a + cos(\theta)\dot{y}_a = 0 \tag{7}$$

The other assumption is pure rolling. Shown in figure (4) This means that the wheels always has one contact point with the ground. No slipping along the robots frame x-axis, nor the y-axis. From the robot frame, the wheel velocities can be described by:

$$\begin{aligned} v_R &= R\dot{\varphi}_R \\ v_L &= R\dot{\varphi}_L \end{aligned} \tag{8}$$

From the center point A in the inertial frame, the velocities can be described by:

$$\begin{aligned} \dot{x}_{wL} &= \dot{x}_a - L\dot{\theta}cos(\theta) \quad & \dot{x}_{wR} &= \dot{x}_a + L\dot{\theta}cos(\theta) \\ \dot{y}_{wL} &= \dot{y}_a - L\dot{\theta}sin(\theta) \quad & \dot{y}_{wR} &= \dot{y}_a + L\dot{\theta}sin(\theta) \end{aligned} \tag{9}$$

By using the transformation matrix $R(\theta)$, equation (9) can be written as:

$$\begin{aligned} \dot{x}_{wR}cos(\theta) + \dot{y}_{wR}cos(\theta) &= R\dot{\varphi}_R \\ \dot{x}_{wL}cos(\theta) + \dot{y}_{wL}cos(\theta) &= R\dot{\varphi}_L \end{aligned} \tag{10}$$

Equation (9) and (10) can be combined and the pure rolling constraint can be written on the final form from its inertial frame:

$$\begin{aligned} \dot{x}_a cos(\theta) + \dot{y}_a sin(\theta) + L\dot{\theta} - R\dot{\varphi}_R = 0 \\ \dot{x}_a cos(\theta) + \dot{y}_a sin(\theta) - L\dot{\theta} - R\dot{\varphi}_L = 0 \end{aligned} \tag{11}$$

The two constraint equations, (7) and (11), can now be written on the final matrix form:

$$\Lambda(q) = \begin{bmatrix} -sin(\theta) & cos(\theta) & 0 & 0 & 0 \\ cos(\theta) & sin(\theta) & L & -R & 0 \\ cos(\theta) & sin(\theta) & -L & 0 & -R \end{bmatrix} \tag{12}$$

C. Lagrange dynamics

This system only has motion in two dimensions, x and y, and thus only kinetic energy. This simplifies the Lagrangian function to $L = T$, where T is the kinetic energy. The Lagrange equation with constraints is defined as follows:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = F - \Lambda^T(q)\lambda \tag{13}$$

F is the generalized force vector and λ is the Lagrange multipliers connected with the constraints.

The kinetic energy T of the robot can be divided into two parts, $T = T_c + (T_{wR} + T_{wL})$. The kinetic energy for the robot:

$$T_c = \frac{1}{2} m_c v_c^2 + \frac{1}{2} I_c \dot{\theta}^2 \tag{14}$$

The kinetic energy for the both wheels:

$$\begin{aligned} T_{wR} &= \frac{1}{2} m_w v_{wR}^2 + \frac{1}{2} I_m \dot{\theta}^2 + \frac{1}{2} I_w \dot{\varphi}_{wR}^2 \\ T_{wL} &= \frac{1}{2} m_w v_{wL}^2 + \frac{1}{2} I_m \dot{\theta}^2 + \frac{1}{2} I_w \dot{\varphi}_{wL}^2 \end{aligned} \tag{15}$$

Where m_c, is the mass of the robot without the wheels and m_w, is the mass of the wheel. I_c is the moment of inertia for the center of mass, rotation around its rotational axix, point A in figure (3). I_w is the moment of inertia for each wheel rotation around its own rotational axis, point B in figure (4) and I_m is the moment of inertia for each wheel rotation around point A.

Now, the energies are expressed from the robot frame. All velocities can be described from the inertial frame by:

$$v_i^2 = \dot{x}_i^2 + \dot{y}_i^2 \tag{16}$$

The position of the center of mass expressed from the rotational center point:

$$\begin{aligned} x_c &= x_a + dcos\theta \\ y_c &= y_a + dsin\theta \end{aligned} \tag{17}$$

The same thing can be made for the wheel position. Differentiating equation (17) will give the velocities for all parts and together with equation (14) and (15) the final kinetic energy and Lagrange function became:

$$\begin{aligned} L = T = \frac{1}{2} m \left(\dot{x}_a^2 + \dot{y}_a^2 \right) + \frac{1}{2} I \dot{\theta}^2 + \frac{1}{2} i_w \left(\dot{\varphi}_R^2 + \dot{\varphi}_L^2 \right) \\ + m_c d \dot{\theta} \left(\dot{y}_a cos(\theta) + \dot{x}_a sin(\theta) \right) \end{aligned} \tag{18}$$

Where, $m = m_c + 2m_w$ and $I = m_c d^2 + I_c + 2m_w L^2 + 2I_m$ to simplify the expression.

Now the Lagrange equation (13) can be solved. The equations of motion:

$$\begin{aligned} m\ddot{x}_a - m_c \ddot{\theta} dsin(\theta) - m_c \dot{\theta}^2 dcos(\theta) &= \Lambda_1 \\ m\ddot{y}_a + m_c \ddot{\theta} dcos(\theta) - m_c \dot{\theta}^2 dsin(\theta) &= \Lambda_2 \\ I\ddot{\theta} + m_c d\ddot{y}_a sin(\theta) - m_c d\ddot{x}_a sin(\theta) &= \Lambda_3 \\ I_w \ddot{\varphi}_R &= \tau_R + \Lambda_4 \\ I_w \ddot{\varphi}_L &= \tau_L + \Lambda_4 \end{aligned} \tag{19}$$

Where Λ_i is connected to the kinematic constraints from equation (12). Now, the system with generalized coordinates and constraints can be written on the following matrix form:

$$M(q)\ddot{q} + V(q, \dot{q})\dot{q} = B(q)\tau - \Lambda^T(q)\lambda \tag{20}$$

Where M is the inertia matrix, V is the centripetal and coriolis matrix and B is the input matrix:

$$\begin{aligned} M(q) &= \begin{bmatrix} m & 0 & -m_c dsin(\theta) & 0 & 0 \\ 0 & m & m_c dcos(\theta) & 0 & 0 \\ -m_c dsin(\theta) & m_c dcos(\theta) & I & 0 & 0 \\ 0 & 0 & 0 & I_w & 0 \\ 0 & 0 & 0 & 0 & I_w \end{bmatrix} \\ V(q, \dot{q}) &= \begin{bmatrix} 0 & 0 & -m_c d\dot{\theta})cos(\theta) & 0 & 0 \\ 0 & 0 & -m_c d\dot{\theta})sin(\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\ B(q) &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{21}$$

In equation (5), we saw that the generalized coordinates can be written as: $\dot{q} = S(q)\eta$. It can be shown that $S(q)$ is the null space of $\Lambda(q)$:

$$S(q)\Lambda(q) = 0 \tag{22}$$

By multiplying equation (20) with $S(q)$ to the left and replace $\dot{q}$ with $S(q)\eta$, the unknown Lagrange multipliers disappears and new $\bar{M}(q)$, $\bar{V}(q, \dot{q})$ and $\bar{B}(q)$ can be formed:

$$\begin{aligned} \bar{M}(q) &= \begin{bmatrix} I_w + \frac{R^2}{4L^2}\left(L^2 m + I\right) & \frac{R^2}{L^2}\left(L^2 m + I\right) \\ \frac{R^2}{L^2}\left(L^2 m + I\right) & I_w + \frac{R^2}{4L^2}\left(L^2 m + I\right) \end{bmatrix} \\ \bar{V}(q, \dot{q}) &= \begin{bmatrix} 0 & \frac{R^2}{2L} m_c d\dot{\theta} \\ -\frac{R^2}{2L} m_c d\dot{\theta} & 0 \end{bmatrix} \\ \bar{B}(q) &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{23}$$

The final step is to write the equation of motion using the linear velocity v and angular velocity ω from the robot frame with the torques τ_R and τ_L as inputs. This can be solved by combining equation (3) and (23) and results in the final form of the equation of motion:

$$\begin{aligned} \left(m + \frac{2I_w}{R^2} \right) \dot{v} + m_c d\omega^2 &= \frac{1}{R} \left(\tau_R + \tau_L \right) \\ \left(I + \frac{2L^2 I_w}{R^2} \right) \dot{\omega} + m_c dv\omega &= \frac{L}{R} \left(\tau_R - \tau_L \right) \end{aligned} \tag{24}$$

D. The motors

The permocar base used in this project is old, well used and not well documented. The motors has some information on them, like some ratings for current and voltage but nothing about the actual mechanics or parameters. Therefore, system identification methods has to be used for modelling them.

Since no information about the motors was available, the most common model for a DC motor was assumed:

$$\Phi = R_i i + L \frac{\partial i}{\partial t} + K_\omega \omega \tag{25}$$

where ω is the rotational speed of the motor, i the electrical current, Φ the voltage across the motor, R is the resistance and L the inductance. Torque, τ is proportional to the current:

$$\tau = K_\tau i \tag{26}$$

where K_ω and K_τ are motor coefficients. In discrete time this model is equivalent to the following:

$$\tau_{k+1} = k_1\tau_k + k_2\omega_k + b_1\Phi_k \tag{27}$$

And with piece-wise constant inputs this model is equivalent to the continuous model:

$$\dot{\tau} = K_1\tau + K_2\omega + B_1\Phi \tag{28}$$

which can be obtained from the discrete time model with the Matlab command `d2c`.

We cannot measure the torque or the current, but we can measure the rotational speed and the voltage. By running the motors with a piece-wise constant input of random time and amplitude for several minutes while recording its speed, blind identification can be used to identify suitable engine model parameters. This means finding an initial guess of the parameters and then using a nonlinear programming solver; `fmincon`, to solve for the best fitting torque vector in time with those parameters. With the best fitting torque vector we then reverse the problem and use the torque vector to solve for the optimal parameters. This process is iterated until the solution converges to a minimum. In this case the convergence rate is very slow. For a modern computer it takes several weeks for a few hundred thousand iterations, which is still not enough for full convergence. A good guess might be enough since we have the simulator to match up with later.

E. Complete Model

With both the dynamics of the system derived and the motor model the complete non linear state space system can be put together.

$$\begin{aligned} \dot{x} &= f(x) + g(x)u \\ y &= h(x) \end{aligned} \tag{29}$$

Where:

$$\begin{aligned} x &= \begin{bmatrix} v \\ \omega \\ \tau_R \\ \tau_L \end{bmatrix} \qquad u = \begin{bmatrix} \Phi_R \\ \Phi_L \end{bmatrix} \\ f(x) &= \begin{bmatrix} A_1\omega^2 + A_2\tau_R + A_3\tau_L \\ A_4\omega v + A_5\tau_L + A_6\tau_R \\ K_1\tau_R + K_2(v + L\omega) \\ K_3\tau_L + K_4(v - L\omega) \end{bmatrix} \\ g(x) &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ B_1 & 0 \\ 0 & B_2 \end{bmatrix} \\ h(x) &= \begin{bmatrix} v \\ \omega \end{bmatrix} \end{aligned} \tag{30}$$

The problem is that there are many large uncertainties in model parameters for every part in the system (except perhaps the total weight). The plan was to take the parameter estimates, try them on the real system and then change parameters in the simulation to match the discrepancies observed in the real system. Once a match is found it is easy to change the parameter values the other way and make the real system behave as planned. I.e. match the simulation parameters to the real system and then use the updated control scheme dependent on the actual parameters.

V. Control theory

It is easy to seen that the system, equation (30), is non-linear. It has both square terms and cross terms of the states. Also, no clear working point can be used for linearization because the robot should be able to handle its whole range of speed, therefore feedback linearization will be a perfect first approach.

A. Feedback linearization

Feedback linearization is a method commonly used for control of non-linear systems. By deriving the systems output until the input is found, a control law can be constructed that exactly cancels the non-linearities making the closed loop system respond according to chosen poles. This is made from changes in variables and choosing the control law:

$$u = \frac{1}{\beta(\xi,\eta)}\left(-L\xi + L_r r - \alpha(\xi,\eta)\right) \tag{31}$$

Where α and β are found by deriving the output y until the input ϕ are found.

$$\begin{aligned} \xi_1 &= y \\ \xi_2 &= \dot{\xi_1} = \frac{\delta h}{\delta x}\dot{x} = \frac{\delta h}{\delta x}\left(f(x) + g(x)u\right) \\ \xi_3 &= \dot{\xi_2} = \dots \\ \xi_p &= \alpha(\xi,\eta) + \beta(\xi,\eta) \end{aligned} \tag{32}$$

Where ξ, are the new state variables, p is the relative degree of the system and η is related to the zero dynamics. L and L_r will decide the poles and zeros of the system, respectively [8].

Here, the input was found after deriving the output twice, giving a relative degree of two and no zero dynamics. The derivation was made in *MATLAB* using the jacobian. It can easily be shown that β is invertible which is a condition for feedback linearization to work.

After deriving the output twice the new state vector was found:

$$\xi = \begin{bmatrix} \xi_1, \xi_2 \end{bmatrix}^T = \begin{bmatrix} v, w, \dot{v}, \dot{w} \end{bmatrix}^T \tag{33}$$

Reordering the state vector makes designing the control law matrices L and L_r easier:

$$\xi = \begin{bmatrix} v, \dot{v}, w, \dot{w} \end{bmatrix}^T \tag{34}$$

The new linear space space model is found by combining equation (31) and (32):

$$\begin{aligned} \dot{\xi} &= (A - BL)\xi + BL_r r \\ y &= C\xi \end{aligned} \tag{35}$$

where A is a integration matrix and B and C are the new state space matrices for inputs and outputs, respectively:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{36}$$

L and L_r will decide the poles and zeros for the new linear system with the following structure.

$$L = \begin{bmatrix} p11 & p12 & 0 & 0 \\ 0 & 0 & p23 & p24 \end{bmatrix}, L_r = \begin{bmatrix} p11 & 0 \\ & p23 \end{bmatrix} \quad (37)$$

With these matrices, equation (36) and (37), the transfer functions from reference to the outputs become:

$$TF = \begin{bmatrix} \frac{p11}{s^2+p12s+p11} & 0 \\ 0 & \frac{p23}{s^2+p24s+p23} \end{bmatrix}, \quad (38)$$

This means no cross coupling in the controller. The reference of the velocity will only affect the velocity and the same for the angular velocity.

B. Estimator

As mentioned in section IV, the torque can not be measured. This is a problem because it is needed for the full state feedback in the feedback linearization. It can be solved by having a estimator for the torques. First a Luenberger observer was tested in simulations but because of the non linearities and no specific working point, it did not work well. A non linear approach is needed to get good performance. Therefore an extended Kalman filter is a suitable estimator.

The extended Kalman filter algorithm is as follows [9]:

1). Set the initial conditions.

$$k = 0, \quad \hat{x}(0|0) = x_0, \quad P(0|0) = P_0 \quad (39)$$

2). Linearization at the current state.

In every loop cycle, the system need to be linearized at the current state. In Matlab the *jacobian* function differentiates our system at the current state and returns a continuous linear state space model:

$$\begin{aligned} Ac_k &= \frac{\partial f_k}{\partial x}\Big|_{\hat{x}(k|k),u(k)}, \quad Bc_k = \frac{\partial f_k}{\partial u}\Big|_{\hat{x}(k|k),u(k)} \\ Cc_k &= \frac{\partial g_k}{\partial x}\Big|_{\hat{x}(k|k),u(k)}, \quad Dc_k = \frac{\partial g_k}{\partial u}\Big|_{\hat{x}(k|k),u(k)} \end{aligned} \quad (40)$$

The Kalman filter needs the linear discrete model:

$$\begin{aligned} Ad_k &= I + TsAc_k \\ Bd_k &= TsBc_k \\ Cd_k &= Cc_k \\ Dd_k &= Dc_k \end{aligned} \quad (41)$$

Where I is the identity matrix and Ts is the sampling time.

3). Prediction.

$$\begin{aligned} \hat{x}(k+1|k) &= A_k\hat{x}(k|k) + B_ku(k) \\ P(k+1|k) &= A_kP(k|k)A^Tk + N_kQ_k{N_k}^T \end{aligned} \quad (42)$$

4). Update.

$$\begin{aligned} &k = k+1 \\ &K_k = P(k|k-1){C_k}^T(C_kP(k|k-1){C_k}^T + R_k)^{-1} \\ &\tilde{y}(k) = y(k) - C_k\hat{x}(k|k-1) \\ &\hat{x}(k|k) = \hat{x}(k|k-1) + K_k\tilde{y}(k) \\ &P(k|k) = P(k|k-1) - K_kC_kP(k|k-1) \end{aligned} \quad (43)$$

The N_k, R_k and Q_k matrices was set to give similar weights on to the model and the measurements. It was designed this way because of uncertainties in the model but also low resolution on the encoders at low speeds. The numerical values was set to:

$$N_k = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}, R_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, Q_k = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix}$$

VI. Simulations

All simulations for testing the control schemes and estimators was made in Simulink. The different blocks for the model, controller and estimator were all implemented in Matlab function blocks to make it easier to translate it to C++ and ROS [15].

To test the performance on the feedback linearization and Kalman filter steps of $0.5\ m/s$ and $0.5\ rad/s$ were set as reference signals one at the time. The velocity step are shown in figure (5) and the estimated torques are shown in figure (6)

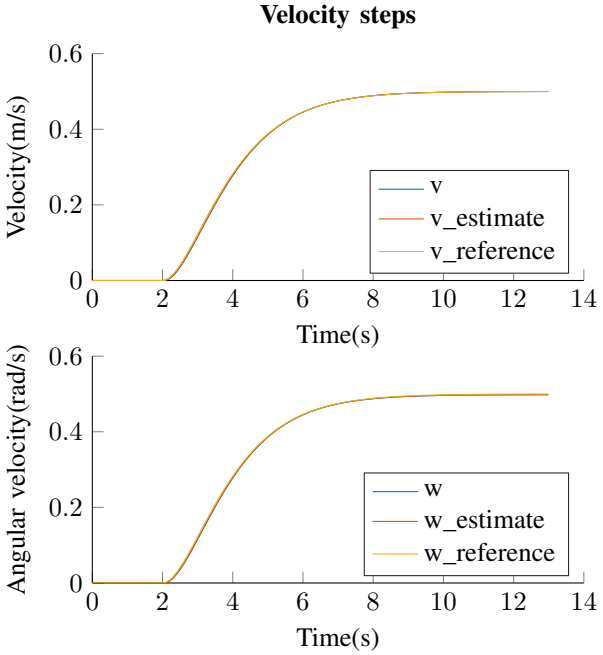

Fig. 5: A step from 0 to 0.5 in linear velocity/angular velocity showing the simulated, estimated and the reference velocities.

The poles of the two transfer function was set to the same and have a rise time of 4 seconds without over shooting.

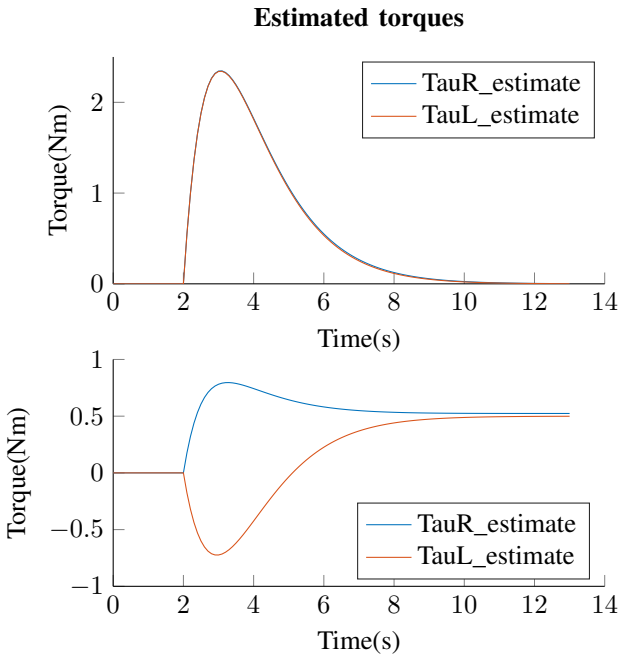

Fig. 6: The estimated torques from a step from 0 to 0.5 in linear velocity/angular velocity.

A. Simulation errors

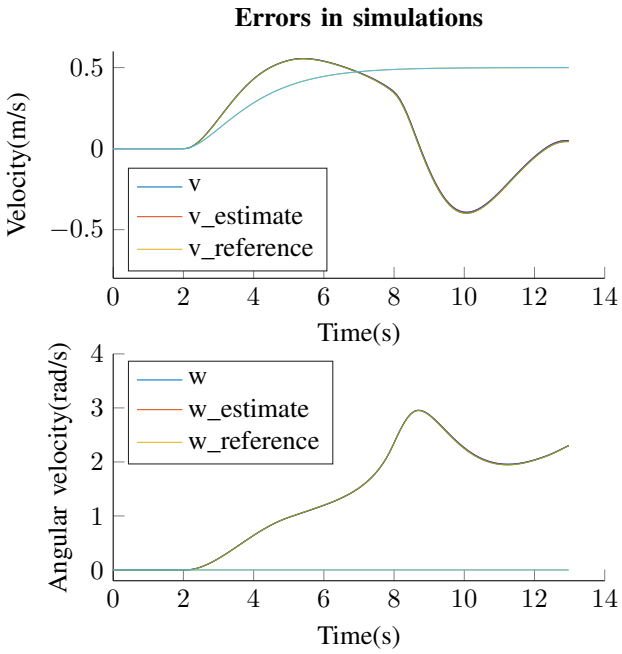

Fig. 7: A step from 0 to 0.5 in linear velocity with added erros in the model.

As mentioned in section IV, there are many uncertainties in the model, therefore a model based controller will not work as expected. This can be illustrated by changing the model parameters in simulation system while keeping the original model for the controller. Decreasing K1 and K2 by 30 percent and increasing K3, K4, B1 and B2 by 40 percent the simulations where completely of. This is shown in figure (7). These parameters were all found by the blind identification.

VII. SLAM

One of the biggest challenges for robots today is localization outside of a controlled environment. For generic cases there are some methods to estimate the position, depending on the application it might be good enough. The Permocar is strictly speaking not a robot since it isn't autonomous; it is a remote controlled vehicle. To get automated driving working it's essential to have consistently good localization.

Getting a map of the surrounding environment can be a useful tool for localization, surveillance or movement analysis. One method to localize and map is simultaneous localization and mapping (SLAM). There are many approaches to SLAM. The approach used in the Permocar is Hector_slam (see section VIII-E). To localize and map simultaneously using a lidar as the only sensor is a chicken and egg problem. You localize based on the map and you create the map based on your position but doing them simultaneously is hard.

A. Localization

In the Permocar there are three sensors installed that can aid in localization. The first is a GPS, it has some severe limitations. A GPS only works outside where it has line of sight to the satellites. Tall structures in its proximity can result in severe errors. Even in the best case scenario the accuracy is not good enough for automation or mapping. But it will keep the position in the ball park, for imprecise navigation over long distances and long time, thanks to its zero drift.

In the Permocar there are also encoders installed. Encoders will have a significant drift so their error will grow over time, but for short distances encoders are a valid option.

The third sensor is the lidar. Using lidar data is a computationally heavy task in comparison to the other sensors. It identifies features, such as walls and corners. With previous scans and those features the lidar tries to match the features to the previous scans and moves the center of the scan (the center of the scan is assumed to be the position) to depict its new position. In an ideal case this is easy, but reality is not that cooperative. In reality there is noise in the measurements and the scan is not instant so the Permocar might move during the scan. This makes it hard to be sure of the position; it can only give an approximation of the position.

More sensors are required to get a position approximation that is close enough to reality, for example an inertial measurement unit (IMU). With more sensors is it possible to perform sensor fusion to estimate the odometry. A Kalman filter can fuse sensor data to get a good estimate of the actual position. With good knowledge of the noise and accuracy

of the various sensors the accuracy of the position estimate increases. If the noises are zero mean and Gaussian a Kalman filter is the optimal estimator. [11].

B. Mapping

Mapping is heavily dependent on localization. If the position is known it's relativity easy to create a map of the surrounding using a lidar (or stereo camera). Even with a known position there is some challenges remaining such as: Is the environment constant or could it change, are there people moving around? If such conditions are known, simplification and optimization can be done in the mapping algorithm. These optimization also improve the mapping if there are uncertainties in the position. For the Permocar no conditions are assumed to be known, so none of the known simplification are applied.

VIII. ROS structure

The structure of this project is comprised of several nodes that send and receive messages from each other creating a segmented system, although one easy to understand. The entire system with the nodes and topics was put in a figure with the ROS GUI plugin "rqt_graph" and can be seen in figure (9).

The following subsections are named after the nodes and will describe each node.

A. Coll_detect

Coll_detect is a node that subscribes to "scan", from the lidar, and "wheel_velocity" then analyses the data to detect impending collision. If a collision is imminent coll_detect will publish "true" on the topic "lidar_stop" (messages type: "std_msgs/Bool") if there is no imminent collision "false" is published.

B. Control_panel

The Control_panel is a HMI (Human Machine Interface) between the driver and the Permocar. A picture of the Control_panel can be seen in figure (12). The Control_panel was created with Pygame, a library using Python, which leads this node to be written in Python instead of C++. There are no prevalent problems using both C++ and python nodes in the same system.

The Control_panel shows the current reference values for the Permocar's velocity, angular velocity, impeding collision and relay statuses. The control_panel does all this by subscribing to "joy"(a topic using "sensor_message" with a float array and int array), "lidar_stop", "referens" and "vw_estimate". Buttons that have a toggle functionality can also be toggled in the Control_ panel by clicking on them with the mouse, when that happens a message will be sent on the topic "control_panel". When one of the buttons is clicked on by the mouse the control panel also publishes messages on the topic joy.

C. Encodercomm

Encodercomm is the ROS node that talks to the encoder circuitry. It listens to a usb serial communication from a dedicated Arduino for encoders. The received data is the number of ticks from the encoder since the last read. The ticks are converted to speed in meters per second and then published on the topic "wheel_velocity" as a "std_msgs/Float32MultiArray". Left wheel speed is published on the "0" position in the array and right wheel speed is on "1".

1) Encoder_listener: Encoder_listener is Arduino code that count the ticks from encoders and send them over serial at a set interval.

D. Engine_mgmt

Engine_mgmt is the main hub of the Permocar. Control signals are fed to Engine_mgmt. The output is the drive signal to the engines. Engine_mgmt advertise a "geometry_msgs/Twist" message to the topic "motor_power". It subscribes to the topics "joy", "lidar_stop", "vw_estimate", "wheel_velocity" and "referens" with the corresponding messages "sensor_msgs/Joy", "std_msgs/Bool", "geometry_msgs/Twist", "std_msgs/Float32MultiArray" and "geometry_msgs/Twist".

Engine_mgmt feeds control loop with the signals and references gathered creating an output speed for the motors. Values from "vw_estimate" is used in the control loop as the feedback term. "Lidar_stop" is a security measure to avoid collision. If it's true the vehicle is about to crash, so the reference values are set to zero which leads to a stop.

As a security measure the node is initialized with the handbrake on. It is released by pressing joy button "7" ("start" on a wired Xbox controller). It is recommended to always leave the vehicle with the handbrake on.

E. Hector_slam

Hector_slam is a collection of nodes that collaborate to keep track of movement and produces a map. The position estimation can be done by other nodes, and then be fed to the mapping part of Hector_slam, specifically hector_mapping. Currently Hector_slam is running both the mapping and localization on its own. The version we use is a fork from hector slams github [16].

F. Kalmanfilter

The node kalmanfilter is an implementation of an extended Kalman filter. It reads the control signal on topic motor_power (geometry_msgs./Twist), the velocity's from

vw_estimate (geometry_msgs/Twist). It outputs the state estimate on vw_hat (geometry_msgs/Twist) and torques for right and left wheel on torque (geometry_msgs/Twist). Currently this node is not in use; it works in simulation (see section X) but in realty it is not good enough.

G. LMS1xx

LMS1xx is a node that communicates with the lidar and publishes the scan data on "scan" as a "sensor_msgs/LaserScan". The lidar has a scanning frequency of 50 hz when it is gathering data, this data is taken and organized by "LMS1xx" and then published. The version of this code that is used is a fork from clearpathrobotics LMS1xx github [14].

H. Motorcomm

Motorcomm is the last ROS node of the system and the only one to come into contact with the engines. Motorcomm subscribes to "engine_power" and sends the data forward through usb serial to a dedicated Arduino that controls the engine drivers. The serial communication uses a custom protocol, design to minimize overhead without loosing precision. Motorcomm maps the incoming values $[-100, 100]$ to two things: $\pm$ and $[0, 200]$. This is done to get a simpler conversion to pwm signals.

1) motor_driver: Motor_driver runs on a Arduino that controls the engine. Motor_driver reads the serial communication and controls the engines. If motor_driver doesn't receive any communication it will put the engines to a halt.

I. Nmea_navsat_driver

This is the GPS node. It parses standard NMEA (National marine electronics association) messages coming over a serial port into a standard ROS message that it then publishes. All the node needs is the path to the serial port and the baud rate, /dev/ttyUSB0 and 9600 in our case.

J. PadPub

PadPub is a publishing node that advertises on the topic "joy" with the messages "sensor_msgs/joy". PadPub listens to the joystick commands (through a hardware driver) and re-structures it into sensor_msgs/joy messages and publish. PadPub is a generic node so it can handle many different Joysticks. PadPub uses parameters to calibrate itself. The parameters that are set for the Permocar are:

- deadzone = 0.07 $\rightarrow$ Sets a marginal for zero on the sticks of the controller so they have a larger area where they are perceived as centered and therefore counter any small unintended movements of the sticks.
- autorepeat_rate = $20hz$ $\rightarrow$ Sets the update frequencies to ensures that the systems runs smoothly.
- coalesce_interval = $0.05hz$ $\rightarrow$ Sets the maximum update frequency of the publish subroutine.
- default_trig_val = $true$ $\rightarrow$ set the initial values of the trigger to the correct values.

The version that is used is a fork from ros-drivers/joystick_drivers. [13]

The button mapping is as follow: (the number is the index in the joy message and then character is the corresponding name on a Xbox 360 controller)

- Button 0 "A" $\rightarrow$ emergency stop, forces output to zero and cut the power to the engines.
- Button 1 "B" $\rightarrow$ overrides lidar_stop.
- Button 2 "X" $\rightarrow$ not used.
- Button 3 "Y" $\rightarrow$ turn on and off warning light.
- Button 4 "LB" $\rightarrow$ not used.
- Button 5 "RB" $\rightarrow$ not used.
- Button 6 "back" $\rightarrow$ not used.
- Button 7 "start" $\rightarrow$ handbrake.
- Button 8 "XBOX" $\rightarrow$ not used.
- Button 9 "L-STICK" $\rightarrow$ not used.
- Button 10 "R-STICK" $\rightarrow$ not used.

And the the mapping for the axes:(the number is the index in the joy message and then character is the corresponding name on a Xbox 360 controller)

- Axes 0 "left stick right/left" $\rightarrow$ steering.
- Axes 1 "left stick up/down" $\rightarrow$ not used.
- Axes 2 "left trigger" $\rightarrow$ revers.
- Axes 3 "right stick right/left" $\rightarrow$ not used.
- Axes 4 "right stick up/down" $\rightarrow$ not used.
- Axes 5 "right trigger" $\rightarrow$ throttle.
- Axes 6 "pad right/left" $\rightarrow$ discreet, not used.
- Axes 7 "pad up/down" $\rightarrow$ discreet, not used.

K. Reference

Reference is an interpreter for the reference signal from joysticks inputs to desired speed. It listen to topic "joy" (sensor_msgs/Joy) and publish on topic "referens" (geometry_msgs/Twist). Input parameters are max velocity and max turning velocity as "max_velocity" and "max_angular_velocity".

L. Relaycomm

Relaycomm is the final ROS node before the relays. It communicates over usb serial to an Arduino that energizes relay accordingly. Relaycomm subscribes to relay_control.

The data sent is an array with 4 values, directly linked to the relay.

1) Relay_setter: Relay_setter runs on a Arduino that controls relays to turn on and off features using the pins on the Arduino. The pin setup is:

- relay 0, pin 2, power to engine.
- relay 1, pin 3, power to engine.
- relay 2, pin 4, warning light.
- relay 3, pin 5, not used.

(pins 0 and 1 are reserved for serial communication).

M. Rosinstall

All the packages and their corresponding nodes have separate git repositories therefore rosinstall is used for greater ease of installation of all the packages and nodes. In the main git there are two bash scripts, UpdatePC.sh and UpdatePi.sh, and two rosinstall files, permo_car_pc.rosinstall and permo_car_rpi.rosinstall. The current function of the bash script is to copy the relevant install file to the relevant workspace, then uses the wstool to initialize the install file and then builds the code that has been put in the workspace by the install file. The install files have git command lines for specified packages which holds: the name of the package, a link to the git repository of the package and the branch of the repository.

N. Roslaunch

The roslaunch function starts a roslaunch file with specified nodes so that it's not needed to start one node per ubuntu terminal, roslaunch also starts a roscore for the nodes. Five roslaunch files are used in this project: one for the computer used to drive the Permocar pc_launch.launch, one for the raspberry pi on the Permocar itself pi_launch.launch The other launch files are launched through one of these two. They each launch a sub part of the system. Parameters such as deadzone mentioned in PadPub can be set in the computer launch file. As long as there are parameters in nodes the launch files can set them on initialization. The setup for the launch files are:

- pi_launch.launch:
  - Coll_detect node.
  - Encodercomm node.
  - Engine_mgmt node.
  - Kalmanfilter node.
  - LMS1xx launch file.
  - Motorcomm node.
  - Nmea_navsar_driver node.
  - Relaycomm node.
  - Settings node.
  - Vw_generator node.
- pc_launch.launch:
  - Control_panel node. Hector_Slam myugv launch file.
  - PadPub node.
  - referens_node.
  - Rviz node (for live lidar plot).
- LMS1xx.launch:
  - LMS1xx node.
- myugv.launch:
  - Geotiff_mapper_only launch.
  - Hector_mapping node.
  - rviz node (for map).
  - tf node.
- geotiff_mapper_only.launch
  - Geotiff_node node.

O. Rviz

Rviz is a node that can listen to a variety of topics and display the data in a graphical manner. In the Permocar project rviz is used to plot the lidar data (topic: "scan"). With help of the plot it is possible to drive the Permocar with only the plot as guidance and help. The version that is used is: ros-visualization/rviz. [12]

P. Settings

The settings node is running on the vehicle and is used for live settings. The settings that should and can be set in run time are handled through the settings node. It is the node that controls relays for engine power and warning light. It subscribes to joy (type sensor_msgs/joy) and control_panel (type std_msgs/Int8MultiArray). After some logic it is publishing to relay_control using a "std_msgs/Int8MultiArray. The relays for the engines and the warning light can be toggled with either "A" and "Y" button respectively on the Wired xbox 360 controller or by clicking on their corresponding buttons in the Control_panel.

Q. vw_generator

The feedback linearization depends on getting the current velocity and angular velocity from sensor data. The encoders only provide wheel velocity measurements, so this node estimates the velocity and angular velocity for The Permocar from that data and published the estimates in a separate topic.

IX. FMEA

A Failure Mode Effect Analysis has been made and can be found in our git. [1]

We have considered failures where:

- A node randomly shuts down or freezes
- A cable is cut
- A node starts sending erroneous data due to bad hardware or software bug.
- A communication link is broken
- Latency/delay issues

X. Results

The Permocar can be remote controlled over LTE with ROS nodes.

A. Results modelling and control theory

Due to large model uncertainties the controller never worked at all, having issues with diverging control signals and instabilities. Instead, to be able to drive the vehicle at all, a feedback linearization controller based on only the model equation (24) was used. A translation table from torque to voltage was estimated by hand to get drivable results. With this ad hoc solution the permocar follows a straight line well and turns nicely. But the controller has large steady state error. About 10 percent in speed and an even larger errors in the angular velocity..

B. ROS results

The Permocar, and teleoperation, run in ROS. Everything is working better than expected and ROS has turned out to be a really useful tool for developing robots. The load on the Raspberry Pi never exceeds half of the CPU capabilities and usually hovers around the 20 percent mark.

Dividing code into nodes for each task greatly improves readability of the code as well as the functionality of it. Since each node is a self standing executable, the robot automatically runs in a threaded environment where the underlying operating system handles the multi-threading. This means a great performance increase. The division also produces much more modular code: should some hardware change only the affected node has to be re-written, the more abstract tasks in other nodes that rely only on the data are unaffected.

Due to the characteristics of ROS our projects code is now much easier to read and the structure is vastly improved.

C. SLAM

In some cases useful maps of the Robots surrounding were produced. They can be used to navigate by, both by human and robots. The tuning of Hector_slam is a whole science in it self. Depending on the data and surrounding circumstances different settings may be optimal. The settings used in the Permocar is tuned to be good all the time. These means that sometimes the maps produced contain enough errors that they are unusable. When the tuning turns out to be well suited for the data the mapping works great. Figure (13) in appendix showing a map of the hallways outside our project room..

D. Hardware

All sensors provide useful data which shows that the hardware involved works well. We never got erroneous data or intermittent data gaps. The permocar is drivable so all propulsion related hardware also works well. There are no know bugs in the hardware, in all testing have the hardware responded as expected on software commands. The encoder hardware could be ordered on a PCB in the future to improve its longevity/life span.

E. LTE

Everything works well over LTE. Small increases in latency can be observed (measured latency 100ms) under normal operating conditions. Since the operator does not set motor signals directly but rather reference values this is not an issue. The local control algorithms handle minor discrepancies and disturbances.

The VPN tunnel also adds some complexity but it seems to not affect the project negatively in any way and also adds a layer of security.

F. SCReAM

The camera feeds work well over LTE in real time as long as the vehicle has at least a 5 Mbps connection. The compression automatically adjusts the bitrate to match a set latency threshold.

XI. Conclusions

The remote operation of the vehicle works well and as intended. To move this system onto another vehicle should be very straight forward. The small modifications that might be needed would be easy to make since the code is built to be modular. The main task of this project is thus fulfilled.

The blind identification did not converge fast enough for us to be able to get useful parameters in the time span of this project. Setting up a test bench to actually measure torque or other parameters might have allowed us to actually use the intended control scheme successfully. This also means that no real results can be produced on the actual system since

it diverges, crashes into walls and has to be stopped with an emergency button.

Had the feedback linearization worked the system would have behaved linearly providing a good platform to implement a linear MPC for path planning and trajectory following.

In figure (7) one can see that the problems we have on the Permocar can be found in simulations by modifying parameters set by the blind identification. This shows that our problems most likely reside in the blind identification.

The readability of the code might be further improved in the larger nodes by the use of .h files. Some nodes also communicate over topics with standard message types that do not fit accurately. Implementing non standard messages for e.g. the motor velocities would further improve the readability of the code, but it also adds complexity to the project.

Adding an inertial measurement unit, IMU to get a good estimate of the current position would have helped the SLAM algorithm, or even better perhaps would be to separate the problem into one positioning node and one mapping node. The positioning node would fuse sensor data from the lidar,the encoders and an IMU to provide an accurate position. The drift would be minimal by tracking landmarks with the lidar. If outside the GPS information could also be used to get an even better estimate without drift. Then a mapping node is quite straightforward to implement. Since the permocar has 4 cameras as well, they could be used as sensors, providing velocity information through e.g. optical flow algorithms.

If the battery voltage became too low under load the power to the computers may drop below useful levels leading to a system crash. This is not deemed an issue as this implementation is meant to run either on an electric vehicle with a separate power supply to the engines that cut out before the system power does, or a vehicle with a separate power supply, e.g. diesel engine with a generator, that do not experience issues like power dropping out. A small backup battery could be installed to really ensure a safe system shutdown in case of power failure.

Acknowledgments: We would like to extend a very warm thank you to Khalid Atta for his never ending patience for all our questions regarding control theory, Damiano Vargonolo for his help with system identification, Dariusz Kominiak for providing the wheelchair base and lots of electronics and Ericsson RnD in Luleå for providing the LTE equipment.

REFERENCES

[1] Main git repository for project https://github.com/Trobolit/PermoCar

[2] SCReAM, Erisccon. https://github.com/EricssonResearch/scream

[3] Mining challenges https://www.svtplay.se/video/20244774/rapport/rapport-30-dec-19-30-3?start=auto&tab=2018

[4] LENNY DELLIGATTI *SysML DISTILLED*, Pearson, Crawfordsville, Indiana

[5] *Lagrange mechanics* https://en.wikipedia.org/wiki/Lagrangian_mechanics

[6] *Euler's laws of motion* https://en.wikipedia.org/wiki/Euler%27s_laws_of_motion

[7] *Generalized coordinates* https://en.wikipedia.org/wiki/Generalized_coordinates

[8] Wolfgang Birk, Andreas Johansson, and Khalid Atta *Nonlinear control: Feedback linearization, gain scheduling and sliding mode* Lecture notes, LTU, Luleå

[9] Wolfgang Birk, Andreas Johansson, and Khalid Atta *Optimal state estimation* Lecture notes, LTU, Luleå

[10] ros.org *ROS* http://www.ros.org

[11] Peter Corke *Robotics, Vision and Control* publiched by Springer international 2017

[12] RViz *rviz* https://github.com/ros-visualization/rviz

[13] PadPub *PadPub* https://github.com/ros-drivers/joystick_drivers

[14] LMS1xx *LMS1xx* https://github.com/clearpathrobotics/LMS1xx

[15] Simulation Repository https://github.com/lookingflaxy/Permocar_simulations

[16] Hector_slam https://github.com/tu-darmstadt-ros-pkg/hector_slam

[17] SICK LMS111-10100 https://www.sick.com/se/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms111-10100/p/p109842

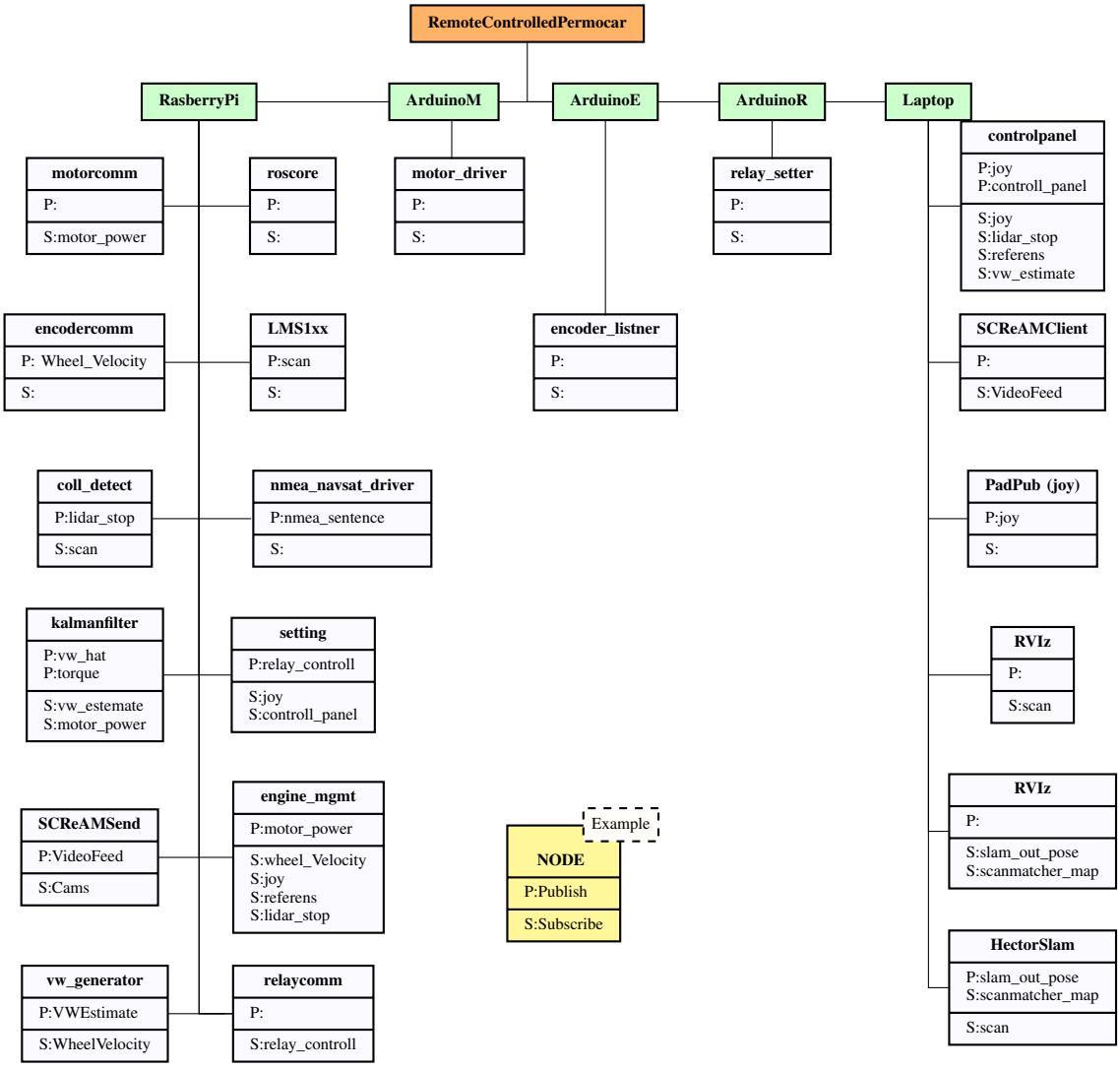

Fig. 8: ROS structure diagram with all nodes with their publish and subscribe channels.

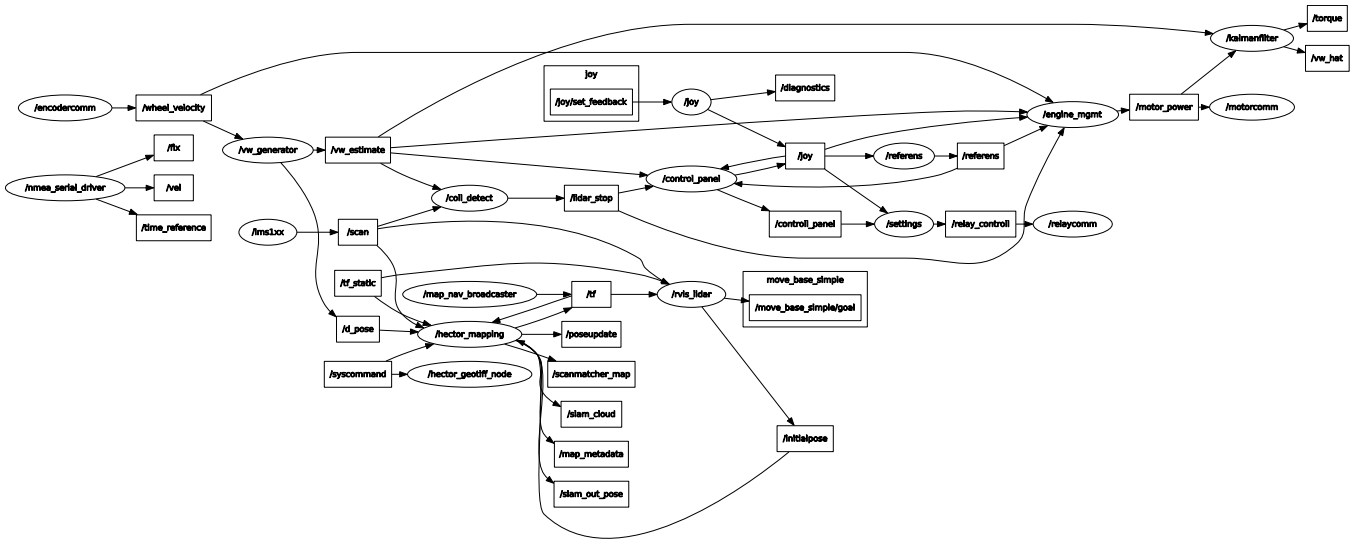

Fig. 9: Scheme of nodes (ovals) and topics (squares).

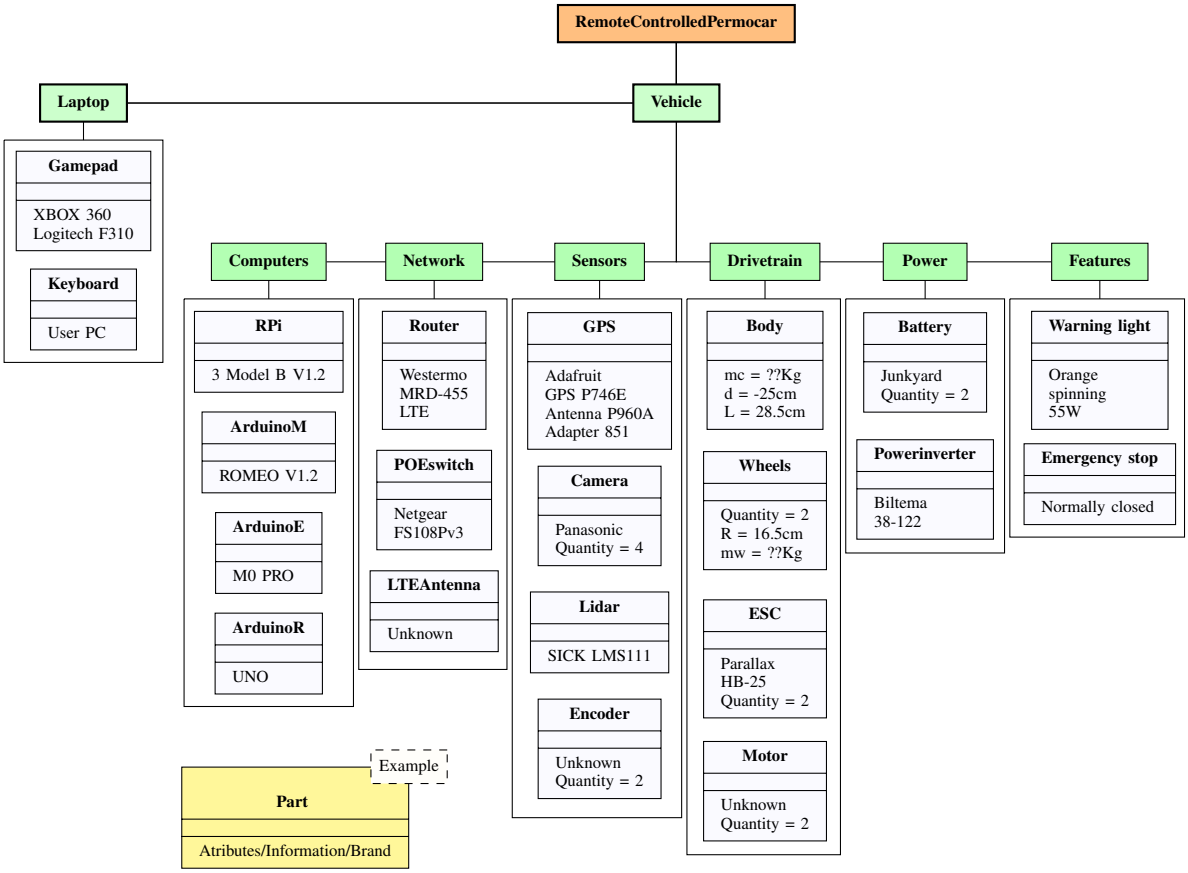

Fig. 10: Hardware diagram showing all the the hardware and information about each part.

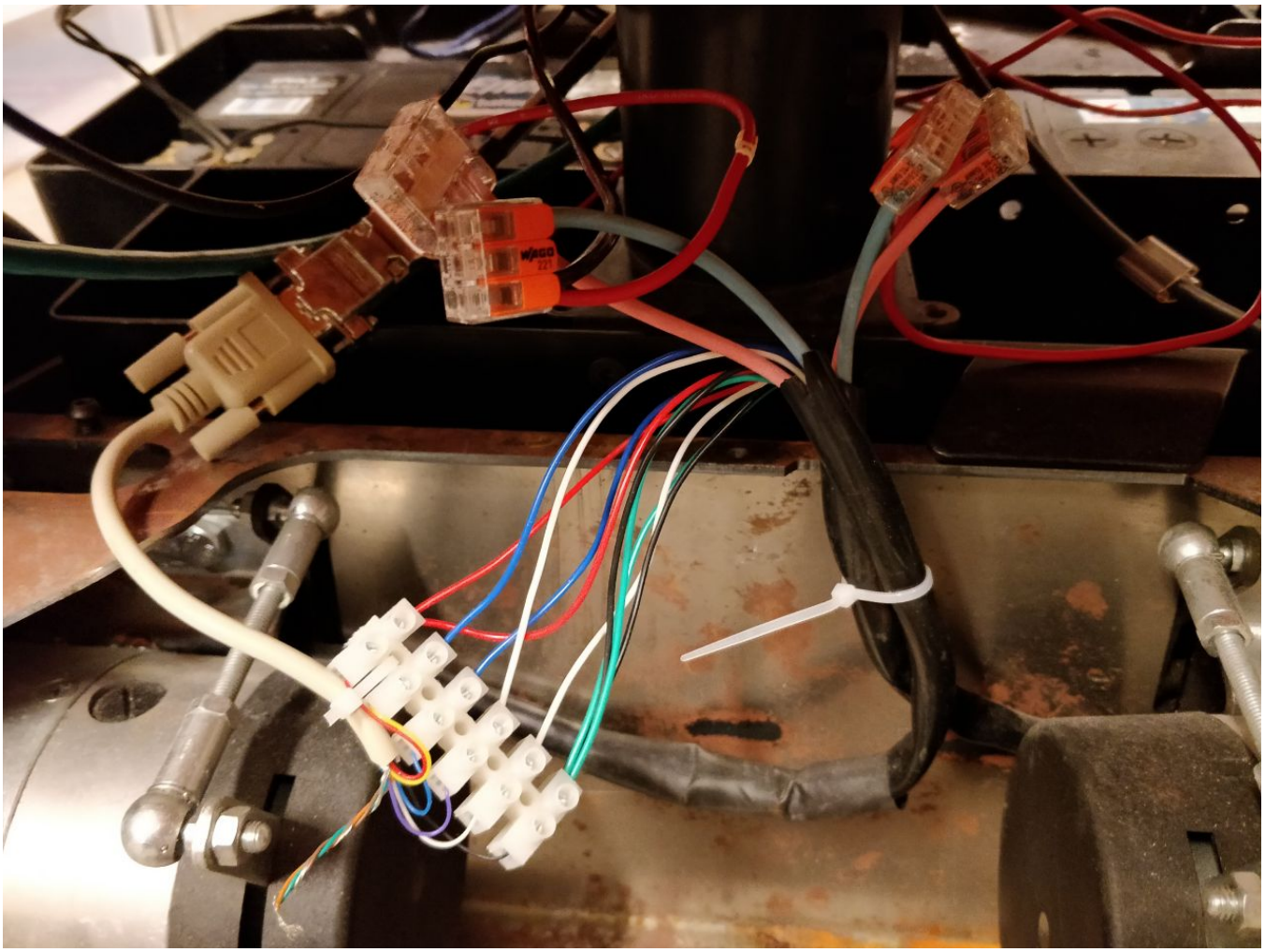

Fig. 11: Encoder cables.

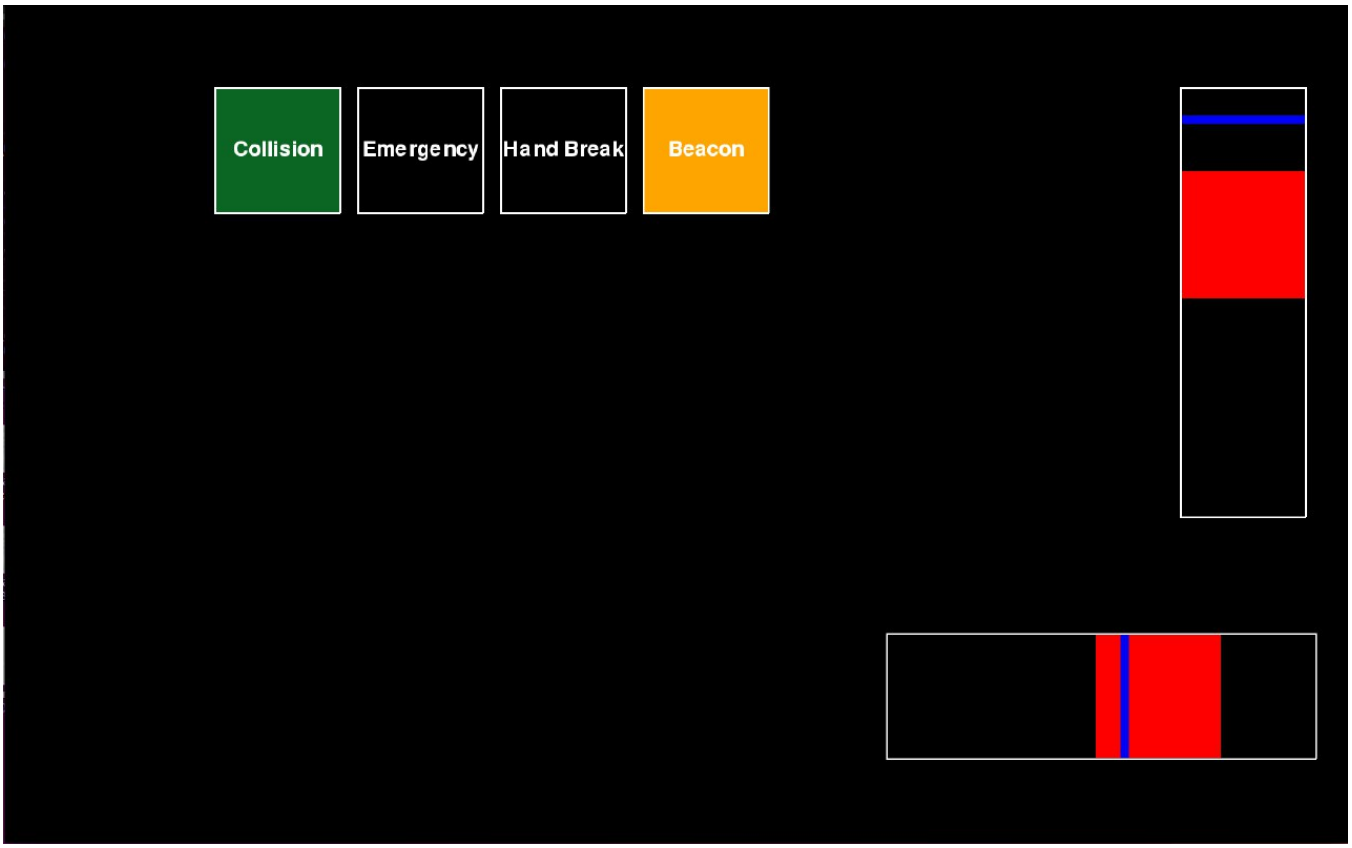

Fig. 12: The Control_panel HMI.

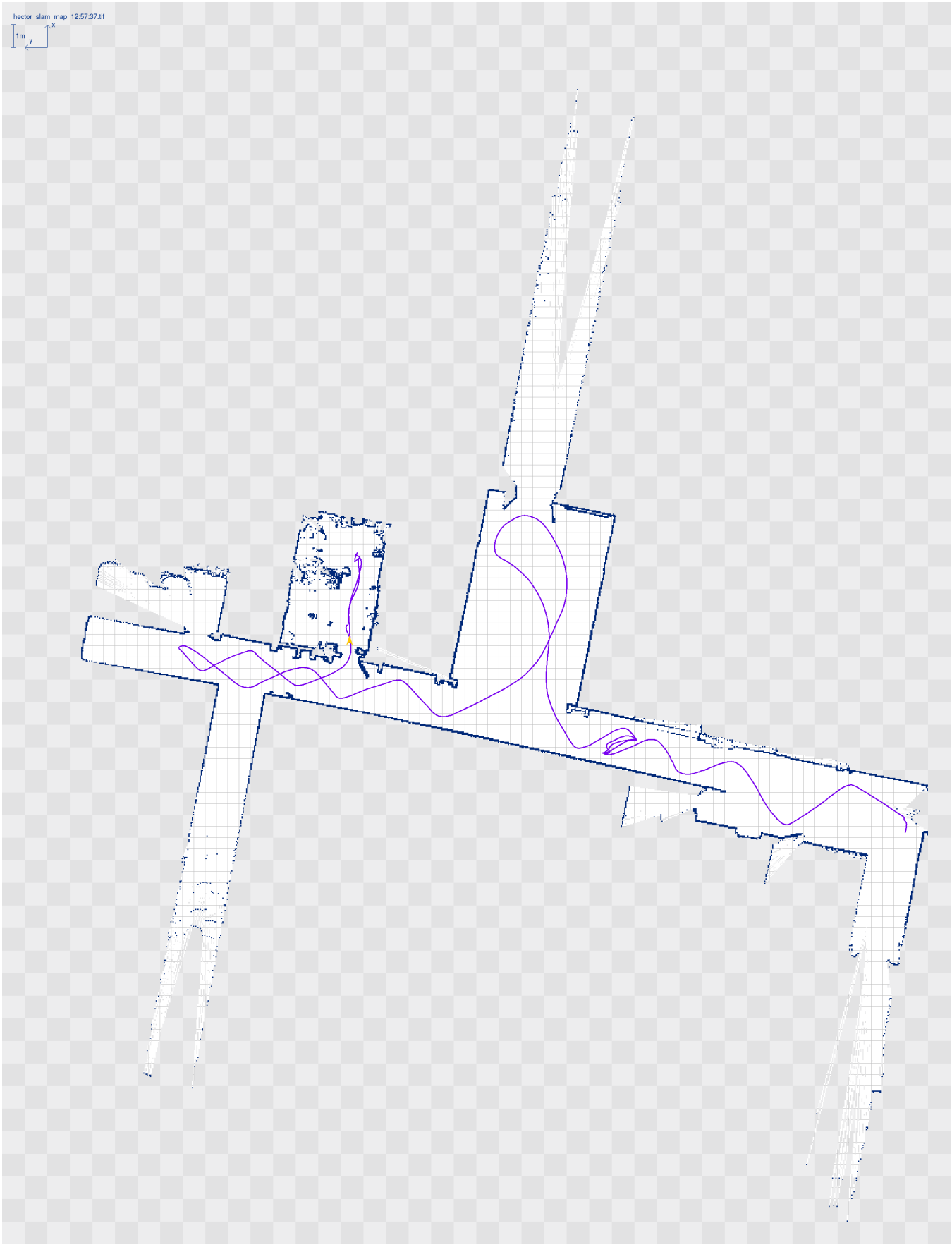

Fig. 13: A map with localization produced of the hallways outside our project rum.