Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Model car for the F1/10 autonomous car racing competition

Martin Vajnar
Cybernetics and Robotics
Systems and Control

May 2017
Supervisor: Ing. Michal Sojka, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Vajnar Martin**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Model formule pro soutěž autonomních aut F1/10**

Pokyny pro vypracování:

1. Seznamte se s pravidly soutěže autonomních modelů aut F1/10.
2. Z dodaných komponent zkonstruujte soutěžní model auta.
3. V systému ROS navrhněte a implementujte software pro autonomní řízení zkonstruovaného modelu auta.
4. Ve spolupráci s dalšími členy týmu implementovaný software otestujte.
5. Výsledky zdokumentujte a odůvodněte rozhodnutí činěná při návrhu.

Seznam odborné literatury:

[1] http://f1tenth.org/
[2] http://www.ros.org/

Vedoucí: ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.

vedoucí katedry

prof. Ing. Pavel Ripka, CSc.

Děkan

V Praze, dne 30. 1. 2017

Translation of the Czech version of the Diploma Thesis Assignment

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Vajnar Martin**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Model car for the F1/10 autonomous car racing competition**

Guidelines:

1. Get familiar with the rules of the F1/10 autonomous car racing competition.
2. Build a racing car model from supplied components.
3. Design and implement software for autonomous control of the built car in the ROS system.
4. Test the implemented software in cooperation with other team members.
5. Document the results and justify decisions taken during the design.

Bibliography/Sources:

[1] http://f1tenth.org/
[2] http://www.ros.org/

Diploma Thesis Supervisor: ing. Michal Sojka, Ph.D.

Valid until the summer semester 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 30, 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne 26.05.2017 ……………………… [illegible] ……………………………………

Podpis

Acknowledgement /

First, I would like to thank my supervisor Ing. Michal Sojka, Ph.D. for useful advices and patience, next I would like to thank Ing. Joel Matějka for help with making videos and his advices concerning electronics. My thanks also goes to Ing. Michal Vokáč and the Flamingos robotic team at the Czech Technical University in Prague for lending me the SICK LiDAR and to other past or present members of the F1/10 team. I would also like to thank Ing. Pavel Píša, Ph.D. for advices concerning the Teensy MCU firmware and other various topics. Last, but not least, I would like to thank my parents for supporting me throughout my studies.

Abstrakt / Abstract

Tato diplomová práce se zabývá konstrukcí autonomního závodního modelu auta. Cílem bylo účastnit se soutěže F1/10. Práce se zabývá mnoha aspekty konstrukce od mechanického návrhu, přípravy elektroniky až po softwarovou architekturu a algoritmy. Software auta je založen na robotickém operačním systému (ROS), který je stručně představen v práci a je zhodnocena vhodnost některých jeho komponent pro auto. Konečná softwarová architektura je navržena a vyhodnocena v experimentech s reálným autem. Výsledkem je funkční model autíčka schopný samostatného průjezdu předem stanovenými kontrolními body. Určitým omezením návrhu je zastavování na kontrolních bodech.

Klíčová slova: SLAM, vizuální odometrie, sledování trajektorie, mobilní robotika, autonomní řízení, ROS

Překlad titulu: Model formule pro soutěž autonomních aut F1/10

This thesis deals with construction of an autonomous race model car. The goal was to participate in F1/10 competition. The thesis covers many aspects of car construction from mechanical design, electronics preparation to software architecture and algorithms. Car software is based on the Robot Operating System (ROS), which is briefly introduced in the thesis and fitness of several of its components for the car is evaluated. The final software architecture is proposed and evaluated in real-world experiments. The result is working model car capable of autonomous passage through predefined checkpoints. Its limitation is stopping at the checkpoints.

Keywords: SLAM, visual odometry, trajectory following, mobile robotics, autonomous driving, ROS

Contents

Tables / Figures

Chapter 1
Introduction

Autonomous robotics becomes ever more popular as the processing power of embedded platforms increases to levels previously seen only in full-fledged desktop computers while their power consumption only reaches fractions of their desktop counterparts. An example of such a platform is the affordable NVIDIA Jetson Tegra K1. These platforms bring new possibilities to the world of mobile robotics as it is now possible to perform computationally intensive tasks such as Simultaneous Localization and Mapping and visual odometry directly onboard a mobile robot.

This is especially useful for the autonomous driving in the automotive industry. This field, however, requires not only high performance, but also high reliability and easy integration into existing vehicle. An example of automotive oriented platform is the recently released NVIDIA Drive PX 2 that can, according to the manufacturer [1], perform sensor fusion from multiple cameras, LiDAR, RADAR and ultrasonic sensors. It is also capable of using Deep Neural Networks (DNN) for the detection and classification of objects as well as mapping "raw pixels from a single front-facing camera directly to steering commands" [2].

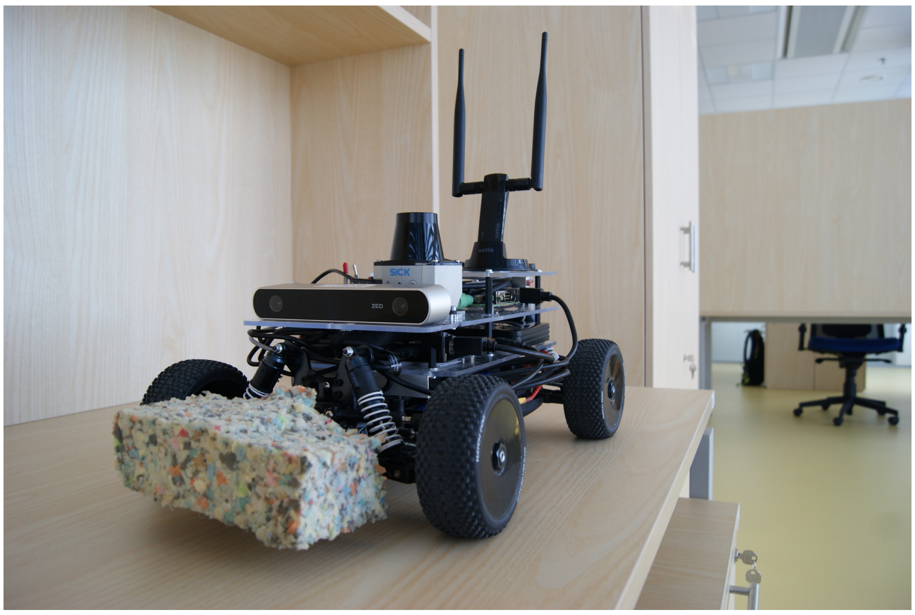
Figure 1.1. Image of the developed platform.

As a consequence of introduction of the mentioned NVIDIA platforms many different autonomous scale model race car competitions and platforms emerged. More about them is in Section 2.5. These competitions have common topics including car localization and mapping, trajectory planning, obstacle avoidance, sensor fusion, visual odometry, etc. In our Department, we have chosen to participate in the F1/10 competition, because a scale model and a LIDAR were available at the Department from previous projects. Unfortunately, the April 2017 round of the competition was cancelled, because the organizers “did not receive enough team confirmations in time for organizing, planning, and ensuring an exciting racing event” [3].

The competition preparation project was run by a small team of students. This thesis documents my contribution to the preparations of our scale model race car platform to take part in the F1/10 competition. The developed platform is depicted in Figure 1.1.

The thesis is structured as follows. In Chapter 2 the F1/10 competition is described. Namely its rules and aims as well as a list of materials and ROS source codes provided by the competition organizers. Additionally, the chapter describes the parts of ROS ecosystem, that we used during the work, and finally the equipment used to construct the platform is described.

Chapter 3 describes my contribution to the project: a custom built chassis to mount the required components, firmware changes to microcontrollers, testing of ROS application software (packages), chosen ROS packages and the final ROS architecture.

Chapter 4 describes various experiments performed with the car.

In Chapter 5 we provide recommendations for future improvement.

In Conclusion we summarize the achieved results.

Throughout the thesis we use abbreviations and their explanation is provided in Appendix A.

Chapter 2
Background

In this chapter we first describe some important parts of rules of the F1/10 competition, that affected software design decisions. Next we comment on materials provided by the competition organizers, why we chose to use ROS to implement the autonomous stack and basics of ROS operation. Then we focus on the hardware platform of the car. Last, we list similar competitions and scale car platforms.

2.1 Rules of the F1/10 competition

The complete rules are available in [4]. Here we only cite parts important for the hardware design and software implementation of the platform.

According to section 4.4 of the rules each competitor has to:

a) Construct a 1/10th scale autonomous vehicle within the constraints described herein.
b) Provide a robust and safe mechanism for bringing an errant vehicle to a stop as described in the rules.
c) Demonstrate teleoperation of the vehicle in order to verify basic functionality.
d) Persistently complete a mission defined by the safe traversal of ordered series of checkpoints with the objective of minimizing completion time.
e) Interpret static obstacles within the environment in order to maintain collision free progress.
f) Exhibit context dependent speed and angular velocity control in a static environment.
g) Interpret dynamic obstacles within the race environment in order to enable predictive controls and planning, such as is necessary to ensure collision free progress.
h) Exhibit context dependent speed and angular velocity control in a dynamic environment.
i) Navigate in areas where sensors may not provide map-based localization (ie. LiDAR range is insufficient).
j) Accomplish these goals using a low power embedded processor specified by the organizers.

Another important part of the rules is 7.1, which describes safety requirements:

a) Each car must have an emergency disconnect switch (affectionately: the big red button) that removes all power from the main power system (i.e. the batteries that supply power to the drive motor) at a point in the circuit as close as practical to the power supply. This switch must be rated for a breaking current of at least 50 percent greater than the power limiting fuse rating.
b) Each car must have the ability to switch to Manual Mode directly from Autonomous Mode at any time using only the Traxxas Remote.

Section 7.2 deals with autonomy and communication.

The F1/10 competition is for fully autonomous, self-contained vehicles, as such:

a) No transmitters or communication beacons (other than Wi-Fi and race infrastructure related communication) of any kind are allowed. Everything necessary for the vehicle's navigation/processing/sensing must be attached and part of the vehicle itself.
b) Cooperation amongst vehicles is strictly prohibited. Two teams cannot collude to manipulate race outcomes or share computational resources.
c) Competitors may not modify the course in any way. Including but not limited to covering of reflective surfaces or the addition of markers/beacons.

Section 7.4 describes hardware of the platform:

a) Sensor Configuration: Each team must choose a sensor configuration only from this subset of sensors. Alternate sensors will not be considered, the purpose of this competition is the development of driving algorithms. Please see the bill of materials for part numbers and ordering information.
   1. Camera (at most 2): ZED, Structure Sensor, Pointgrey, Minoru, Webcam
   2. LIDAR (at most 1): Hokuyo 10LX, RP Lidar, Hokuyo 04LX
   3. IMU (at most 1): Memsic IMU440CA-200, Razor
   4. WIFI (exactly 1): Ubiquiti Picostation M2
b) Computing: Each car must use only:
   1. Planning and Perception (exactly 1): NVIDIA Jetson TK1
   2. Control (exactly 1): MBed or Teensy
c) Chassis and Vehicle:
   1. Vehicle (exactly 1): Traxxas Rally 1/10
   2. Suspension (swappable): Any
   3. Axle Conversion (swappable): Stock, ST Racing Concepts ST3654-17S CNC Machined Aluminum 17mm Hex Conversion
   4. Tires (swappable): Stock, Duratrax Bandito Buggy Tire C2 Mounted White (1/8 Scale)

However, the list of allowed components (Bill of Materials) changed during time as various components such as battery pack, IMU and vehicle went out of stock [5].

Another important part of the rules is 9.1 which describes the race track:

a) The race track will be located at Carnegie Mellon University in Pittsburgh. Although the final layout has not yet been selected, teams may expect that racing will occur in hallways roughly 2 – 5 meters wide. Furthermore, teams may expect multiple turns (in both directions!), uneven walls, varying lighting conditions, and in later rounds another autonomous competitor.
b) Maximum and minimum width of the course: 4 meters. Minimum centerline radius turn on the course: 3 meters details will be provided in the RNDF and ROSBAG data.

The track layout specification was later updated by organizers on the F1/10 forum [6]:

1. The track width will be at-least 4 – 5 car widths,
2. The track will be a loop,
3. The height of the track boundary/walls will be high enough that a LIDAR mounted on the top of the car can detect the boundaries.

Section 10.2 of the rules describes the race starting procedure:

1. The participants will be expected to connect to a common wireless network. It is the responsibility of the team to make sure they have been assigned unique IPs and test the connection to the wireless network
2. The organizers will send a trigger signal that is meant to start the race. To ensure that all cars receive the signal at the same time, it is required for all cars to “subscribe” to the topic “ReadySetGo” that will be “published” on the common wireless network.
3. As soon as a boolean “True” is received on this topic, the competing cars are expected to begin the race. The dataType that is expected on the the “ReadySetGo” topic is Boolean. Teams should verify that their starting protocol works as expected in their run of the Qualifying lap.

2.2 Materials provided by the F1/10 organizers

The F1/10 competition organizers provide numerous materials to make it easier for newcomers to start work [5, 7]. These are either video tutorials or written tutorials or source code.

The first batch of tutorials contains build instructions for the chassis used for mounting the used equipment onto the vehicle. These tutorials show how to assemble the chassis with parts laser cutted according to the provided chassis drawings.

Another batch of tutorials focuses on system bring-up[1]), installation of Ubuntu OS and ROS on the NVIDIA TK1 platform and sensor device drivers installation.

Some tutorials also provide basic insight into localization and PID control topics.

There is also some source code provided by the organizers to speed-up the development. Namely the Teensy firmware and some basic keyboard teleoperation of the car. We refer to tutorials used later on (for example in Section 2.4.2 and 3.7]).

2.3 Robot Operating System

The tutorials provided by the organizers make use of the Robot Operating System (ROS). Specifically the Indigo version is recommended. According to [8] the ROS “is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

ROS was built from the ground up to encourage collaborative robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other’s work, as is described throughout this site.”

Possibility of ROS integration with other projects and tools is described in [9]. Currently we use the Gazebo simulator for car simulation. The developed simulator is the topic of Jiří Kerner’s thesis (see [10]). The core components of ROS are described in more detail in [11]. Here we only provide short overview of ROS components, that we used during the development.

[1]) `http://f1tenth.org/lectures`

2.3.1 ROS Architecture

The ROS system consists of independent ROS nodes[1]). The root component of the ROS system is called “roscore”. Roscore starts up a ROS Master, a ROS Parameter Server and a “rosout” logging node. For more details see [13].

“The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. The Master also provides the Parameter Server. The Master is most commonly run using the roscore command, which loads the ROS Master along with other essential components” [14].

In order for a node to communicate with others it is necessary to define data type of the message. The way this is handled in ROS is, according to [15], the following: “Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type. The Master does not enforce type consistency among the publishers, but subscribers will not establish message transport unless the types match. Furthermore, all ROS clients check to make sure that an MD5 computed from the msg files match. This check ensures that the ROS Nodes were compiled from consistent code bases.”

The transport mechanisms used are either TCP or UDP. “ROS currently supports TCP/IP-based and UDP-based message transport. The TCP/IP-based transport is known as TCPROS and streams message data over persistent TCP/IP connections. TCPROS is the default transport used in ROS and is the only transport that client libraries are required to support. The UDP-based transport, which is known as UDPROS and is currently only supported in roscpp, separates messages into UDP packets. UDPROS is a low-latency, lossy transport, so is best suited for tasks like teleoperation.

ROS nodes negotiate the desired transport at runtime. For example, if a node prefers UDPROS transport but the other Node does not support it, it can fallback on TCPROS transport. This negotiation model enables new transports to be added over time as compelling use cases arise” [15].

Parameter server allows for configuration and preservation of state of nodes. According to [16]: “A parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

The Parameter Server is implemented using XMLRPC and runs inside of the ROS Master, which means that its API is accessible via normal XMLRPC libraries.”

The “rosout” topic provides a logging facility, that could be used by all nodes running on the system. There is a Qt-based application for viewing the published data – rqt_console [17].

Multiple nodes are usually launched at once. The spawning of processes is facilitated by “roslaunch” file in XML syntax and a developer can, for example, add nodes to be

[1]) “A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot’s wheel motors, one node performs localization, one node performs path planning, one node provide a graphical view of the system, and so on” [12].

launched or provide configuration values published to the parameter server. For more details see [18]. It is possible to include other launch files and form complex structures with separated functionality. For example in our configuration we use a separate launch file to initialize device drivers and setup ROS topic publications for the measured data and then include this launch file in other files requiring access to the data.

An important part of the ROS system is a build system used to compile user nodes. We use “catkin_make” [19] system. The user application software is divided into packages. A make file is written for each package specifying which source files to use for building which nodes and which libraries need to be linked with the nodes.

One drawback of defining configuration options through a launch file is, that it can be only changed by using “rosparam” command. A more user friendly version is called “dynamic_reconfigure” [20] which provides a graphical tool to change configuration of nodes. It is necessary to define data types of parameters, describe their purpose and provide bounds. All this is accomplished through “ParameterGenerator” Python class. Then in user application one has to instantiate a Server instance and pass it a callback function, that will be invoked on parameter change and passed the new parameter values.

Nodes in ROS can be written using C++ [21], Python [22] or Java [23] language. Rosjava is useful for development of Android applications, that use ROS. One example of such an application is the ROS Control[1]), that allows to remotely control a ROS-based robot and to receive sensor data and visualize them.

2.3.2 ROS Packages

In this section we describe ROS packages, that we used during the development of our racing car. It consists of two separate types of packages. The first group is formed by low-level software (device drivers), that publish data through ROS topics and the other group is high-level application software formed by localization, mapping and planning-oriented packages.

2.3.2.1 Device drivers

ROS includes device drivers for all components used on the car. The specific packages are:

- “hokuyo_node” – package containing drivers for older Hokuyos, such as our 04LX. It publishes the data through the “sensor_msgs/LaserScan” message type [24]. For more details, see [25].
- “sick_tim” – provides a driver for the SICK LiDAR. The output is again published over the “sensor_msgs/LaserScan” message type. For more details, see [26].
- “rosserial” – provides a ROS wrapper, that is capable of serializing/deserializing data sent through ROS topic and sending/receiving them over/through a character device. This is used to communicate with the Teensy MCU. For more details, see [27].
- “razor_imu_9dof” – provides a driver and a firmware for the Inertial Measurement Unit (IMU). The firmware needs to be compiled and downloaded to the CPU on the IMU through Arduino IDE. The data are published through a “sensor_msgs/Imu” [28] topic. For more details see [29].
- “zed-ros-wrapper” – is a wrapper [30] around the manufacturer supplied proprietary ZED SDK [31]. It provides odometry estimates on the “nav_msgs/Odometry” [32] topic, images from left an right camera along with the camera calibration matrices

[1]) `https://play.google.com/store/apps/details?id=com.robotca.ControlApp`

through the “sensor_msgs/Image” [33] topic and the “sensor_msgs/CameraInfo” [34] a depth map is also available through the “sensor_msgs/Image” topic.

My experience with the proprietary SDK is described in Sections 3.4 and 3.13.3.

2.3.2.2 Application software

In this section we provide overview of various ROS components, that we use on the car.

- “navigation” – it is relatively complex set of ROS packages [35]. But the most important one is “move_base”, which, according to [36] “links together a global and local planner to accomplish its global navigation task. It supports any global planner adhering to the nav_core::BaseGlobalPlanner interface specified in the nav_core package and any local planner adhering to the nav_core::BaseLocalPlanner interface specified in the nav_core package. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner (see the costmap_2d package) that are used to accomplish navigation tasks.” As a global planner plugin we use the “global_planner” package. In order to plan a trajectory it uses potential field constructed from the global costmap and a variant of Dijkstra’s algorithm[1]). More details about the “global_planner” package can be found in [37]. The local planner plugin used was developed by me and it’s function is described in Section 3.15. The “move_base” node architecture is depicted in Figure 2.1.

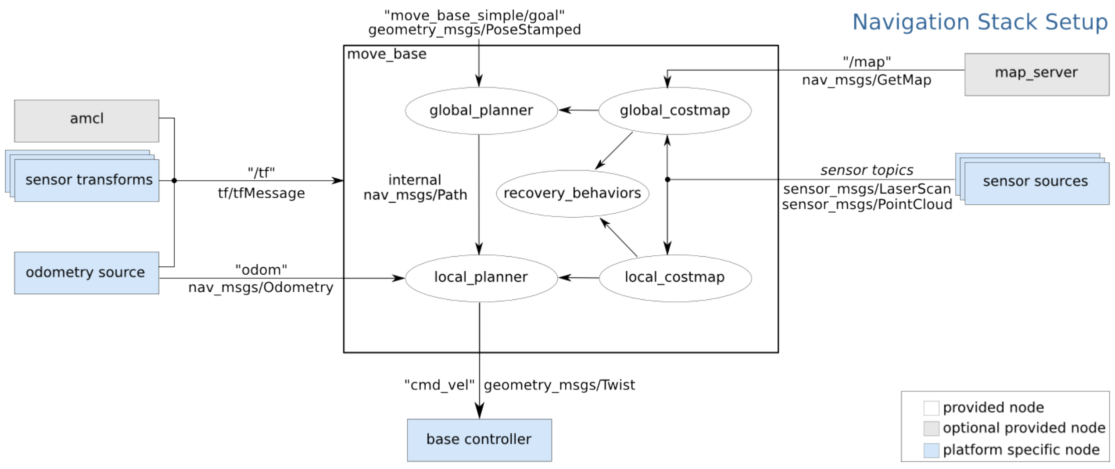

Figure 2.1. Navigation stack setup, source: [36].

- “robot_localization” – this package provides robot’s state estimation performed in either Extended Kalman Filter (EKF) or Unscented Kalman Filter (UKF). We use the UKF variant. For more details about the UKF node, see [38]. Its configuration and tuning is described in Sections 3.11 and 3.9.
- rViz – this is a visualisation tool [39]. It can visualise constructed map, show position of a robot in the map, it can display robot’s odometry as well as planned trajectory. It can even visualize LiDAR scans and IMU heading and acceleration vectors. It allows for interactivity. User can define a pose estimate for localization nodes by selecting “2D Pose Estimate” tool. “2D Nav Goal” tool allows to set a goal for the robot to drive to. The goal is passed to global planner plugin of “move_base” package.
- rqt_graph – visualizes topic connections between nodes [40].

[1]) https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

- rqt_tf_tree – visualizes the ROS Tf subsystem structure [41].
- "map_server" – "map_server provides the map_server ROS Node, which offers map data as a ROS Service. It also provides the map_saver command-line utility, which allows dynamically generated maps to be saved to file" [42].

2.4 Hardware platform

The rules give guidelines on what parts to purchase and use in the competition. However, due to the fact, that when we started working on the F1/10 project, there were already a LiDAR and a vehicle from previous project purchased at the Department, we deviated from the prescribed platform in order to save time and money. The components we used are described in the following subsections. First we describe sensors used and then the computing units.

2.4.1 Sensors

We have two LiDARs available, a stereo camera and an Inertial Measurement Unit (IMU).

- Hokuyo URG-04LX – is a LiDAR with distance measurement ranging from 60 mm to 4,000 mm in a 240 degrees angle. It has angular resolution of 0.36 degree and sampling frequency of 10 Hz. It needs 5 V power source and current consumption is at most 800 mA. It has USB and RS-232C communication interfaces. Unfortunately, we have an older version of this model, that did not support powering through the USB connector.
- SICK TiM551-2050001 – is a LiDAR with distance measurement ranging from 50 mm to 10,000 mm in a 270 degrees angle. It has angular resolution of 1 degree and sampling frequency of 15 Hz. It needs 9 – 28 V DC power source and power consumption is typically 4 W. It has USB and Ethernet communication interfaces.
- ZED stereo camera – according to the manufacturer [43], the camera is capable of working in the following video modes. However, due to insufficient processing performance of the TK1 the attainable framerate and the available range of resolutions is limited (see Section 3.4).

Video Mode	Frames per second	Frames per second (maximum attainable on TK1)	Output Resolution (side by side)
2.2K	15	0	4416x1242
1080p	30	0	3840x1080
720p	60	15	2560x720
WVGA	100	30	1344x376

Table 2.1. ZED camera video modes. Source:[43] and my experiments on TK1

The camera has a baseline of 120 mm. Lenses have 110 degrees Field of View. Sensor size is 1/3″ with backside illumination. It has USB 3.0 interface and draws 380 mA of current from the USB Hub.

- IMU Razor 9DOF – it is a Sparkfun made IMU. It incorporates triple-axis digital-output gyroscope (ITG-3200), triple-axis accelerometer (ADXL345) and triple-axis digital magnetometer (HMC5883L). The sensor outputs are processed using the onboard ATmega328 running firmware specially designed to be compatible with the ROS driver. The maximum sampling frequency of current firmware implementation is only 50 Hz. The ATmega communicates with host over the serial interface. In order to connect it directly to a USB Hub. We use "SparkFun FTDI Basic Breakout – 3.3V" [1]). It includes an FTDI FT232RL integrated circuit.

2.4.2 Computing units

As prescribed by F1/10 rules (see Section 2.1), we have two dedicated computing units (apart from ATmega onboard the IMU). The first one runs Ubuntu and all ROS nodes and the other one is used to capture the output PWM duty cycle of a RF transceiver onboard the vehicle and to generate PWM duty cycle for the servo an the ESC according to values received through the "rosserial" interface from ROS topic subsystem.

2.4.2.1 NVIDIA Jetson TK1

The NVIDIA Jetson TK1 comes with a Ubuntu 14.04 flashed onto the onboard eMMC memory chip. It has numerous interfaces. Mainly it has USB 3.0 controller, mini-PCIe slot, HDMI connector, 1000M Ethernet, SATA port and MOLEX power connector and SD card slot. It uses the Tegra K1 SOC that is formed by an ARM Cortex-A15 CPU (NVIDIA 4-Plus-1 Quad-Core) and NVIDIA Kepler GPU with 192 CUDA Cores.

We used some of the tutorials provided by the F1/10 organizers to setup ROS on the kit. The kit runs and Ubuntu 14.04 and ROS Indigo. Both software components are run from an SD card to save the internal eMMC storage from excessive reads and writes due to rosbag recordings (for the description of a rosbag, see [44]).

The platform with software as provided by NVIDIA has some unusual "features". The kernel by default puts cores offline if the system load is low. This causes delayed reactions after periods of inactivity. Another intriguing property of the kernel is that by default the USB 3.0 controller is downgraded to only USB 2.0 and one has to specifically enable the USB 3.0 functionality.

Due to the mentioned properties of the default kernel, we replaced it by the Grinch kernel[2]) and applied fixes recommended by the F1/10 organizers in the provided written tutorials (see Section 2.2).

2.4.2.2 Teensy 3.2 MCU

The main component of this development board is the Kinetis K20 CPU featuring an ARM Cortex-M4 core. It is a 32-bit core, 1.25 DMIPS/MHz[3]). It is based on the ARMv7 architecture. The board schematic is in Figure 2.2. It can be connected to the host through a Micro-USB connector. Linux system on TK1 sees it as a character device. For more details about the MCU, see [46].

Programming is done via a modified Arduino IDE called Teensyduino [4]). The modifications also include an extensive set of libraries [5]).

The Teensy board includes a preflashed bootloader used to program user applications compiled by the Teensyduino. For usage, see [48].

[1]) `https://www.sparkfun.com/products/9873`

[2]) available here: `http://www.jarzebski.pl/files/jetsontk1/grinch-21.3.4`, change log is available in [45]

[3]) `https://en.wikipedia.org/wiki/Dhrystone`

[4]) `https://www.pjrc.com/teensy/teensyduino.html`

[5]) `https://www.pjrc.com/teensy/td_libs.html`

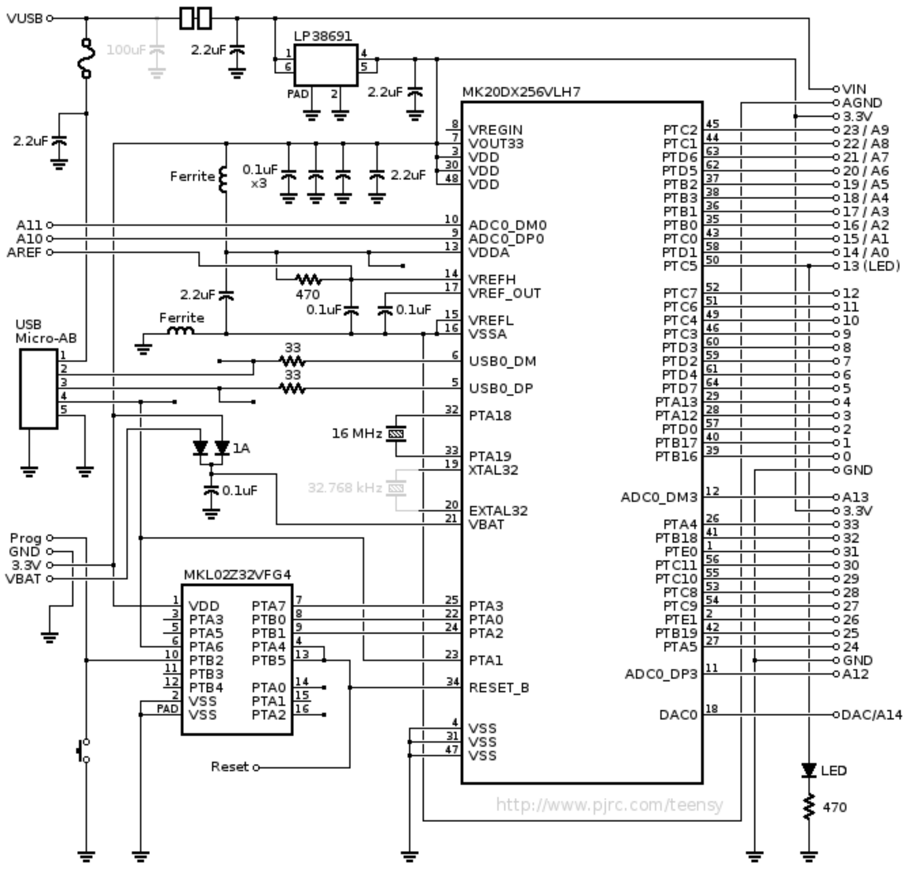

Figure 2.2. Teensy 3.2 board schematic, source: [47].

2.4.3 Other equipment

This section includes description of the supplementary equipment. In the following list we provide a short description of it.

- Netis WF2190 – a WiFi module that has 2 transmit and 2 receive antennas and is capable of attaining speeds of up to 300 Mbps in 2.4 GHz mode and 867 Mbps in 5 GHz mode. It is based on the Realtek rtl8812AU chipset. The required driver is not part of the upstream Linux kernel. It has to be compiled and installed from GitHub[1]). I used the “driver-4.3.22-beta” branch.

  From benchmarks performed with the iperf3 [2]) tool it provided steady throughput of around 20 MB/s when connected to the Linksys E4200 V1 router on 5 GHz band and 40 MHz wide channel.
- AmazonBasics 7 Port USB 3.0 Hub – it is a USB 3.0 Hub used to connect equipment to the TK1. Namely LiDARs, ZED camera, IMU and Teensy board. For details, see [49].
- Energizer XP18000 Battery Pack – is a powerbank used to power all the electronics onboard the car, except for the ESC and servo. It has a 12 V DC output capable of supplying 2 A, 5 V USB charge output capable of supplying 1 A and 20 V DC output capable of supplying 3.5 A. It can accumulate total energy of 70 Wh.

1) `https://github.com/diederikdehaas/rtl8812AU`
2) `https://iperf.fr/`

2.4.4 Vehicle

The vehicle is a leftover from some previous projects at the Department. It is not listed as allowed by competition organizers, therefore we cannot participate in the competition with it.

The car is ECX Revenge 1/8 (P/N: ECX04000), it has been phased out.

The vehicle's frame is made of alluminium. It has 4WD drive and three locking differentials. As a drivetrain the Dynamite Fuze 550 2500 RPM/V (P/N: DYN4960) 3-phase sensorless brushless motor is used.

The motor is controlled by the Dynamite Fuze 130A (P/N: DYN4955) ESC. It is capable of supplying 130 A to the motor and automatically detects the number of cells of the connected battery pack. It also has a number of user-configurable settings (see [50]). On the car, we use the "Forward/Reverse with Brake" running mode of the ESC. To engage reverse while moving forward, in this mode, a user has to apply the brake until the vehicle has come to a complete stop, release brake, then apply the brake again. The ESC also supports the more natural mode "Forward/Reverse" without the required brake, but we do not use it, because the manufacturer discourages its use: "Do not use this mode with car types other than Rock Crawlers as this mode can overload and/or damage the ESC" [50]. We use the ESC with 20 % "Drag Brake Force". This setting causes the brake to be "automatically applied when the throttle is returned to the neutral position" [50].

The steering mechanism is controled by Dynamite Servo Car 3905 (P/N: DYN3905) with 8.5 kg.cm torque and angular speed of $\frac{5\pi}{3}\,\mathrm{s}^{-1}$ at 6 V power supply.

The RC kit is the Spektrum DX2E with RF transceiver Spektrum SR200.

2.5 Similar competitions

As mentioned in Chapter 1 the concept of autonomous scale models moving around buildings, corridors and basements or in a scaled mock-ups of a real-world scenarios is becoming popular. Here we list some of the contests using a similar concept of car:

- MIT Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot (RACECAR)[1]). This platform uses a very similar hardware setup[2]) to the F1/10 competition. Namely the Hokuyo UST-10LX LiDAR, the ZED camera, the Sparkfun IMU and the Traxxas 1/10-scale chassis. But the organizers chosed a more powerful development kit (NVIDIA Jetson TX1) and they equip the vehicles with an open-source ESC – the VESC. The Jetson TX1 provides more computing power than the TK1 we use and the VESC allows to compute wheel odometry estimates and control motor speed. This would be useful to have also on our platform as discussed in Section 5.1.
- Audi Autonomous Driving Cup 2017[3]) – this competition uses a vehicle and chassis specifically designed by Audi. The competition is focused on the development of new ground-breaking algorithms in the field of autonomous driving.

[1]) `http://fast.scripts.mit.edu/racecar`
[2]) `http://fast.scripts.mit.edu/racecar/hardware`
[3]) `https://www.audi-autonomous-driving-cup.com`

Chapter 3
Racing Vehicle Construction

In this chapter we present a thorough description of my contribution to the preparation of our scale model race car platform. First we describe the chassis preparation and completion procedures, then we move on to hardware bring-up and MCU firmware changes, identification of basic kinematic properties of the car, testing of various Simultaneous Localization and Mapping ROS packages as well as stereo odometry ROS packages and describe the components used in the final control architecture. We also present evaluation of the chosen approach in a similar environment to where the competition is to be held.

Througout the chapter we use the notation of different ROS-specific reference frames. For details about the reference frames and their specific axis orientation, please, refer to [51–52].

The LiDAR scans and IMU measurements are transformed to a single reference frame called the "base_link". The orientation is as described by [51, Axis Orientation, In relation to a body].

3.1 Chassis preparations

Since we have a different vehicle than is prescribed for the competition and we were not able to obtain ABS plastic boards 3 and 6 mm thick [5, Bill of Materials], that were recommended by the organizers to use the provided mounting chassis drawings. We had to design our own mounting chassis to attach the required sensors and development kits to the car.

Since we have so many components, they would not fit onto a single level chassis, so a two level chassis was designed. The design of the chassis layout was based on the following requirements:

- Heavy components should go to the lower level to keep the center of mass near the ground to improve handling in sharp turns.
- The space around LiDAR can be occupied only below the height of the LiDAR dome to not obstruct the view, because the SICK LiDAR has a wide viewing angle of 270 degrees.
- The camera needs to be in the front part of the chassis and it's view should not be obstructed by parts of the front wheel suspension.
- The breadboard with Teensy MCU mounted on top needs to be on the upper level as it has manual switches to bypass the Teensy board and connect RF transceiver directly to the ESC and servo.
- Place the emergency stop button on the top level to make it easily accessible.
- Connectors of the power bank and the USB Hub should be easily accessible.
- Respect the workplace size restrictions of the CNC machine available at our Department (32 cm x 21.5 cm)[1])

[1]) `http://rtime.felk.cvut.cz/robot/index.php/Comagrav_MT_Profi`

Two versions of the chassis were made. The first version was made by teammate Jitka Hodná with my assistance.

After the first version was done I discovered, that unfortunately, the camera can sometimes see the front wheels suspension and that we will need to house a newly obtained LiDAR – the SICK. Therefore I prepared a second version of the chassis.

During the drawing phase I used measurements obtained from the specific datasheets of sensors and development kits used and from my own measurements.

Due to thick and inflexible SICK LiDAR power cable SICK LiDAR, I designed a pad to put under it, so that the cable could fit into the small space between connector headers on the TK1 module (see Figure 3.6 and 3.7).

I drew the chassis in AutoDesk's AutoCAD 2017 and then imported it to AutoDesk's ArtCAM 2017 Free to generate a G-code for the CNC machine available at our department. The drawings are on the attached CD.

The material used for the milling was the 4 mm thick Guttagliss Hobbyglas. It is similar to a plexiglass, but is made out of polystyrene [53].

3.2 Car assembly

The assembly of car's chassis consisted of the following steps:

- Prepare all the power and communication cablework.
- Solder switches, connector headers and the Teensy MCU to a breadboard. The switches enable to enforce manual control mode, that bypasses the Teensy board. The specific connections can be seen in Figure 3.1. The switching functionality is described in Section 3.6. The final module is on Figure 3.2. The resulting connections between components are depicted in Figure 3.1.
- Attach a Velcro fastener to the Energizer battery pack and the USB Hub to enable their easy removal from the chassis.
- Screw all the components in their place.
- Join the lower and upper level of the chassis by plastic standoffs.
- Stiffen back and front shock absorbers of the vehicle by inserts.
- Add front bumper made from acoustic foam to reduce risk of damage in case of an accident.

The assembled chassis with all components mounted can be seen from different angles in Figures 3.3, 3.4 and 3.5. Figures 3.6 and 3.7 provide detail of the LiDAR pad and power cable management.

3.3 NVIDIA Jetson TK1 setup

By following tutorials described in Section 2.2 I installed Ubuntu 14.04 and ROS Indigo onto an SD card. I also installed all the required driver packages.

I also installed the Grinch Linux kernel (see Section 2.4.2).

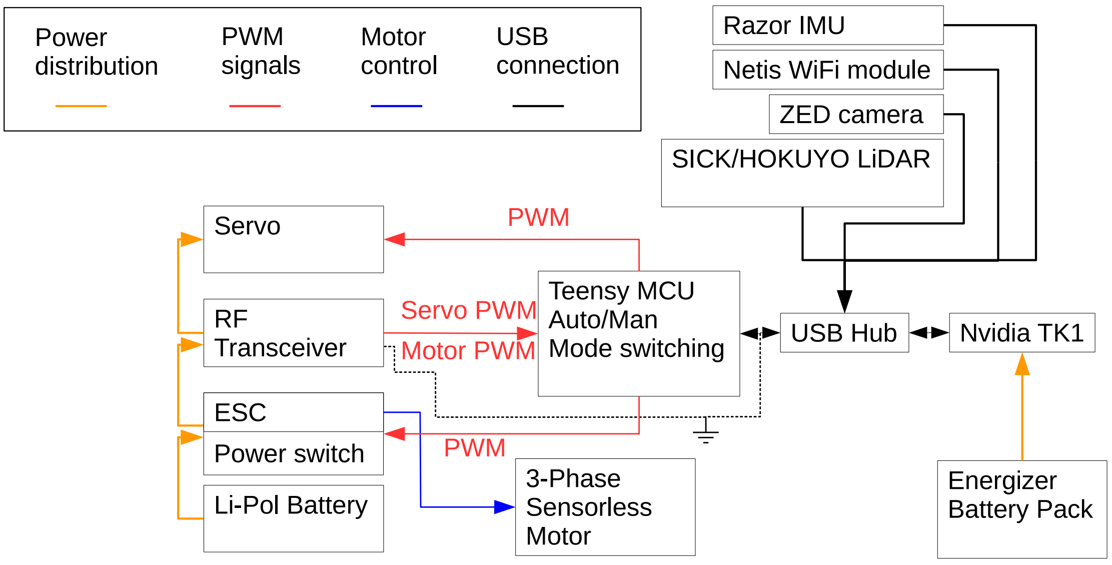

Figure 3.1. Connections between hardware components of the car.

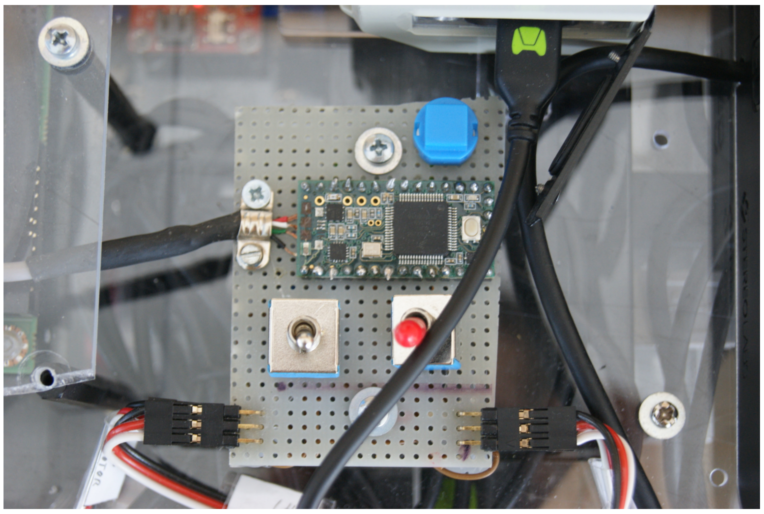

Figure 3.2. Picture of the Teensy module.

3.4 ZED Camera bring-up

The manufacturer provides proprietary binary ZED SDK [31] for the camera.

The ZED SDK requires an NVIDIA GPU and a fast CPU in order to work properly [43]. The ZED SDK initially supported the NVIDIA Jetson TK1 board, that we use, but since there are considerable problems with using this camera on the TK1 mainly related to insufficient performance of the TK1 (see [54, section "ZED SDK Supported Modes"] and [55]) the manufacturer recently dropped the support [31] for the TK1 and only supports the newer TX1 and TX2 boards starting from SDK 2.

Thus in order to use the ZED camera with TK1 an older version of the ZED SDK is required (version 1.2). The same applies to the "zed_ros_wrapper" ROS wrapper package [30].

After ZED SDK installation it is possible to test the camera by running "ZED Explorer" tool. This tool displays live images from both camera sensors.

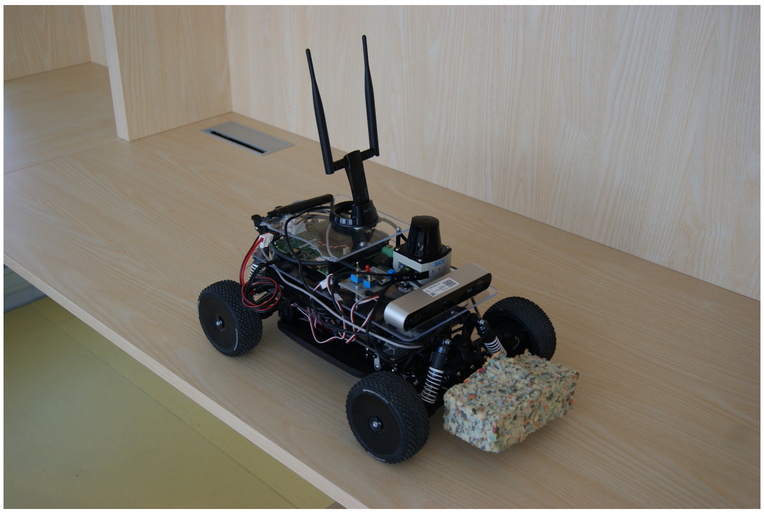

Figure 3.3. Overall view of the car.

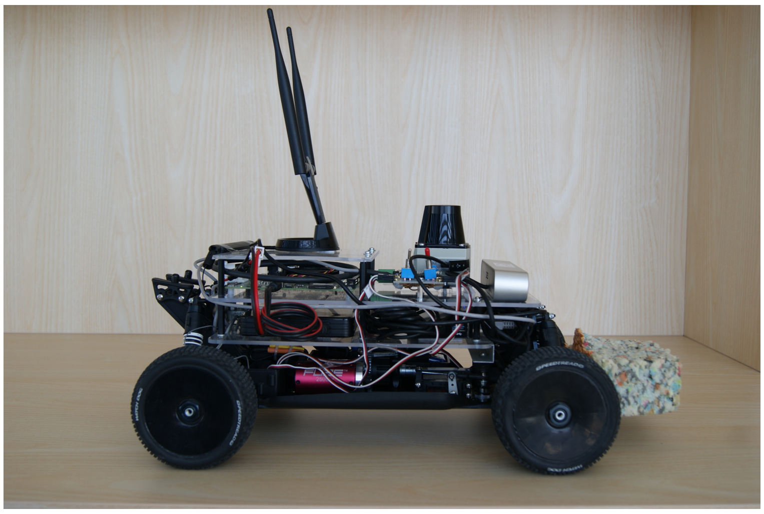

Figure 3.4. Picture of the right side of the car.

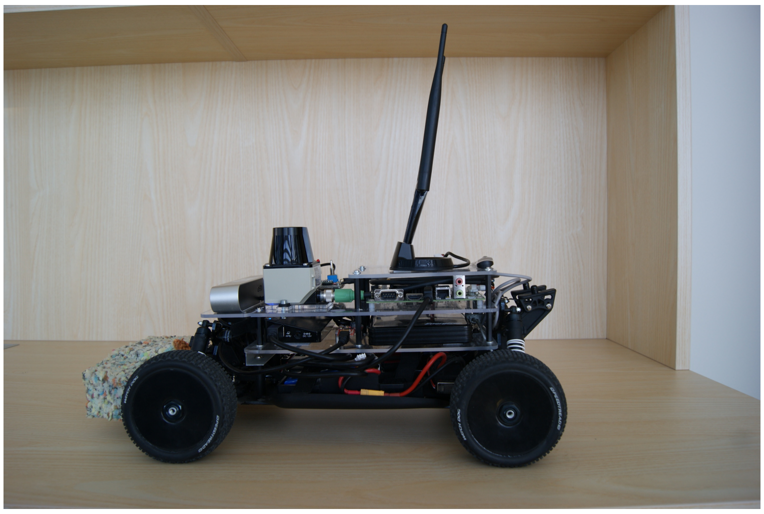

Figure 3.5. Picture of the left side of the car.

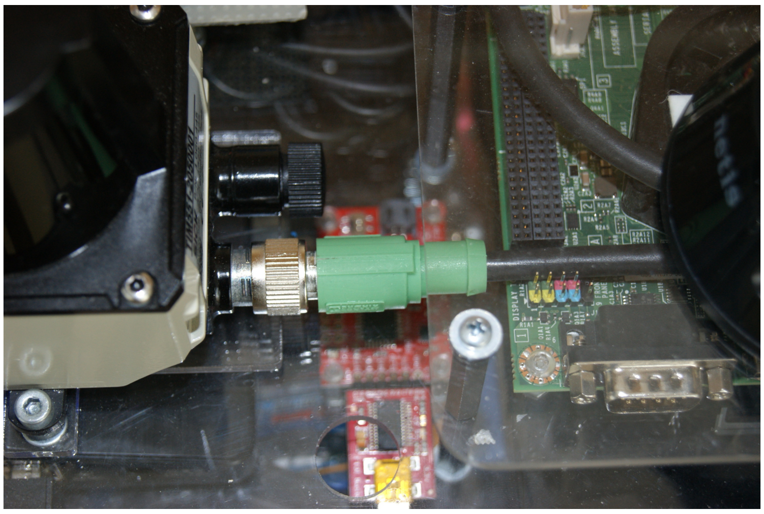

Figure 3.6. View from top of the SICK LiDAR power cable management.

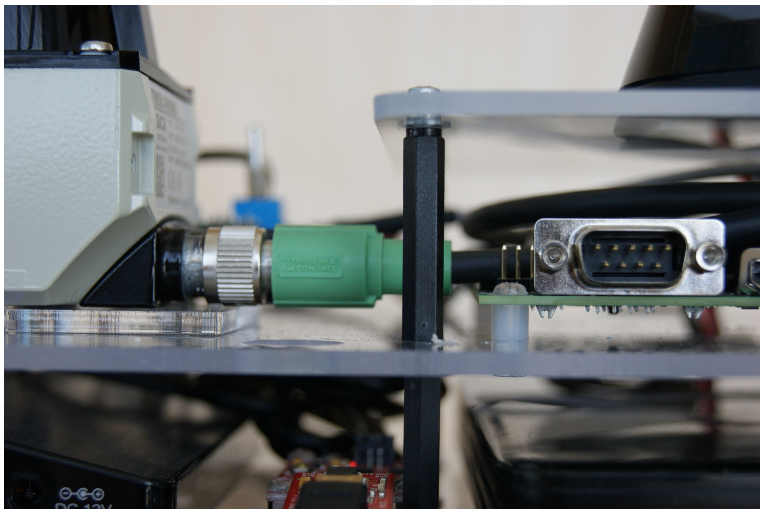

Figure 3.7. Level view of the SICK LiDAR power cable management.

Afterwards it is necessary to download the "zed_ros_wrapper" package and extract it into the catkin workspace and build it. The package contains "zed.launch" file designed to publish everything, the camera and SDK can provide. This includes odometry estimates, images for the left and right sensor, a depth map and a point cloud. The reliability of the provided odometry is discussed in Section 3.13.3.

3.5 Recording of rosbag files

During the testing of various ROS packages it was helpful to be able to record data from the experiments and perform manual tuning of specific configuration parameters without the need to repeat the experiments. Unfortunatelly, the amount of data generated per time unit is considerable. Especially if we want to record the camera images. The ROS provides the "compressed_image_transport" plugin[1]), that enables JPEG or PNG compression of the images, but this leads to loss of information in the data. Combined with the low resolution of the recorded images (see Section 2.4.1) this would degrade

[1]) `http://wiki.ros.org/compressed_image_transport`

the images and thwart any attempts to perform stereo odometry. Thus I had to find a way to record the data stream. At first I tried to transmit the data through the WiFi adapter and record the rosbag file on a computer. Unfortunately the WiFi is not that reliable and the data had to be re-transmitted and got recorded with wrong timestamps on the computer. The ROS allows to switch to UDP transport (see Section 2.3.1) to prevent this scenario, but it would lead to data loss, which is also undesirable. In order to solve this we use a high speed SD card capable of writes at up to 30 MB/s[1]).

This speed would still be insufficient, because the image data stream alone (considering 672 x 376 pixels per image resolution and 30 frames per second framerate) amounts to approximately 44 MB/s of data. To solve this we compress the data stream on-the-fly using the LZ4 algorithm (for details see[2])). This decreased the data stream to approximately 25 MB/s and the speed of the data card used is sufficient to handle this (peaks are covered by 1300 MB large memory buffer) and in order to decrease the number of disk writes we use 3 MB internal chunk size.

The implementation is the "competition" ROS package on the attached CD in file "record.launch".

3.6 Teensy firmware

We use the Teensy MCU to generate PWM signals to steer the servo and to command the ESC by means of a ROS topic. We also use it to capture the PWM signal received from the RF transceiver. We use this information for automatic switch from autonomous control to manual control mode and we also publish the resulting PWM duty cycle through a ROS topic.

The organizers provided a sample Teensy MCU firmware that used the "rosserial" interface in order to subscribe to PWM drive topic to generate the appropriate PWM signals for the ESC and the servo (see Section2.2). Unfortunately, the provided firmware did not offer functionality required by the rules. Specifically the ability to take over control with only the car's RF remote controller was not implemented. Therefore we had to change the firmware to allow for that.

In the following subsections we first provide some details about the CPU used on the Teensy module. Then we describe the firmware implementation.

3.6.1 Details of Kinetis K20 CPU

The Kinetis K20 CPU onboard Teensy features a FlexTimer module. This module is capable of running in multiple modes. We use only two:

- Edge-Aligned PWM mode – this is used to generate the desired PWM signals with specified frequency and duty cycle to command the servo and the ESC of the car. The mode is described in [57, sec. 36.4.6, case with ELSnB:ELSnA = 1:0].
- Input Capture mode – allows to detect an edge event on an input pin and record corresponding timestamp of the event and then issue an interrupt request to the NVIC interrupt controller. The mode is described in [57, sec. 36.4.4]. We use this mode for capturing the PWM values received from the RF transceiver of the remote control. The input capture features a channel input filter. This filter is used for debouncing/deglitching purposes. At present it is not used. It could be useful in

[1]) This value was determined using the GNU dd tool which is part of the GNU Coreutils [56]. The specific command used was: "dd if=/dev/zero of=test_dd oflag=nocache bs=1M count=1024 conv=fsync"
[2]) `http://wiki.ros.org/rosbag/Commandline#record`

case of increased noise in the captured PWM signals in case a new component was added to the car, which could induce parasitic voltage on the PWM signal loop and cause an edge event.

The CPU contains three FlexTimer modules. The PWM generation needs to be done on a different module than we use for the input capture mode, because the period of the PWM signal is determined by value in the MOD register. Since each FlexTimer module constains only a single timer, the MOD register value is shared by all channels of a single FlexTimer module (see [57, Figure 36-1]). The MOD register's value defines the internal counter overflow value. Since the counter has only 16 bits we do not want to loose any more bits in the input capture mode, which would be the case if we allow both of the required operations (PWM generation and input capture) on the same FlexTimer module. PWM generation is done on module FTM1 and input capture on module FTM0.

3.6.2 Firmware implementation

The implementation consists of a function used for FTM initialization setupFTM(). This function sets proper pinmux, resets the counter value, sets the counter overflow register, sets the status and control register (enable timer overflow interrupts, set clock prescaler factor to 1 and select the system clock as a clock source). Next the required channel interrupts are enabled (one channel for servo PWM and another for ESC PWM signal) and set to trigger on raising edge. Finally the interrupt is enabled in the NVIC.

The corresponding ISR first detects if the timer had overflown and then detects if an event actually occured on any of the two channels. If an event has occured, we first read the timestamp, handle possible overflow and set the channel interrupt to be triggered on either falling edge, if it previously triggered on rising, or on raising edge, if it previously triggered on falling edge. We then subtract the falling edge timestamp from the raising edge timestamp obtaining Δt and then calculate a scaled duty cycle using the following formula

$$scaled\_duty\_cycle = \frac{f_{PWM}\ \Delta t}{f_{BUS}}\ k_{PWM\_GEN},$$

where $f_{PWM} = 91\,\text{Hz}$ is the frequency of the PWM signal, $f_{BUS} = 48\,\text{MHz}$ is the frequnecy of the bus clock to which the timer's counter is connected and $k_{PWM\_GEN} = 65535$ is scaling factor that represents the range of values accepted by the analogWrite() function in order to generate the corresponding PWM signal.

In order to determine the f_{PWM} I measured the period of the respective signal by an oscilloscope.

If any of the computed duty cycles were out of a predefined tolerance band, we switch to manual mode and set the calculated PWM duty cycle to apropriate counter value register used for PWM generation on the other FlexTimer module. The ISR also blinks the onboard LED when the manual mode is active.

The manual mode can be switched back to ROS controlled mode by pushing blue button on the Teensy housing breadboard. The button is connected to an input pin, that is pulled-up by an internal resistor, and to the ground pin of Teensy. The pin is periodically polled for current state and 10 ms debouncing is used (Bounce class is used for this[1])).

Another functionality that I implemented in the firmware is timeout for PWM ROS topic reception. In case no ROS message on the topic specifying the required PWM

[1]) `https://www.pjrc.com/teensy/td_libs_Bounce.html`

duty cycles is received within 300 ms of the last message. The ESC is commanded to neutral effectively causing brake to engage (see Section 2.4.4 describing ESC modes of operation) and servo to center value.

I also implemented publication of the computed PWM duty cycles captured from the RF transceiver. This was useful during the identification experiments (see Section 3.10).

Here we list the board pin numbers used (for their respective CPU pins see Figure 2.2):

- 3 – output PWM signal for the servo (FTM1, channel 0)
- 4 – output PWM signal for the ESC (FTM1, channel 1)
- 8 – manual to autonomous mode switch button
- 13 – onboard LED
- 20 – input PWM signal (from the RF transceiver) for the servo (FTM0, channel 5)
- 23 – input PWM signal (from the RF transceiver) for the ESC (FTM0, channel 1)

The implementation is in file ”teensy_drive.ino“ on the attached CD.

3.7 IMU firmware

The ROS ’razor_imu_9dof’ package contains a firmware for the ATmega328 MCU onboard the IMU. The firmware is used to read data from the sensors onboard the IMU and to transmit them over the USB to the host.

The tutorial published by the organizers [58] is misleading in that it does not mention the need to flash a new firmware to the IMU, but the ROS wrapper will not work properly without doing so.

I had to make some changes to this firmware:

- Use the z gyro reference instead of the internal oscillator as the device clock source. According to [59, Section 8.7]: ”On power up, the ITG-3200 defaults to the internal oscillator. It is highly recommended that the device is configured to use one of the gyros (or an external clock) as the clock reference, due to the improved stability.“ However, in the provided firmware this was not done.
- Set proper model of our accelerometer. The firmware supports multiple types so we need to select the correct one (SEN-10736).
- Swap axis orientation of the sensors due to the module mounting on the car.

The firmware is written in the form of an Arduino IDE sketch. The Arduino IDE is used to download compiled firmware to the MCU. The firmware can be found on the attached CD.

I also calibrated the onboard sensors by following the procedure in [29].

After mounting onto the chassis and recording a few rosbags I noticed the data had been published with wrong axis orientation. After looking at the ROS wrapper part I discovered, that it does another transformation of the reference system. In order to save myself work with re-recording rosbag files I wrote a ROS node to handle the transformation of the reference system separately and publish the fixed data to a separate topic ”rotated_imu“ instead of the original ”imu“. This could be also useful if the location of the sensor changed in the future.

3.8 Testing of publication of the required sensor data

In order to verify, that all ROS wrappers and drivers for the various sensors work properly and publish the required data, I wrote ”publish_all.launch“ file. This launch file brings up the ROS driver for the SICK LiDAR, Teensy board (rosserial node) and the Razor IMU. The launch file is on the attached CD in the “competition” package.

3.9 Unscented Kalman Filter

In order to fuse the resulting pose from the Adaptive Monte-Carlo Localization (AMCL) node with the IMU measurements, we use the Unscented Kalman Filter from the “robot_localization” ROS pakage (for the package description, see Section 2.3.2). The filter provides the estimated car's position in the 2D map (x and y coordinates and heading) and both angular and linear velocities of the car.

The kinematic model used in the UKF uses constant acceleration model in all three axes. Thus for the v_y component of linear velocity we would get

$$v_y = a_y t,$$

where a_y is linear acceleration in the y-axis direction and t is time.

Instead of the y axis component of the linear acceleration we measure the centripetal acceleration a_d caused by the rotational motion of the car. Its equation is given by

$$a_d = \omega^2 r,$$

where ω is the angular velocity of the car and r is the turning radius. This acceleration, however, does not relate to translational motion it is only caused by rotational motion, so we cannot use it to compute linear velocity estimates of the UKF. Thus we only fuse the x component of the accelerometer measurements.

We use this filter in a so called 2D mode. This mode will cause the filter to ”fuse 0 values for all 3D variables (Z, roll, pitch, and their respective velocities and accelerations). This keeps the covariances for those values from exploding while ensuring that your robot's state estimate remains affixed to the X–Y plane“ [38]. The system model equations used by the UKF can be found in[1]).

Due to a bug affecting the magnetometer (see Section 3.11.3), we fuse only the measurements obtained from the gyroscope and accelerometer.

3.10 Car identification

In this section we describe the experiments, that I carried out in order to get a basic idea of the car's kinematics. The objectives were to identify the steering mechanism and powertrain properties.

Identification of the car dynamic properties was not in the scope of this thesis. It was the topic of another member of the team, but unfortunately it was not done yet.

[1]) https://github.com/cra-ros-pkg/robot_localization/blob/indigo-devel/src/ukf.cpp

3.10.1 Steering mechanism identification

In this section we describe the experiment used for the identification of the steering mechanism which consists of a servo and Ackermann steering geometry. What is of importance to us is the relationship between the PWM duty cycle sent to the servo and the resulting turning radius. By turning radius we mean a positive or negative distance in meters, that represents a circle circumscribed around the vehicle when turning. The sign of the turning radius represents the direction of a rotation. It could be either counterclockwise corresponding to positive turning radius or clockwise corresponding to negative turning radius.

The data for the identification were obtained in two ways. The first way consisted of driving in a circle with servo being commanded to turn by given PWM duty cycle. The linear velocity of the car was given by another PWM duty cycle value and the corresponding linear velocity was low (approximately $1\,\mathrm{m.s^{-1}}$) in order to maintain rolling motion between the wheels and the floor and to eliminate sliding. Then the turning radius was measured using a tape measure.

The second way involved obtaining data from the UKF estimates of angular velocity and linear velocity. Dividing the linear velocity by angular velocity we obtain the resulting turning radius corresponding to applied PWM duty cycle to the servo. This method was used to obtain values for larger absolute values of turning radius where the tape measure was short.

In order to avoid infinite radius during straight motion, in the graphs below we show the reciprocal value of turning radius – curvature. The resulting graph depicting the identification results is in Figure 3.8. Final PWM duty cycle can be calculated as follows

$$PWM = 13.1375 + 2.8752\,\frac{1}{r},$$

where $\frac{1}{r}$ represents the curvature and PWM represents the corresponding PWM duty cycle value.

The minimum turning radius is approximately 1 m. This value has changed over time due to various accidents the car has experienced. The maximum measured change was 4 %.

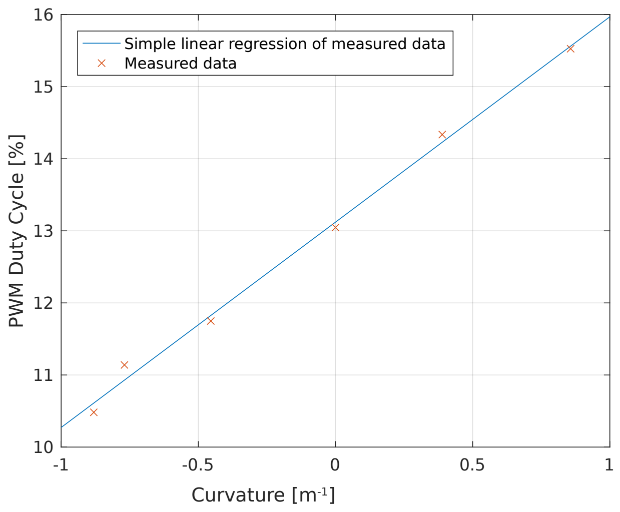

Figure 3.8. Steering mechanism identification results.

3.10.2 Powertrain identification

In this section we describe the experiment used for the identification of the powertrain. Of importance to us is to get a basic understanding of relationship between PWM duty cycle applied to the ESC and the resulting linear velocity when driving straight. We used two ways to measure the resulting linear velocity, that we describe in the following sections.

The linear speed also depends on the charge level of the Li-Pol battery. If the level drops then the measured speeds no longer match the identified ones for a given PWM duty cycle value.

The maximum linear velocity of the vehicle was not determined, because the RF remote control loses connection with the car, when it is further away than 60 m and when this happens the car, unfortunatelly, does not stop, but continues in motion. This is a bug in the supplied RF transceiver.

3.10.2.1 Measuring the travelled distance

The first way of powertrain identification consists of measuring the distance travelled by the car in a fixed amount of time.

To implement this we use a ROS node that publishes fixed PWM duty cycle for the ESC and also steers the car to follow the wall. The ESC PWM duty cycle is dynamically configurable (see Section 2.3.1).

When the specified amount of time elapses, the ESC duty cycle is set to neutral value causing braking (see ESC modes in Section 2.4.4).

When the car stops moving we measure the travelled distance. We repeate this procedure for a number of ESC PWM duty cycle values.

The implementation of the wall following part consists of determining the car's distance to the wall and a PD controller to maintain the distance constant. The distance from the wall is calculated by line fitting all the LiDAR scan beams in -112.5 – -67.5 degrees, relative to car heading, using the simple linear regression.

The obtained measurements and corresponding velocities are in Table 3.1.

This experiment provides us only rough estimate of the kinematic properties, because the braking distance increased with the speed as the applied PWM duty cycle increased.

The implementation can be rfound on the attached CD in file “experiment1.py” in the package “my_race”.

PWM Duty Cycle [%]	time [s]	distance travelled [m]	velocity [m.s^{-1}]
14.65	3	12	4
14.50	3	11	3.67
14.34	3	8	2.67
14.19	3	6	2

Table 3.1. Powertrain identification by measuring the travelled distance.

3.10.2.2 Using estimate from the UKF

In order to get better results we again use estimates obtained from the UKF. This time we use the linear velocity estimate. During the experiment we drive straight and maintain constant PWM duty cycle sent to the ESC. We consider only the measurements taken when the speed allowed pure rolling motion of the wheels. The results of the identification can be seen in Figure 3.9. Final PWM duty cycle value can be calculated as follows

$$PWM = 13.2267 + 0.5742\ v,$$

where v represents the linear velocity and PWM represents the corresponding PWM duty cycle value.

There is a deadband around the centre PWM duty cycle value. To cross the deadband threshold and start a movement of the car it is necessary to apply a higher PWM duty cycle corresponding to velocity of approximately $0.7\ \mathrm{m.s}^{-1}$.

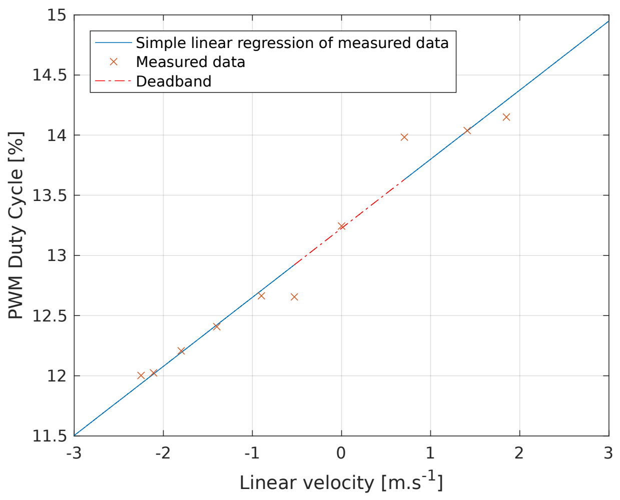

Figure 3.9. Powertrain identification results.

3.10.3 Discussion of relevance of obtained results

The identification results are valid only for small speeds at which we perform subsequent testing of the ROS application software. There are numerous reasons for this:

- Limited sampling frequency of our LiDAR. The SICK LiDAR is only capable of providing measurements at 15 Hz and the described wall following controller computes steering commands at the same rate. This means, that if we, for example, move at the velocity $15\,\mathrm{m.s}^{-1}$ we would compute a new steering command only after traversing the distance of 1 m. This way we could easily hit the wall.
- The floor of our laboratory is very slippery and when moving at higher velocities, we cannot guarantee pure rolling motion. Since we have neither the dynamic model of the car nor do we know the friction coeficient between the tires and the surface, the obtained data would be useless.

The results for steering mechanism identifiction could be used to directly command certain turning radius. However, the powertrain identification is valid only with fully charged batteries and the car moving on a level surface. In order for the car to move at a predefined linear velocity, it would be necessary to implement a linear velocity controller, that would use the information about current linear velocity estimate from the Unscented Kalman Filter (see Section 3.9 for details about the UKF) to close the feedback loop. This is not done yet.

In order to get a better idea how the car behaves when moving at high linear and angular velocities it would be necessary to identify also the car dynamics. This was left for future work as mentioned in Chapter 5.

3.11 IMU experiments

In this section we describe experiments performed in order to discover how the IMU readings are affected by motor vibrations during car's movement and if we need to update the measurement noise covariance matrix for the Unscented Kalman Filter to compensate for the vibrations. Next we also discuss the reliability of the magnetometer in an indoor environment.

3.11.1 Accelerometer measurements and tuning of the measurement noise covariance matrix

Since we are only moving in a plane, we analyze the role of only the x and y components of the acceleration vector. The x component represents acceleration of the car caused by the motor. The y component represents centripetal acceleration related to circular motion in turns.

The noise variance of the measurements is, unfortunately, increased by vibrations from the motor as can be seen on Figure 3.10 depicting the measured accelerometer data (the gravitational acceleration is subtracted). The data are used in the following experiments in this subsection.

The original measurement noise covariance matrix, that was part of the source code for "razor_imu_9dof" package is the following

$$R_0 = \begin{pmatrix} 0.04 & 0 & 0 \\ 0 & 0.04 & 0 \\ 0 & 0 & 0.04 \end{pmatrix}.$$

. Velocity estimation with this covariance matrix is in Figure 3.13. In order to improve the UKF behavior we compute a covariance matrix of 50 measurements when moving straight ahead with constant linear velocity. Then we use this new covariance matrix in the UKF. The covariance matrix is the following

$$R_1 = \begin{pmatrix} 0.227 & 0.100 & -0.032 \\ 0.100 & 0.250 & -0.061 \\ -0.032 & -0.061 & 0.380 \end{pmatrix}.$$

. The resulting linear velocity estimates are in Figure 3.12.

For comparison, Figure 3.11 shows linear velocity estimates from UKF fusing only the robot's pose obtained from the AMCL and gyroscope measurements. As can be seen by comparing Figure 3.11 and 3.12 the fusion of measured acceleration in the UKF improved the velocity estimate during braking maneuver. When not fusing the measured acceleration, the estimated linear velocity converges to zero very slowly after the vehicle stops moving.

In order to investigate the improvement further, we zoom in on the braking part of the velocity estimate (Figure 3.14), acceleration plot (Figure 3.15) and the control inputs to the car (Figure 3.16).

We want to verify, that the linear velocity estimate of the car in the forward direction (represented by the x axis) starts dropping when we begin the braking maneuver and that the velocity estimate reaches zero by the time the car stops moving. We assume that the car reaches full stop when the measured deceleration reaches zero. The applied deceleration is constant (see Figure 3.15), therefore we also want to check, that the estimated velocity drops linearly during the braking maneuver.

In Figure 3.16 we can see, that the car engages brakes at 14.9 s as the ESC input PWM duty cycle drops to neutral and ceases to be published. Next in Figure 3.15

we can see, that a constant deceleration is measured at around 15.1 s. This confirms, that the car started braking. We can also see, that the deceleration reached zero at approximately 16.3 s.

Next we look at Figure 3.14 depicting the linear velocity estimate. We can see, that the estimated linear velocity starts to drop linearly at the same time the deceleration occured. We can also see, that it reaches approximately zero at the same time the deceleration reached zero. This is in accordance with the expectations.

Next, we also want to verify, that the newly computed measurement noise covariance matrix really improved the linear velocity estimate. We do this by comparing Figure 3.12 and 3.13. We can see, that the linear velocity estimate obtained by using the new measurement noise covariance matrix approaches zero more quickly, than the original covariance matrix.

Thus we conclude that the best estimate of the linear velocity is obtained from the UKF when we use the newly constructed measurement noise covariance matrix. This was an important test to verify correct behavior of the Unscented Kalman Filter.

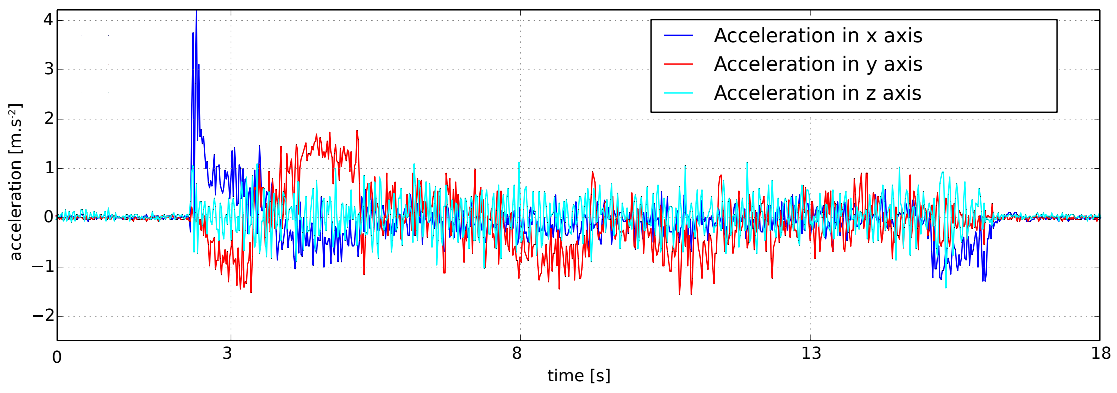

Figure 3.10. Illustrative accelerometer measurements.

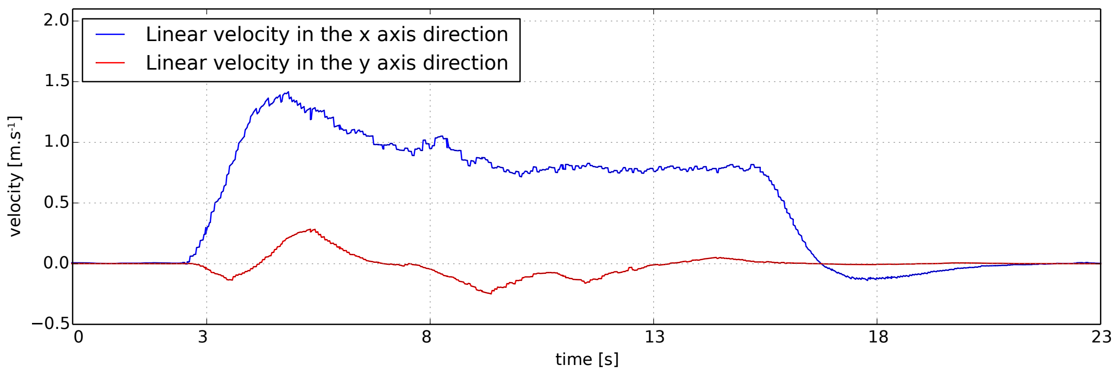

Figure 3.11. Velocity estimate using only AMCL and gyroscope.

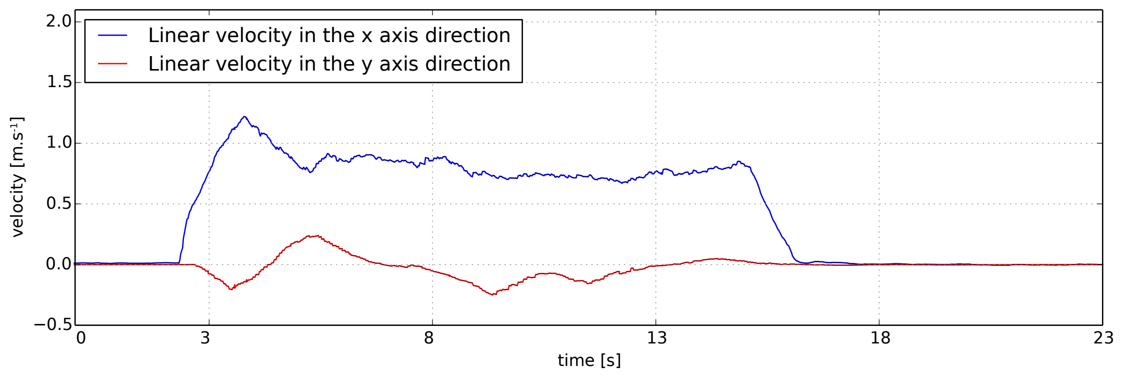

Figure 3.12. Velocity estimate using AMCL, gyroscope and accelerometer data with corrected measurement noise covariance matrix.

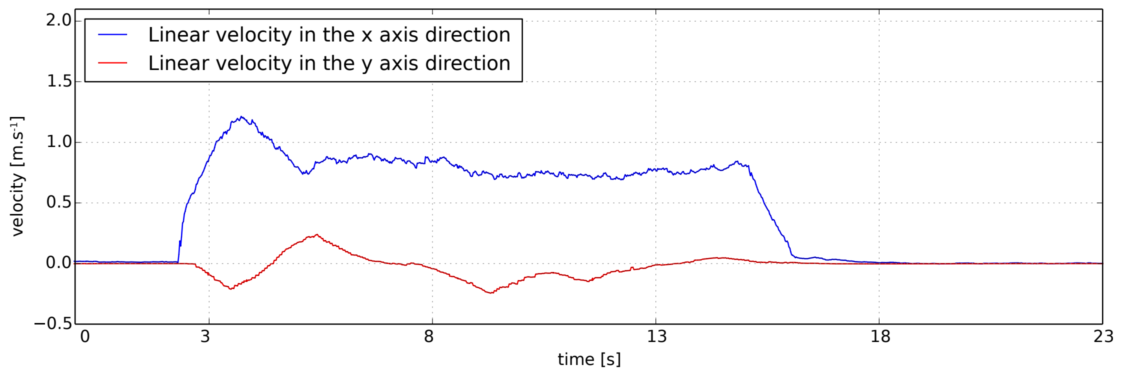

Figure 3.13. Velocity estimate using AMCL, gyroscope and accelerometer data with the original measurement noise covariance matrix.

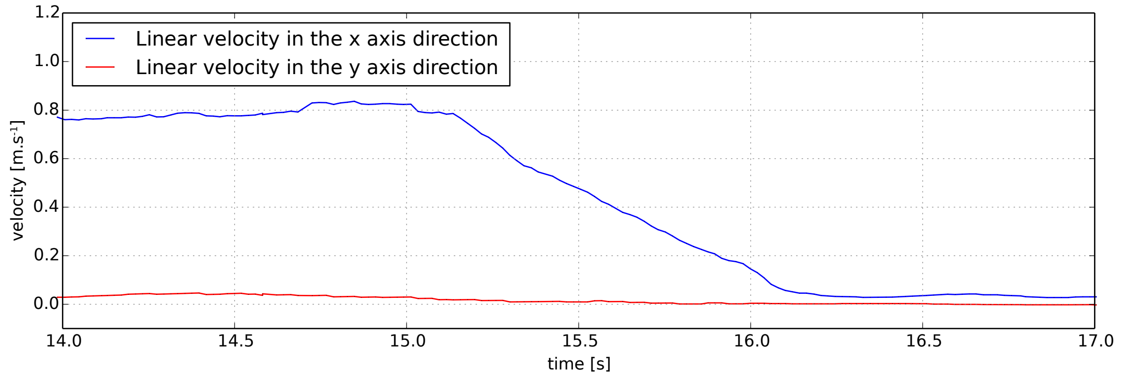

Figure 3.14. Linear velocity estimate during braking maneuver.

3.11.2 Gyroscope measurements

We want to determine the effect of motor vibrations on the measured angular velocity. Should the effect be significant, we need to update the measurement noise covariance matrix accordingly. We also want to determine the behavior of the UKF when we do not fuse the gyroscope measurements, but we fuse only the position estimate obtained from the AMCL and the accelerometer measurements.

Since the car is only moving in a plane, we consider only the z component of the angular velocity vector.

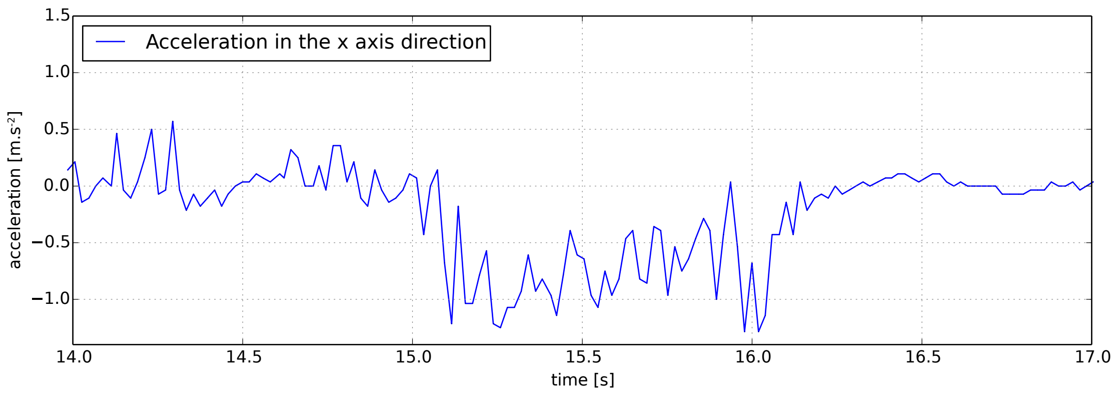

Figure 3.15. Measured linear acceleration during braking maneuver.

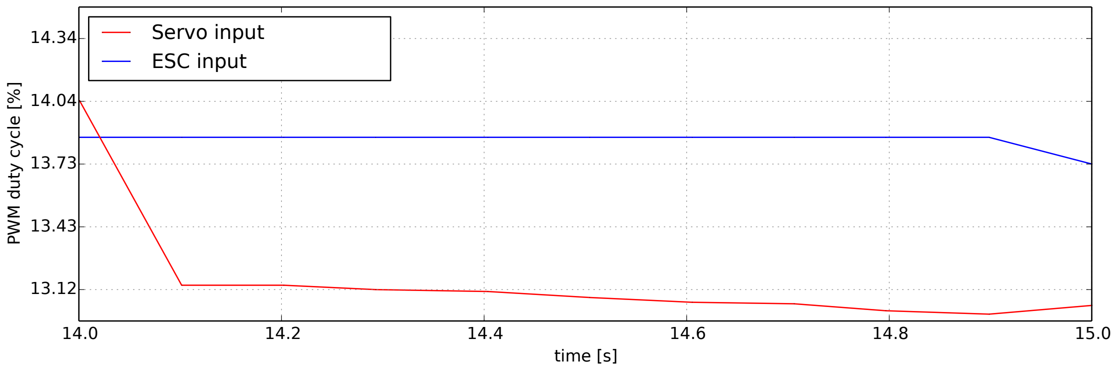

Figure 3.16. Control input during braking maneuver.

In Figure 3.17 we can see the measured angular velocity. The measured data are from the same run as the data used in Section 3.11.1 for evaluating the accelerometer. As can be seen on the figure the variance of the measurements is not drastically increased, so no additional tuning of measurement noise covariance matrix is necessary in this case. To verify, that this is indeed so, we compare the measured values and the obtained estimates from the UKF (see Figure 3.18). We can see, that the covariance of the obtained estimate is smaller than, the that of the measurements.

Figure 3.18 demonstrates the behavior of Kalman filter with fused gyroscope data and without them. It can be seen, that even without measuring the angular velocity the estimate converges to the same value as the measurement, only slower. This was also an important test to verify correct behavior of the Unscented Kalman Filter.

3.11.3 Magnetometer usability

The magnetometer is not used at all, because a known bug demonstrated itself too often for the readings to be useful.

“The magnetometer becomes “locked”, resulting in the heading always drifting/converging back to the same angle. Resetting the microcontroller which runs the Razor AHRS firmware does not help in this case (so it’s not a firmware bug!), but turning power off and on again should do the trick. I think this must be a bug inside the magnetometer“ [29].

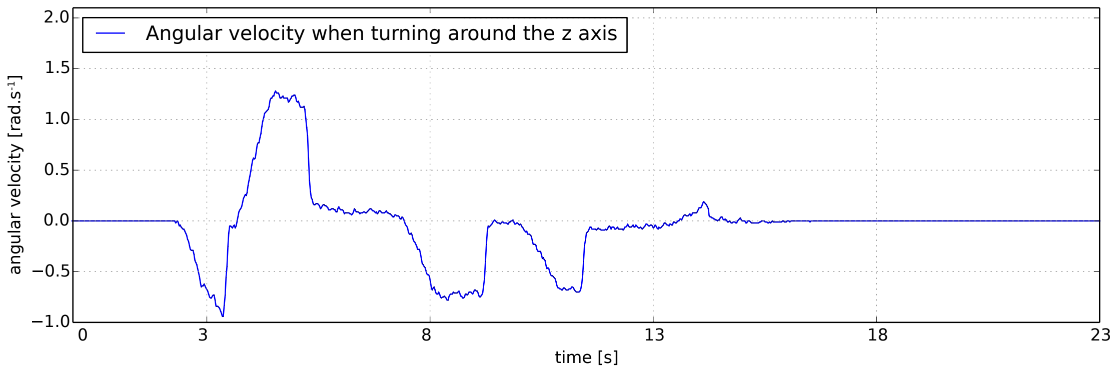

Figure 3.17. Angular velocity measurements from the gyroscope.

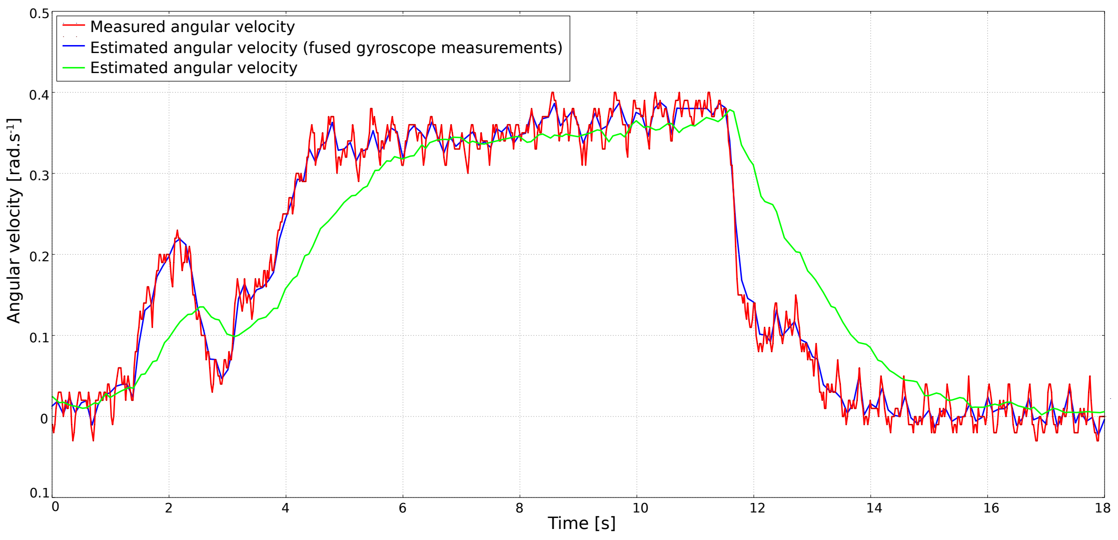

Figure 3.18. Angular velocity estimate of the UKF.

3.12 Testing of 2D SLAM ROS packages

The Simultaneous Localization and Mapping is a task where a map of the environment is constructed simultaneously with determining the car's position in it.

In this section we first briefly introduce the tested packages and then we look at how they perform in a specific environment (our laboratory). All the respective figures in the following sections were created from the same data (the same rosbag file) to allow for comparison.

3.12.1 CRSM SLAM

The algorithm uses a ray-selection method and Random Restart Hill Climbing to match the scan to the map. In order for the map to be usable it needs thousands of Hill Climbing iterations and is therefore computationally demanding and working configuration needs around 80 % of CPU time when executed on the PC (equipped with Intel Core i5-6200U CPU). The implementation can utilize at most one CPU core only. The used algorithm is described in [60]. The resulting map of our laboratory is in Figure 3.19. For the construction of the map we use the "crsm_slam_real.launch" file from the package "my_race", that can be found on the attached CD.

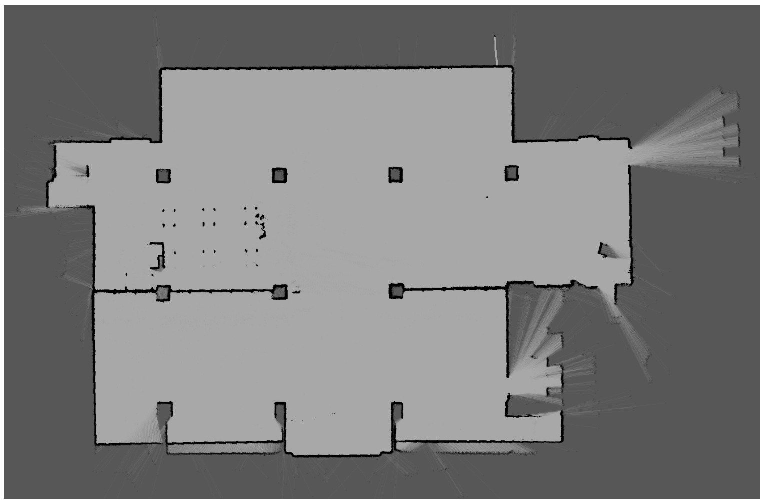

Figure 3.19. Map of a laboratory generated by the CRSM SLAM package.

3.12.2 Hector SLAM

Uses gradient-based method to match scans to the map learnt so far. Unlike the CRSM SLAM method, it does not perform Hill Climbing by random perturbations of the current car's coordinates, but rather uses gradient-based algorithm. For more details, see [61]. This approach requires an approximation for the map gradient which is also described in [61]. The resulting map of our laboratory is in Figure 3.20. For the contruction of the map we use the "demonstrations_and_mapping.launch" file from the package "competition", that can be found on the attached CD.

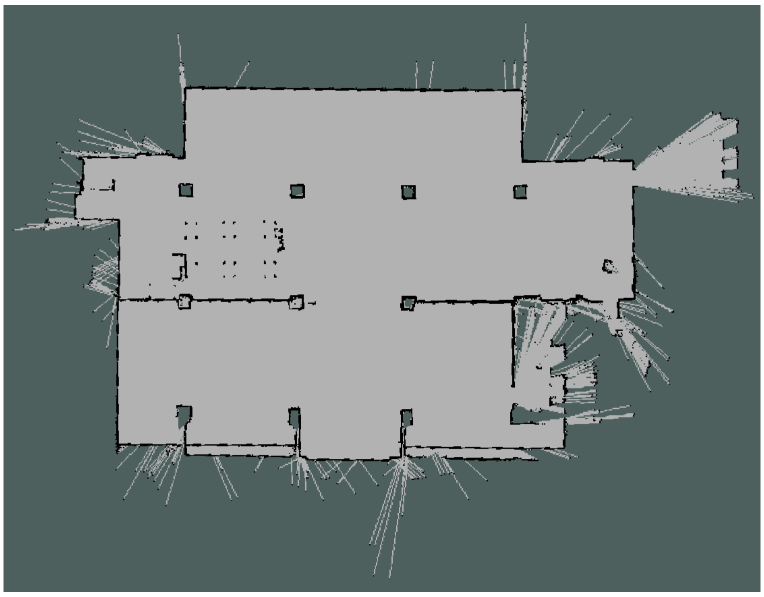

Figure 3.20. Map of a laboratory generated by the Hector SLAM package.

3.12.3 Gmapping

The algorithm uses a particle filter and known odometry to construct the occupancy grid map of the environment. Gmapping requires relatively good odometry estimates. Unfortunately, the car is not equipped with wheel odometry, so instead we used the odometry estimate of the "laser_scan_matcher" node (uses ICP algorithm, for details see Section 3.14). This approach leads to failure of Gmapping in long corridors, where the ICP gets lost. The implementation is described in [62]. The resulting map of our laboratory is in Figure 3.21. For the construction of the map we use the "demo_gmapping.launch" file from the package "my_race", that can be found on the attached CD.

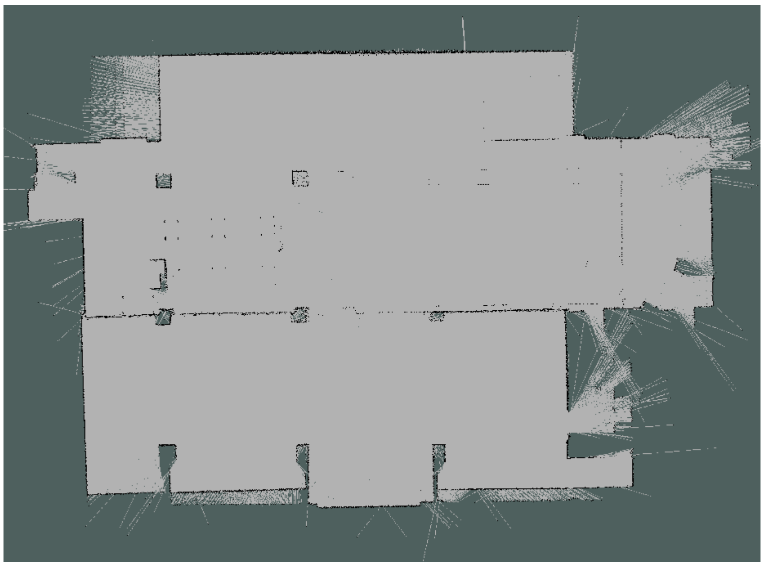

Figure 3.21. Map of a laboratory generated by the Gmapping package.

3.12.4 Selected package

We select the Hector SLAM to perform the 2D SLAM, because it produces good quality maps and needs the least computational power. Thus it can also run directly on the TK1.

3.13 Testing of stereo odometry ROS packages

In this section we first briefly introduce the tested packages and then we look at how they perform in an indoor environment. Specifically in an office and a corridor. All the mentioned packages use only images captured by the ZED camera and calibration matrices computed by the ZED SDK. The images used are rectified by the ZED SDK. All packages are tested against the same data from a single test run. A 2D occupancy grid map of the environment is put into corresponding figures for better clarity (except for the ORB SLAM – it does not publish its odometry over the ROS odometry topic). The starting position of the car in the map was measured by a tape measure. Figures 3.22 and 3.23 show the location, where the testing was performed.

3.13.1 libviso2

"LIBVISO2 (Library for Visual Odometry 2) is a very fast cross-platfrom (Linux, Windows) C++ library with MATLAB wrappers for computing the 6 DOF motion of a moving mono/stereo camera. The stereo version is based on minimizing the reprojection error of sparse feature matches and is rather general (no motion model or setup

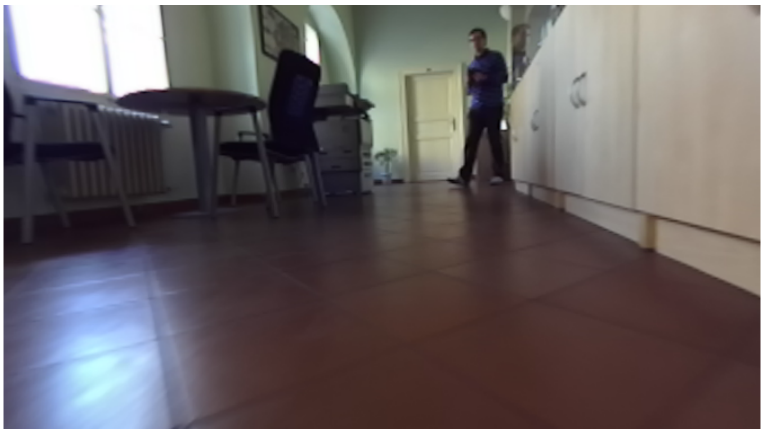

Figure 3.22. An illustrative ZED camera image captured on the corridor.

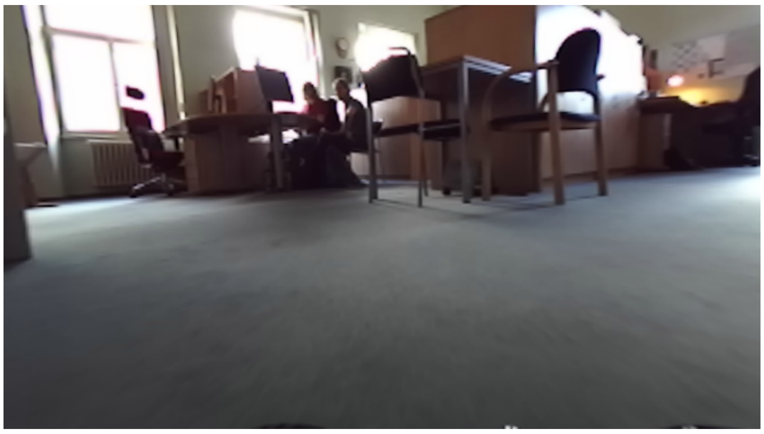

Figure 3.23. An illustrative ZED camera image captured in the office.

restrictions except that the input images must be rectified and calibration parameters are known)" [63].

More detailed description of the used algorithm can be found in [64]. The used ROS wrapper is documented in [65].

The resulting odometry estimate can be seen in Figures 3.24, 3.25. The estimate is unsatisfactory, because most of the time the algorithm estimates the car to be outside of the environment where it moved. Sometimes it even estimates that the car is a few meters below the ground.

The respective ROS launch file used for the testing is "demo.launch", that is part of the ROS wrapper.

3.13.2 ORB SLAM2

According to its authors [66]: "ORB-SLAM2 is a real-time SLAM library for Monocular, Stereo and RGB-D cameras that computes the camera trajectory and a sparse 3D reconstruction (in the stereo and RGB-D case with true scale). It is able to detect loops and relocalize the camera in real time."

The ROS package source codes are available in [66]. The algorithm itself is described in [67].

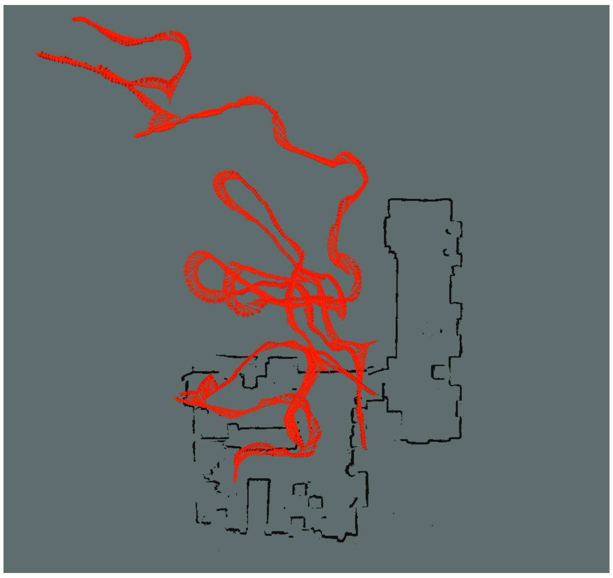

Figure 3.24. Top view of the odometry output of the libviso2 library.

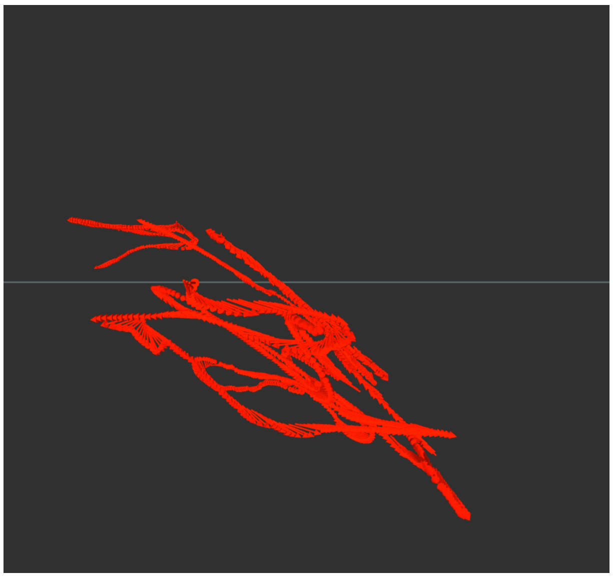

Figure 3.25. Horizontal view of the odometry output of the libviso2 library.

In order to obtain odometry estimates we run the "Stereo" node, which is a part of the ROS package, and use custom provided configuration. The configuration file can be found on the attached CD (ZED.yaml).

The resulting odometry estimate can be seen in Figures 3.26 and 3.27. The estimate obtained is unsatisfactory, because the implementation loses track of the position of the car almost every turn.

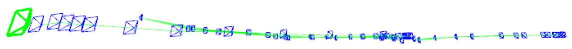

Figure 3.27. Horizontal view of the odometry output of the ORB-SLAM2 library.

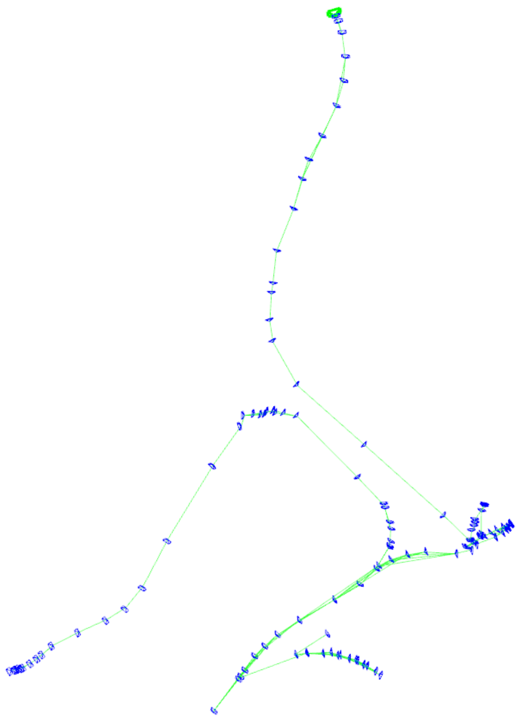

Figure 3.26. Top view of the odometry output of the ORB-SLAM2 library.

3.13.3 ZED SDK

The ZED SDK mentioned in Sections 2.3.2, 3.4 includes a proprietary algorithm for visual odometry. The resulting odometry estimate can be seen in Figures 3.28 and 3.29.

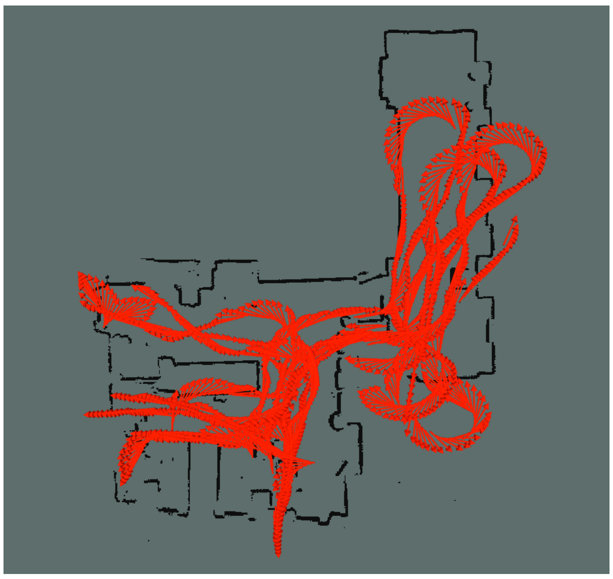

Figure 3.28. Top view of the odometry output of the ZED SDK 1.2.

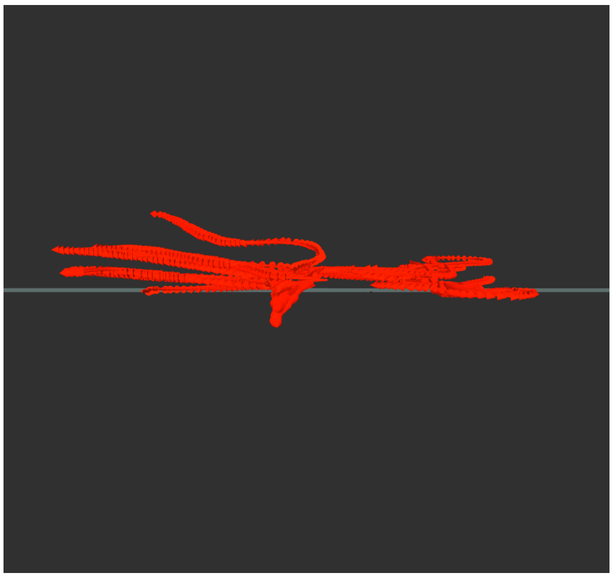

Figure 3.29. Horizontal view of the odometry output of the ZED SDK 1.2.

The obtained odometry estimates proved to be the best of all tested stereo odometry packages and thus it was tested in a more demanding environment representing the worst-case scenario with very low number of visual markers (see Figure 3.30). The resulting odometry estimate is in Figure 3.31. As could be seen from the figure even this package was not able to produce a good odometry estimate.

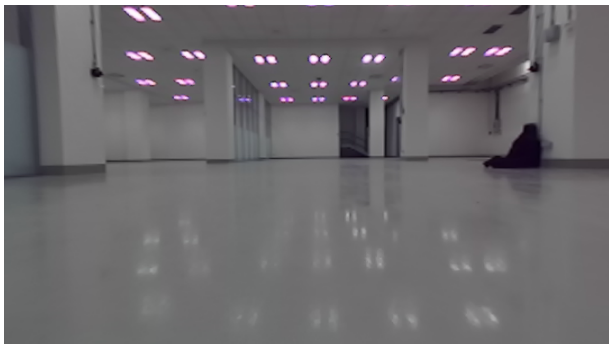

Figure 3.30. An illustrative picture of the laboratory for ZED odometry testing.

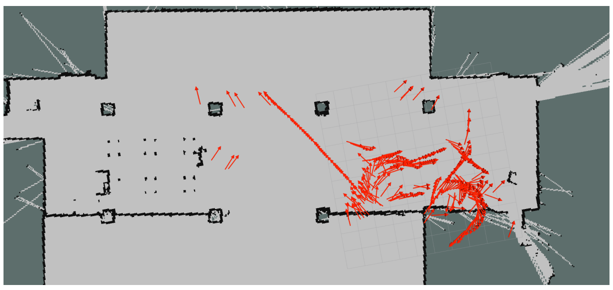

Figure 3.31. Top view of the odometry output of the ZED SDK 1.2 when tested under more challenging conditions.

In order to obtain odometry we use the "zed.launch" file, which is part of the ZED ROS wrapper.

3.13.4 Discussion of the obtained results

As was experimentally confirmed, the environment of the competition is a very demanding one for the current stereo odometry algorithms and the odometry is basically useless. Thus the final architecture (see Section 3.17) does not use the stereo odometry estimates at all. Instead we rely on LiDAR and map of the environment constructed using 2D SLAM (see Section 3.12) and Adaptive Monte-Carlo Localization (see Section 3.14).

3.14 Localization in a known map

Using the map constructed using the Hector SLAM we would like to be able to produce a position estimate of the car. In order to accomplish this we use the AMCL ROS package. According to [68] this node "is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox [69]), which uses a particle filter to track the pose of a robot against a known map."

The AMCL node subscribes to LiDAR scans, odometry estimate and to a known map. It's output is the estimate of the car's position and orientation in the map. The estimated position and orientation is then fused inside the Unscented Kalman Filter with the IMU measurements as described in Section 3.9.

Since the car is not equipped with a wheel odometry, we have to use the "laser_scan_matcher" node which uses the Iterative Closest Point algorithm to estimate car's movement. This node takes the laser scans and the measured angular velocity of the car and produces estimated position with respect to car's initial position when the node was launched. We use the angular velocity to improve detection of rotation movement of the car. Details about the algorithm can be found in [70].

Trajectory estimate of the car during one of test runs can be seen in Figure 3.32.

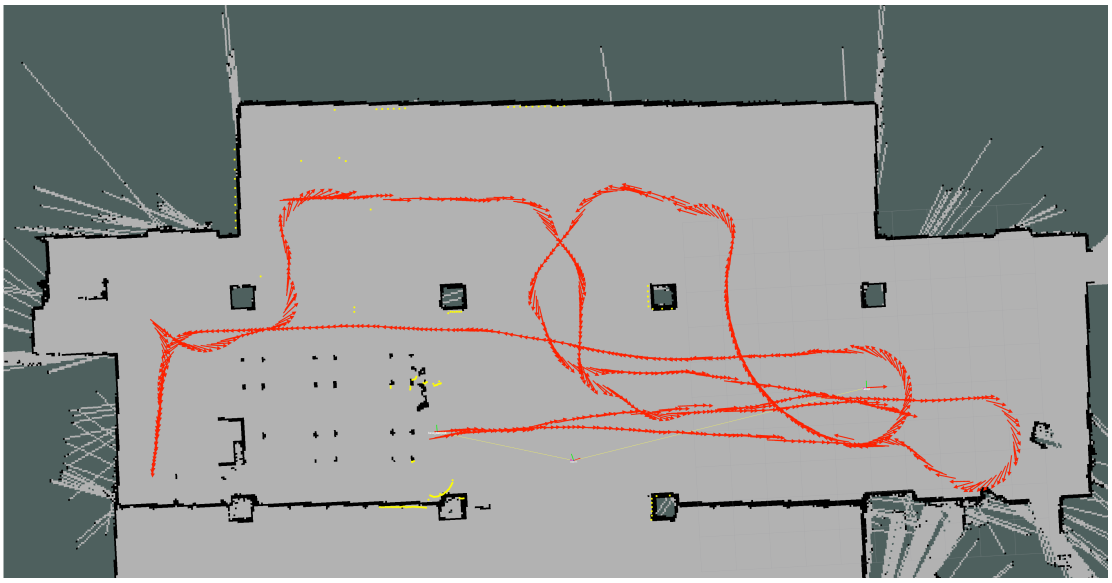

Figure 3.32. Trajectory estimation in known map.

The AMCL is capable of utilizing at most one CPU core. In order to use the AMCL directly on the TK1, we had to decrease the maximum number of particles in the particle filter to 1300. The solution works well in our laboratory, but in other environments the limitation on the maximum number of particles may prove problematic.

3.15 Local Planner

In this section we describe the developed ROS local planner plugin “trgen_local_planner” for the ROS “move_base” package. The local planner takes as input a global plan produced by ROS global planner and outputs the required steering and velocity command through the “cmd_vel” topic (see Figure 3.36). The local planner consists of two components: the trajectory following part and a motor “starting” controller. The role of the “starting” controller is to overcome deadband of the ESC (see Figure 3.9). The design requirement for the “starting” controller is to respect the ActionLib API [71] used in the race manager (described in [10]) and the “move_base” components to manage the switching of goals. The ActionLib works as a Finite State Machine and reports the state of the goal (ie. Active, Pending, Succeeded, etc. – see [72]). This means, that when a goal is reached we do not know in the local planner whether a new goal will be provided by the race manager through the ActionLib or not, thus we have to stop the car and wait.

The implementation of both the trajectory following and the “starting” controller is described in the following paragraphs.

First we describe the implementation of the trajectory following part. Once a goal is sent to the “move_base” node a global plan is devised (for details about the global planner, see Section 2.3.2). This plan is passed to the local planner plugin. The global plan contains coordinates of points forming the planned trajectory as well as the car’s heading at these coordinates. Through the “costmap2DROS” [73] the local planner plugin can obtain current car’s position and heading in the map from the “move_base” package (see Figure 2.1).

In order to follow trajectory of the global plan, we use a simple PI controller. The concept will be explained with the help of Figure 3.33.

In each iteration we find the closest point on the global plan to the current position of the car and compute a difference between these two points. The difference forms a vector denoted by $\vec{r}$. In order to use the length of $\vec{r}$ as part of a control input commanding the steering of the car, we need to determine the direction of rotation. In order to do this we construct a unit vector in the planned car’s direction at the closest point on the trajectory. We denote this vector by $\vec{h_p}$. Then we compute a cross product of $\vec{r}$ and $\vec{h_p}$. The sign s of the cross product determines the direction of rotation of the car. We then multiply the length of $\vec{r}$ by the obtained sign. The procedure is clarified by the following equation

$$p = s\ \|\vec{r}\|.$$

A second component of the controller is the integrator (more precisely the summation). It sums differences between the desired heading angle (denoted by ϕ) and the current car’s heading angle (denoted by α). The ROS uses quaternions to express heading [74], so problems with discontinuity at headings $\pm\pi$ do not occur. Angle difference can thus be obtained by multiplying one heading with the inverse of the other (considering their ROS representation as quaternions). We denote the obtained difference by

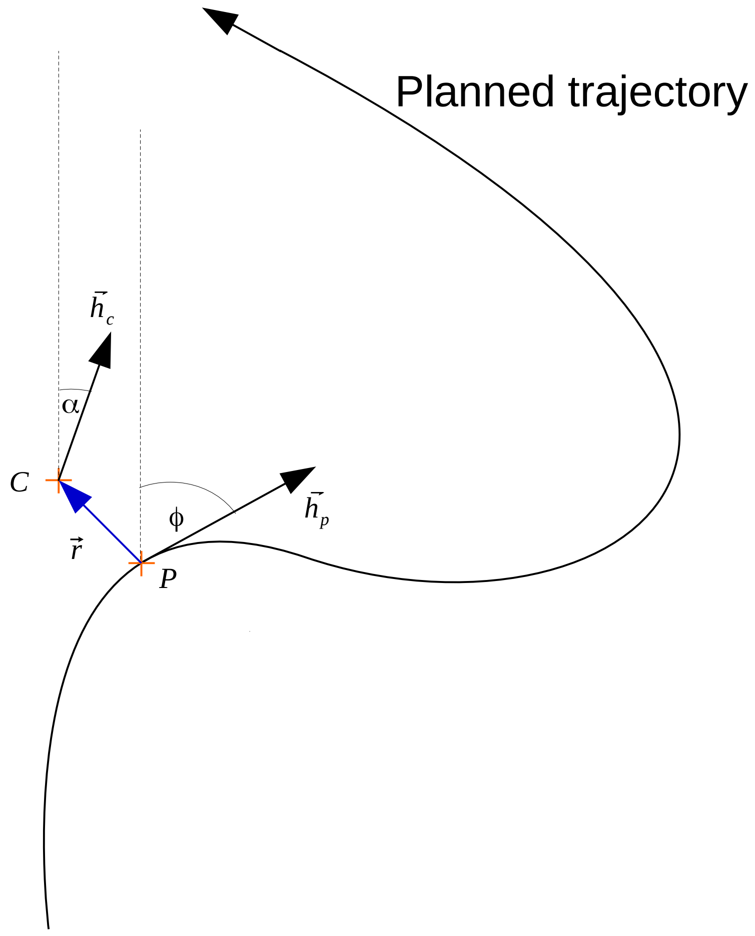

Figure 3.33. Situation picture for the local planner implementation.

Δ_k. The integrator is zeroed if the current angle difference is less than 0.2 rad. This is intended to suppress oscillations. The integrator is also zeroed if the goal is reached or if a new plan is set by the global planner. The described procedure (except for the zeroing) can be described using the following equation

$$i = \sum_{k=1}^{n} \Delta_k,$$

where n is the total number of local planner invocations.

Finally, we produce the output value of the controller as follows

$$o = K_p\, p + K_i\, i,$$

where K_p is a proportional constant and K_i is the integrator constant. The resulting value o is published as the desired "angular" velocity on the "/cmd_vel" topic. We put quotes around the angular, because we cannot directly control the angular velocity of a car using Ackermann steering geometry. The value rather represents the desired turning radius of the car, provided the wheels are in a pure rolling motion.

The "starting" controller is implemented as follows. When the first goal is set, this controller sets the desired PWM duty cycle a little above the minimum duty cycle for the car to start moving ahead and then after 1.5 s the duty cycle value is decreased and the resulting linear velocity is around 0.5 m.s^{-1}. This procedure is repeated when the defined goal is reached and a new one is accepted.

The performance of the implemented controller (in the ROS terminology local planner) was tested on the car and the behavior can be seen in Figure 3.34. The blue line represents the global plan, red arrows represent car's odometry (position and heading), white segments represent matched LiDAR scans to the underlying map. I used the Hector SLAM package to provide the odometry estimate. The developed local planner plugin can be found on the attached CD.

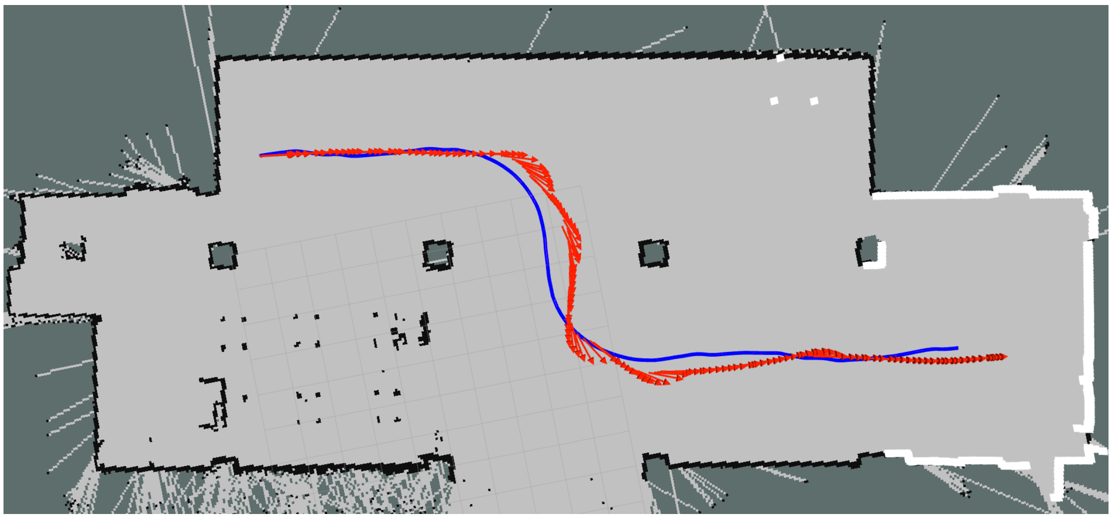

Figure 3.34. Example of trajectory following.

I chose to implement the controller as a local planner plugin to enable easy testing in the car's simulator implemented in [10]. This way I was able to run the same code both on the simulator and on the real car without any modifications, except for the controller constants. This was useful during the programming phase to debug issues.

3.16 Race starting

In this section we describe how the race starting procedure is handled on our car. Due to the fact, that ROS does not support transparent multimaster mode I had to come up with an architecture, that would allow to separate organizer's ROS master from ours, because otherwise other competitors would be able to easily intercept data sent through the ROS topics and in the worst case take over the control of our car. The architecture is depicted in Figure 3.35.

It consists of two separate processes running on the TK1. First, the race starter node, is bound to the organizer's ROS master and waits for a ReadySetGo topic reception. The other process, the race manager is bound to ROS master launched on the TK1 and waits for a message from the first process. When the race is started the race manager sets the internal nav_flag variable to true, which in turn causes periodic invocation of Spin() function of the NavigationManager, that is responsible for trajectory goals management. The goal management is described in [10].

Communication between the two mentioned processes is managed by POSIX message queue. First the race manager tries to open the queue with the name "/f1tenth_start_race" in read-only mode and creates it, if it does not exist with read and write permissions granted to the user that executed the process. Then a callback

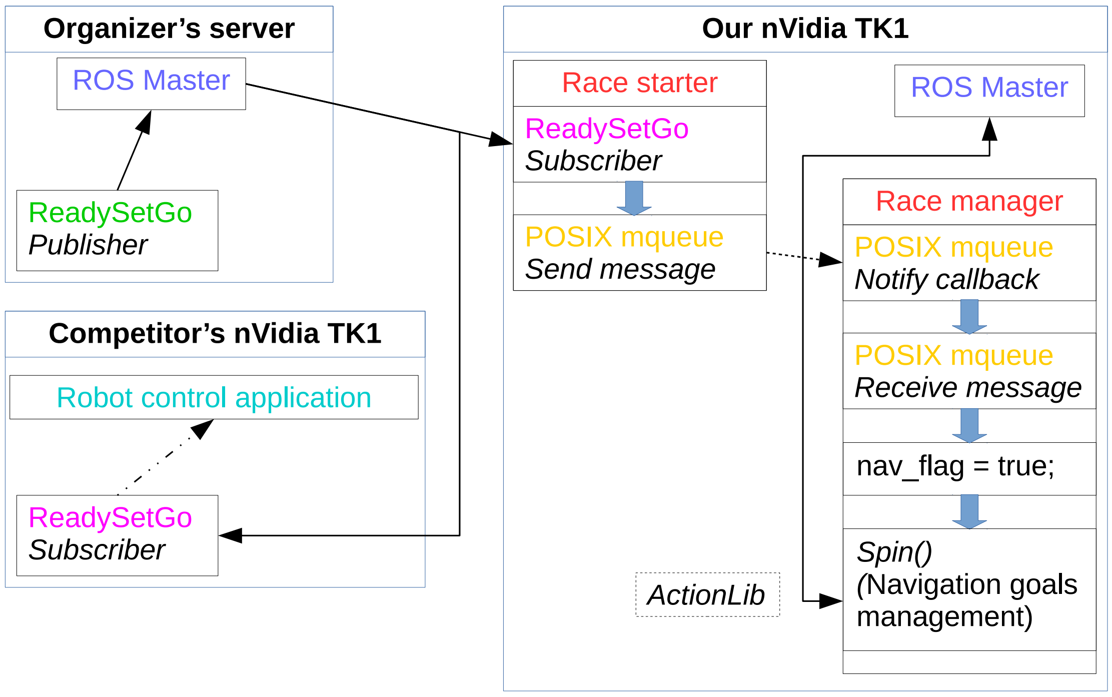

Figure 3.35. Race starter architecture

function is registered for this queue. We use the SIGEV_THREAD mode [75]. This callback function is invoked when a new message is received to the queue. The only problem with this approach is that the callback function will not be invoked if the queue was not empty at the time of callback registration. We will describe later, how this problem was solved.

The race starter node tries to open the same message queue, but, unlike the race manager, opens it in write-only mode and exits if the queue does not exist. This is to notify the user that nobody actually listens for the race starting message, for example if the race manager node is not running for some reason. Then the node waits for the reception of ReadySetGo topic with the value set to true. After the reception of this value a POSIX message is sent to the queue and the queue is unlinked and descriptor closed on the race starter node.

Queue unlinkage [76] causes the Linux kernel to remove the queue's name from the system table of known message queues, but the queue itself is only deleted when every process, that has the queue still open terminates or closes the queue.

This way we can be sure, that if the race manager dies for some reason after the queue's creation, it will not remain dangling in the kernel, causing problems with notification callback in the race manager not being invoked bacuse the queue could still contain a message sent by the race starter after the crash of the race manager.

The notifications callback registration will fail with error number EBUSY if there is another process, that has already registered for the reception of notifications for the specific message queue [75]. We use this to discover if there is another instance of race manager running and that we should leave the message queue to that process. The case with two race managers running concurrently occurs in the work of Jiří Kerner [10]. He is using two race managers to drive two simulated cars at once on the same race circuit.

The race starter is implemented in the package "f1tenth_competition_race_starter" and the race manager in the package "f1tenth_race_mgr" on the attached CD.

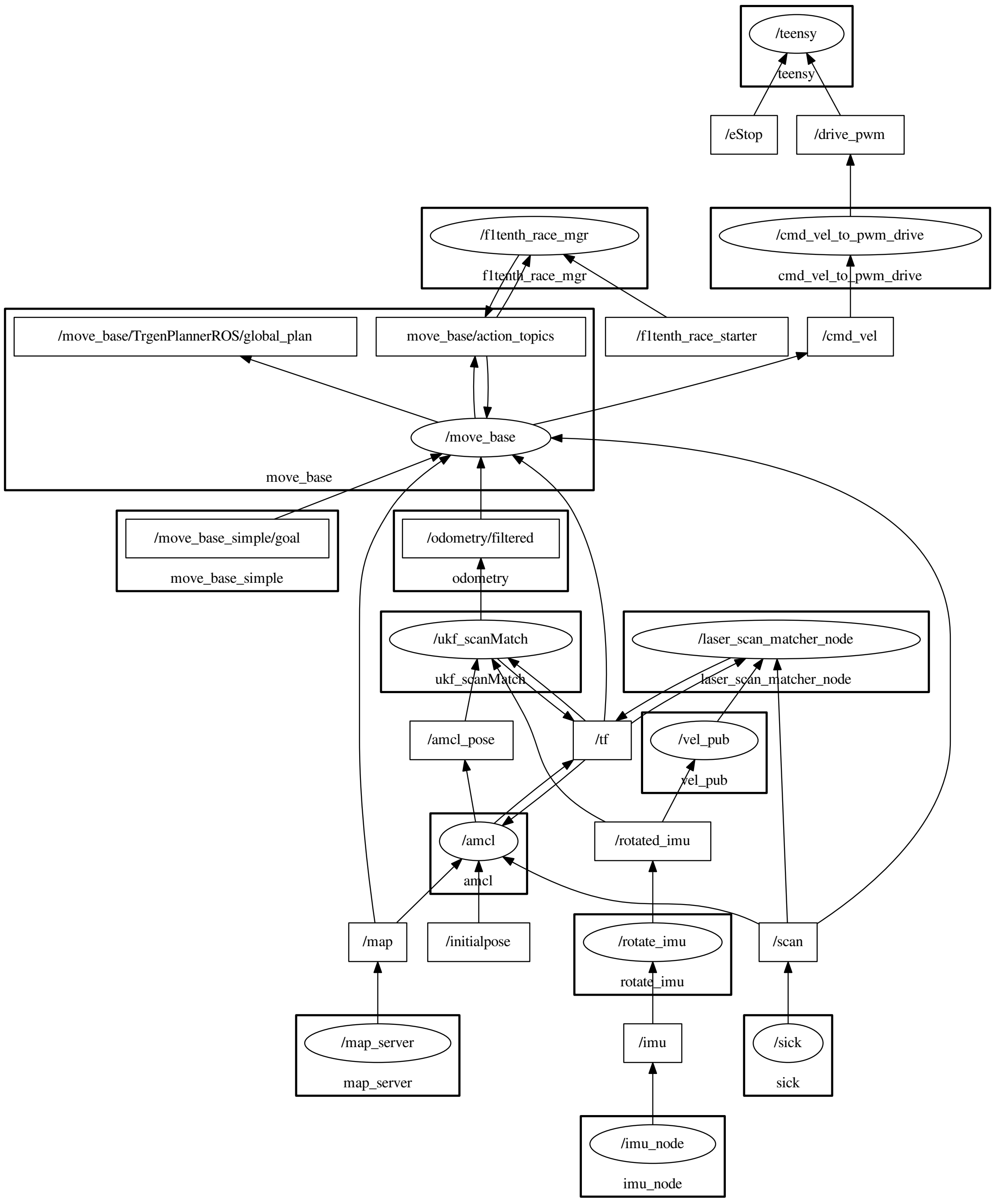

Figure 3.36. Communication between ROS nodes.

3.17 The final software architecture for competition

In this section we describe the resulting software architecture of the system to be used for the F1/10 competition. In order to get a better understanding of interconnection between various ROS nodes we provide a directed graph depicting nodes and subscribed or published topics in Figure 3.36. This figure was initially created by the "rqt_graph" tool and modified by hand for better clarity. In Figure 3.37 we can see the "Tf" subsystem transforms published by the specific nodes.

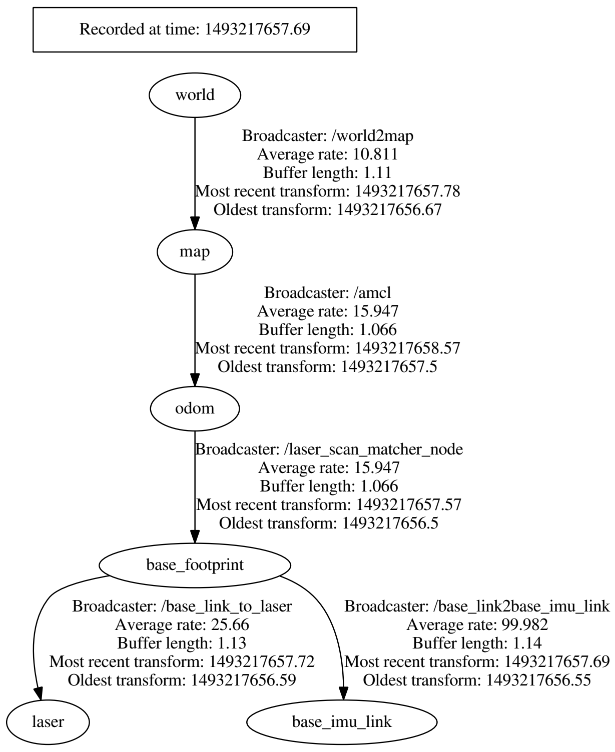
Recorded at time: 1493217657.69
world
Broadcaster: /world2map
Average rate: 10.811
Buffer length: 1.11
Most recent transform: 1493217657.78
Oldest transform: 1493217656.67
map
Broadcaster: /amcl
Average rate: 15.947
Buffer length: 1.066
Most recent transform: 1493217658.57
Oldest transform: 1493217657.5
odom
Broadcaster: /laser_scan_matcher_node
Average rate: 15.947
Buffer length: 1.066
Most recent transform: 1493217657.57
Oldest transform: 1493217656.5
base_footprint
Broadcaster: /base_link_to_laser
Average rate: 25.66
Buffer length: 1.13
Most recent transform: 1493217657.72
Oldest transform: 1493217656.59
Broadcaster: /base_link2base_imu_link
Average rate: 99.982
Buffer length: 1.14
Most recent transform: 1493217657.69
Oldest transform: 1493217656.55
laser
base_imu_link

Figure 3.37. ROS Tf frames.

On the bottom of Figure 3.36 we see the "sources" of information in the whole system. These are:

- "/imu_node" – publishes the IMU measurements. It is a part of the "razor_imu_9dof" package (see Section 2.3.2). The measurement are then transformed to account for physical placement of the unit on the car (see Section 3.7).
- "/sick" – publishes LiDAR scans. It is a part of the "sick_tim" package (see Section 2.3.2).
- "/map_server" – publishes the prepared 2D map of the environment. It is a part of the "map_server" package (see Section 2.3.2).

The map provided by the "map_server" is sent to the "move_base" (see Section 2.3.2) node and to the AMCL node (see Section 3.14 for explanation of the AMCL function and the role of "laser_scan_matcher" node). The "move_base" package uses the map to construct costmaps for the global planner in order to avoid static obstacles.

Apart from the usage of sensor data described in Section 3.14, we use the IMU measurements to improve the UKF odometry estimate (see Section 3.11).

The position and heading estimate produced by the AMCL is then processed by the UKF and sent to the "move_base" where it is used by both the global and the local planner (see Figure 2.1).

The "move_base" receives goals to plan trajetory to from the "f1tenth_race_mgr" (see Section 3.16). The goals are sent to the "move_base" through the ActionLib[1]) interface (see Figure 3.35). The "f1tenth_race_mgr" is described in [10].

The plan produced by the global planner of the "move_base" node is followed by the local planner (see Section 3.15). The local planner produces the "/cmd_vel" topic which contains the requested movement commands and these are passed to

[1]) `http://wiki.ros.org/actionlib`

the “cmd_vel_to_pwm_drive” node which acts as a feedforward controller and uses the results of identification obtained in Section 3.10. The node sends scaled PWM duty cycle (for the explanation of scaling, see Section 3.6) values to the “/teensy” node representing the “rosserial” interface to the Teensy MCU. The Teensy MCU then generates the required PWM signals and sends them to the ESC and servo (see Figure 3.1).

It is also possible to force the car to stop moving by sending the value “true” on the “eStop” topic. This is used to react to dynamic obstacles and also static obstacles during the race. If an obstacle is too close (0.5 m) in the LiDAR range, we stop the car through the “eStop” topic. The implementation is in file “emergency_stop.py” in package “competition” on the attached CD.

If the car starts behaving strangely it’s control can be manually taken over by using the RF controller (for details about implementation, see Section 3.6).

The described stack can be launched using the “competition_known_map.launch” launch file, that is part of the “competition” package. It can be found on the attached CD.

Details about how to use the outcomes of this thesis is briefly described in Appendix C. We provide it for newcomers to our F1/10 team. For more detailed description, see [10].

Chapter 4
Evaluation of the resulting architecture

In this chapter we describe the experiments used to evaluate the control architecture described in Section 3.17.

4.1 Teleoperation of the vehicle

In order to test the teleoperation of the vehicle, that is required by competition rules (see Section 2.1, rule 4.4), we use the ROS Control Android application mentioned in Section 2.3.1. The application can use the accelerometer of the Android device to emulate a joystick and transform the measured acceleration vector into a “cmd_vel” topic to control the car.

4.2 Switching to manual control mode

The implementation of automatic switching to manual mode (for implementation see Section 3.6) as required by the F1/10 rules (see Section 2.1, rule 7.1) was tested on numerous occasions and it prevented the occurence of many accidents during the testing phase.

4.3 Wall following

A wall following controller, that uses the LiDAR scans was developed during the powertrain identification experiments (see Section 3.10.2) a modified version was used during the Faculty’s Door Open Days at the Karlovo Náměstí campus. The implementation is in the file “scan_regression.py” in the ROS package “competition” on the attached CD. The CD also contains a video from the event – “DoD.mp4”.

4.4 Trajectory following

The implemented trajectory following controller (for details, see Section 3.15) was tested in our laboratory. Video from the testing is attached on the CD – “trajectory_following.mp4”.

4.5 Traversal of ordered series of checkpoints

The traversal of ordered series of checkpoints, that is required by the rules (see Section 2.1, rules 4.4) was tested in our laboratory. The test was a success and the resulting video documenting the experiment is on the attached CD – “checkpoint_traversal.mp4”.

Chapter 5
Future work

In this chapter we explore some of the possibilities for improvement of the developed platform. These are of two kinds. The first one focuses on the hardware of the platform and the second one focuses on the software and algorithms.

5.1 Hardware improvements

As stated in Section 2.4. The car is not fit for competition, because it uses a different LiDAR and vehicle, than what was prescribed by the rules of the F1/10 competition. Since the April 2017 round of the competition was canceled, the purchase of the components was postponed, but for the October 2017 round it needs to be sorted out. In the following list we present other ideas for improvement:

- Use the VESC-X [77] Electronic Speed Controller instead of the one supplied with the vehicle. It provides current and voltage measurement on all motor phases and thus is capable of providing a wheel odometry. One limitation of the wheel odometry is, that it is not accurate when a sliding motion of the car's wheels occur, but even under such circumstances it would be possible to control the motor speed. The odometry estimate could also be used to improve performance of the "laser_scan_matcher" package.

  The competing venture MIT-RACECAR even (see Section 2.5) provides ROS packages[1]) to interface with the VESC-X and provide the odometry estimates. This modification is, however, subject to the approval of F1/10 organizers, but we mention it in case we switch to the MIT-RACECAR competition in the future.
- Put rubber washers between the IMU board and the standoffs to decrease the effect of motor vibrations on the IMU. This would improve the estimates of the UKF as the measurement noise would likely decrease.
- Finish the stop button. The current solution relies only on the power switch of the ESC, which is not very easily accessible. In order for the button to fit in the intended position on the upper chassis plate it would be necessary to use longer plastic standoffs than are currently used.

5.2 Software improvements

In this section we take a closer look at possible improvements of the software stack.

- Prepare a more rigorous model of the car than what was done in Section 3.10. The obtained model of car's dynamics could be easily incroporated into the source code of the UKF node (see Section 3.9) and the accuracy of the provided estimates could be enhanced especially when rapid changes of angular or linear velocity should occur.

[1]) `https://github.com/mit-racecar/vesc`

- Another possible improvement is connected with the previous one and consists of determining the centre of mass of the car and using this as the "base_link" reference frame in ROS. This frame is used for transformation of received data into a common frame. At present the frame is identical to the frame of the LiDAR.
- Use the measured centripetal acceleration to help determine the turning radius more accurately. At present we divide the linear velocity estimate of the UKF by the measured angular velocity, but this is prone to possible errors in the estimate. A more accurate solution would be to divide the measured centripetal acceleration by square of the measured angular velocity.
- Devise a reliable algorithm for indoor stereo odometry. According to my experiments (see Section 3.13), it is a difficult algorithmic task, because the environments we are dealing with lack visual markers and the fact, that the viewing angle of the ZED camera is only 110 degrees does not help either. Another difficulty is the fact, that the camera can only provide a somewhat limited framerate compared to using the ZED on better supported platforms than the NVIDIA TK1 (see Figure 2.1). Some of the best currently available visual odometry solutions use the measurement of angular velocity to better deal with fast rotation movement. However these implementations are not publicly available. One such example would be [78]. The problem with rotation is, that the algorithms usually are not able to find new markers in the turning scene and maintain sufficient number of already identified markers and they get lost quickly.
- Another possibility to improve the odometry would be to use the VESC-X and a particle filter localization implemented by the MIT RACECAR authors[1]). It uses CUDA to accelerate the required computations and solves the problem experienced with the AMCL. The AMCL runs only on a single CPU core and this proved to be a bottleneck (see Section 3.14).

[1]) `https://github.com/mit-racecar/particle_filter`

Chapter 6
Conclusion

The aim of this thesis was to develop a race car to take part in the F1/10 scale model car competition. The car was successfully constructed and is capable of autonomous drive. Most of the functionality needed for F1/10 participation is implemented.

Our implementation utilizes many components available in ROS, that were selected based on the evaluation in this thesis. The trajectory follower and the handling of the race starting procedure was implemented in this thesis. What is missing is the interpretation of dynamic obstacles and subsequent predictive control, also the odometry needs to be improved by utilizing the stereo camera.

We described the hardware used on the platform. This included various sensors and processing units. What system we use on the platform (ROS Indigo and Ubuntu 14.04) and the device drivers used for interfacing the devices with the ROS topic subsystem.

We designed the chassis and the car completion procedure. Then we moved on to the hardware-bringup phase and basic testing of sensors (publication of the sensor data over the ROS topics).

We changed firmwares of the Teensy board and of the IMU.

Later we explored the possibilities for localization in the environment using existing ROS packages. This consisted of testing the various SLAM and visual odometry packages in an environment similar to where the competition is to be held and of description of the outcomes.

A trajectory following controller was implemented and its performance demonstrated. The trajectory following controller was tested inside the Gazebo simulator. The world and car model for the simulator were prepared as a part of Master's Thesis of another team member [10].

Also a simple kinematics model of the car was found and used to transform commanded velocity received from the trajectory following controller to the appropriate PWM duty cycle.

A solution to the race starting problem and the lack of ROS support for multi-master mode was also presented.

The final control architecture was presented and its performance assesed.

Finally we proposed some possibilities for future improvement with the respect to both the hardware and the software of the platform.

References

[1] *Introducing the new NVIDIA Drive PX 2*. Online.
http://www.nvidia.com/object/drive-px.html. Visited 2017-05-24.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. *CoRR*. 2016, abs/1604.07316

[3] Madhur Behl. *F1/10 CPS Week 2017 Competition Cancelled*. Online.
http://f1tenth.org/forum?place=msg%2Ff1_10%2FKAJRPAqWJkU%2Fubs7iRg6BwAJ. Visited 2017-05-24.

[4] F1/10 Team. *Rules*. Online.
http://f1tenth.org/rules. Visited 2017-05-24.

[5] F1/10 Team. *Car Assembly*. Online.
http://f1tenth.org/car-assembly. Visited 2017-05-24.

[6] Madhur Behl. *Does anyone know what the laps will be look like?* Online.
http://f1tenth.org/forum?place=msg%2Ff1_10%2F716PEL-HVLo%2F1c0gee2yCQAJ. Visited 2017-05-24.

[7] F1/10 Team. *Code and documentation provided by the F1/10 organizers*. Online.
https://github.com/mlab-upenn/f1tenthpublic. Visited 2017-05-24.

[8] Open Source Robotics Foundation. *About ROS*. Online.
http://www.ros.org/about-ros. Visited 2017-05-24.

[9] Open Source Robotics Foundation. *Integration with Other Libraries*. Online.
http://www.ros.org/integration/. Visited 2017-05-24.

[10] Jiří Kerner. *Software testing for embedded applications in autonomous vehicles*. Master's Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering. 2017.

[11] Open Source Robotics Foundation. *Core Components*. Online.
http://www.ros.org/core-components. Visited 2017-05-24.

[12] *ROS community, ROS Wiki: Nodes*. Online.
http://wiki.ros.org/Nodes?distro=indigo. Visited 2017-05-24.

[13] *ROS community, ROS Wiki: roscore*. Online.
http://wiki.ros.org/roscore?distro=indigo. Visited 2017-05-24.

[14] *ROS community, ROS Wiki: Master*. Online.
http://wiki.ros.org/Master?distro=indigo. Visited 2017-05-24.

[15] *ROS community, ROS Wiki: Topics*. Online.
http://wiki.ros.org/Topics?distro=indigo. Visited 2017-05-24.

[16] *ROS community, ROS Wiki: Parameter Server*. Online.
http://wiki.ros.org/Parameter Server?distro=indigo. Visited 2017-05-24.

[17] *ROS community, ROS Wiki: rqt_console*. Online.
http://wiki.ros.org/rqt_console?distro=indigo. Visited 2017-05-24.

[18] *ROS community, ROS Wiki: roslaunch*. Online.
http://wiki.ros.org/roslaunch?distro=indigo. Visited 2017-05-24.

[19] *ROS community, ROS Wiki: catkin*. Online.
http://wiki.ros.org/catkin?distro=indigo. Visited 2017-05-24.

[20] *ROS community, ROS Wiki: dynamic_reconfigure*. Online.
http://wiki.ros.org/dynamic_reconfigure?distro=indigo. Visited 2017-05-24.

[21] *ROS community, ROS Wiki: roscpp*. Online.
http://wiki.ros.org/roscpp?distro=indigo. Visited 2017-05-24.

[22] *ROS community, ROS Wiki: rospy*. Online.
http://wiki.ros.org/rospy?distro=indigo. Visited 2017-05-24.

[23] *ROS community, ROS Wiki: rosjava*. Online.
http://wiki.ros.org/rosjava?distro=indigo. Visited 2017-05-24.

[24] *sensor_msgs/LaserScan Message*. Online.
http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html. Visited 2017-05-24.

[25] *ROS community, ROS Wiki: hokuyo_node*. Online.
http://wiki.ros.org/hokuyo_node?distro=indigo. Visited 2017-05-24.

[26] *ROS community, ROS Wiki: sick_tim*. Online.
http://wiki.ros.org/sick_tim?distro=indigo. Visited 2017-05-24.

[27] *ROS community, ROS Wiki: rosserial*. Online.
http://wiki.ros.org/rosserial?distro=indigo. Visited 2017-05-24.

[28] *sensor_msgs/Imu Message*. Online.
http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html. Visited 2017-05-24.

[29] *ROS community, ROS Wiki: razor_imu_9dof*. Online.
http://wiki.ros.org/razor_imu_9dof?distro=indigo. Visited 2017-05-24.

[30] StereoLabs Inc. *zed-ros-wrapper*. Online.
https://github.com/stereolabs/zed-ros-wrapper/releases/tag/v1.2.0. Visited 2017-05-24.

[31] StereoLabs Inc. *ZED SDK 1.2*. Online. 2016.
https://www.stereolabs.com/developers/release/1.2. Visited 2017-05-24.

[32] *nav_msgs/Odometry Message*. Online.
http://docs.ros.org/api/nav_msgs/html/msg/Odometry.html. Visited 2017-05-24.

[33] *sensor_msgs/Image Message*. Online.
http://docs.ros.org/api/sensor_msgs/html/msg/Image.html. Visited 2017-05-24.

[34] *sensor_msgs/CameraInfo Message*. Online.
http://docs.ros.org/api/sensor_msgs/html/msg/CameraInfo.html. Visited 2017-05-24.

[35] *ROS community, ROS Wiki: navigation*. Online.
http://wiki.ros.org/navigation?distro=indigo. Visited 2017-05-24.

[36] *ROS community, ROS Wiki: move_base*. Online.
http://wiki.ros.org/move_base?distro=indigo. Visited 2017-05-24.

[37] *ROS community, ROS Wiki: global_planner*. Online.
http://wiki.ros.org/global_planner?distro=indigo. Visited 2017-05-24.

[38] *State Estimation Nodes – ukf_localization_node*. Online.
http://docs.ros.org/kinetic/api/robot_localization/html/state_estimation_nodes.html#ukf-localization-node. Visited 2017-05-24.

[39] *ROS community, ROS Wiki: rViz*. Online.
http://wiki.ros.org/rviz?distro=indigo. Visited 2017-05-24.

[40] *ROS community, ROS Wiki: rqt_graph*. Online.
http://wiki.ros.org/rqt_graph?distro=indigo. Visited 2017-05-24.

[41] *ROS community, ROS Wiki: rqt_tf_tree*. Online.
http://wiki.ros.org/rqt_tf_tree?distro=indigo. Visited 2017-05-24.

[42] *ROS community, ROS Wiki: map_server*. Online.
http://wiki.ros.org/map_server?distro=indigo. Visited 2017-05-24.

[43] StereoLabs Inc. *ZED – Depth Sensing and Camera Tracking*. Online.
https://www.stereolabs.com/zed/specs. Visited 2017-05-24.

[44] *ROS community, ROS Wiki: rosbag*. Online.
http://wiki.ros.org/rosbag?distro=indigo. Visited 2017-05-24.

[45] *The Grinch 21.3.4 for Jetson TK1 / developed*. Online.
https://devtalk.nvidia.com/default/topic/906018/jetson-tk1/-customkernel-the-grinch-21-3-4-for-jetson-tk1-developed/post/4765630/#4765630. Visited 2017-05-24.

[46] *Teensy 3.2 & 3.1: New Features*. Online.
https://www.pjrc.com/teensy/teensy31.html#specs. Visited 2017-05-24.

[47] *Teensy and Teensy++ Schematic Diagrams*. Online.
https://www.pjrc.com/teensy/schematic.html. Visited 2017-05-24.

[48] *Basic Teensyduino Usage*. Online.
https://www.pjrc.com/teensy/td_usage.html. Visited 2017-05-24.

[49] *AmazonBasics 7 Port USB 3.0 Hub with 12V/3A Power Adapter*. Online.
https://www.amazon.com/AmazonBasics-Port-USB-Power-Adapter/dp/B00E6GX4BG. online store listing, visited 2017-05-24.

[50] *Instruction Manual, Fuze 130A Sensorless Brushless ESC*. Online.
http://www.dynamiterc.com/ProdInfo/Files/DYN4955_Manual.pdf. Visited 2017-05-24.

[51] Tully Foote, and Mike Purvis. *Standard Units of Measure and Coordinate Conventions*. Online. 2014.
http://www.ros.org/reps/rep-0103.html. Visited 2017-05-24.

[52] Wim Meeussen. *Coordinate Frames for Mobile Platforms*. Online. 2010.
http://www.ros.org/reps/rep-0105.html. Visited 2017-05-24.

[53] *Gutta Hobbyglas*. Online.
https://www.guttashop.cz/data/upload/ke-stazeni/plexisklo_hobbyglas_cz.pdf. Visited 2017-05-24.

[54] *Getting Started with Jetson TK1 and the ZED*. Online.
https://www.stereolabs.com/blog/index.php/2015/09/24/getting-started-with-jetson-tk1-and-zed. Visited 2017-05-24.

[55] *How to increase frame rate* . Online.
https://github.com/stereolabs/zed-ros-wrapper/issues/17#issuecomment-182875722. Visited 2017-05-24.

[56] *Coreutils - GNU core utilities*. Online.
https://www.gnu.org/software/coreutils/coreutils.html. Visited 2017-05-24.

[57] Inc. Freescale Semiconductor. *K20 Sub-Family Reference Manual*. Online.
https://www.pjrc.com/teensy/K20P64M72SF1RM.pdf. Visited 2017-05-24.

[58] The F1/10 Team. *Tutorial 5: Sensor Configuration and Rosbags*. Online.
https://github.com/mlab-upenn/f1tenthpublic/blob/master/docs/t5.pdf. Visited 2017-05-24.

[59] InvenSense Inc. *ITG-3200 Product Specification Revision 1.4*. Online. 2010.
https://www.sparkfun.com/datasheets/Sensors/Gyro/PS-ITG-3200-00-01.4.pdf. Visited 2017-05-24.

[60] E. Tsardoulias, and L. Petrou. Critical Rays Scan Match SLAM. *Journal of Intelligent & Robotic Systems*. 2013, 2 (3), 441–462. DOI 10.1007/s10846-012-9811-5.

[61] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf. *A flexible and scalable SLAM system with full 3D motion estimation.* In: *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*. 2011. 155-160.

[62] G. Grisetti, C. Stachniss, and W. Burgard. Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics*. 2007, 23 (1), 34-46. DOI 10.1109/TRO.2006.889486.

[63] *LIBVISO2: C++ Library for Visual Odometry 2*. Online.
http://www.cvlibs.net/software/libviso. Visited 2017-05-24.

[64] A. Geiger, J. Ziegler, and C. Stiller. *StereoScan: Dense 3d reconstruction in real-time.* In: *2011 IEEE Intelligent Vehicles Symposium (IV)*. 2011. 963-968.

[65] *ROS community, ROS Wiki: viso2_ros*. Online.
http://wiki.ros.org/viso2_ros?distro=indigo. Visited 2017-05-24.

[66] Raúl Mur-Artal, Juan D. Tardós, José M. M. Montiel, and Dorian Gálvez-López. *ORB-SLAM2 repository*. Online.
https://github.com/raulmur/ORB_SLAM2. Visited 2017-05-24.

[67] Raúl Mur-Artal, and Juan D. Tardós. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *arXiv preprint arXiv:1610.06475*. 2016,

[68] *ROS community, ROS Wiki: amcl*. Online.
http://wiki.ros.org/amcl?distro=indigo. Visited 2017-05-24.

[69] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots.* In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence.* Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999. 343–349. ISBN 0-262-51106-1.
http://dl.acm.org/citation.cfm?id=315149.315322.

[70] A. Censi. *An ICP variant using a point-to-line metric.* In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Pasadena, CA: 2008.
http://purl.org/censi/2007/plicp.

[71] *ROS community, ROS Wiki: actionlib*. Online.
http://wiki.ros.org/actionlib?distro=indigo. Visited 2017-05-24.

[72] *actionlib_msgs/GoalStatus Message*. Online.
`http://docs.ros.org/api/actionlib_msgs/html/msg/GoalStatus.html`. Visited 2017-05-24.

[73] *costmap_2d::Costmap2DROS Class Reference*. Online.
`http://docs.ros.org/indigo/api/costmap_2d/html/classcostmap__2d_1_1Costmap2DROS.html`. Visited 2017-05-24.

[74] *geometry_msgs/Pose Message*. Online.
`http://docs.ros.org/api/geometry_msgs/html/msg/Pose.html`. Visited 2017-05-24.

[75] *mq_notify – Linux Programmer's Manual*. Online.
`http://man7.org/linux/man-pages/man3/mq_notify.3.html`. Visited 2017-05-24.

[76] *mq_unlink – Linux Programmer's Manual*. Online.
`http://man7.org/linux/man-pages/man3/mq_unlink.3.html`. Visited 2017-05-24.

[77] *NEW: Vesc-X Motor Controller*. Online.
`http://www.enertionboards.com/electric-skateboard-parts/vesc-x-programmable-brushless-motor-controller`. Visited 2017-05-24.

[78] I. Cvišić, and I. Petrović. *Stereo odometry based on careful feature selection and tracking.* In: *2015 European Conference on Mobile Robots (ECMR)*. 2015. 1-6.

Appendix A
Glossary

ABS	Acrylonitrile butadiene styrene
AMCL	Adaptive Monte-Carlo Localization
API	Application Programming Interface
CNC	Computer numerical control
CPU	Central Processing Unit
DNN	Deep Neural Network
eMMC	embedded Multimedia Card
ESC	Electronic Speed Controller
FTM	FlexTimer module
GPU	Graphics Processing Unit
HDMI	High-Definition Multimedia Interface
ICP	Iterative Closest Point
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
IP	Internet Protocol
ISR	Interrupt Service Routine
JPEG	Joint Photographic Experts Group
LED	Light-Emitting Diode
LiDAR	Light Detection and Ranging
MCU	Microcontroller Unit
NVIC	Nested Vectored Interrupt Controller
PCIe	Peripheral Component Interconnect Express
PNG	Portable Network Graphics
POSIX	Portable Operating System Interface
PWM	Pulse-Width Modulation
RADAR	Radio Detection and Ranging
RF	Radio Frequency
RNDF	Route Network Definition File
ROS	Robot Operating System
RPC	Remote procedure call
SATA	Serial AT Attachment
SD	Secure Digital
SDK	Software Development Kit
SLAM	Simultaneous Localization and Mapping
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UKF	Unscented Kalman Filter
USB	Universal Serial Bus
XML	Extensible Markup Language
4WD	Four-wheel drive

Appendix B
Contents of the attached CD

The CD contains a complete repository of the preparation project with work of all the teammates. Here we only list parts implemented or modified in this thesis.

```
/
├── Chassis_drawings ........................... AutoCad drawings of the chassis
├── F1_10_Project ................................ Main repository of our project
│   ├── HW
│   │   ├── imu_calibration ........................ Calibration values for the IMU
│   │   ├── Razor_AHRS ........................................... IMU firmware
│   │   ├── teensy_firmware.ino ............................ Teensy MCU firmware
│   │   └── ZED.yaml ........................... Configuration for the ORB SLAM2
│   └── catkin_ws ........................................ ROS Catkin workspace
│       └── src
│           └── ctu_packages
│               ├── competition
│               │   ├── launch
│               │   │   ├── competition_known_map.launch ............... Race mode
│               │   │   ├── create_waypoints.launch .......... Creation of waypoints
│               │   │   ├── demonstrations_and_mapping.launch .... Hector 2D SLAM
│               │   │   ├── pc_visualize.launch ..... Odometry and map visualization
│               │   │   ├── publish_all.launch ................ Publishes sensor data
│               │   │   └── record.launch ........... Records sensor data to a rosbag
│               │   └── src
│               │       ├── emergency_stop.py ....... Emergency response to obstacles
│               │       └── scan_regression.py ....... Doors Open Day demonstration
│               ├── f1tenth_packages
│               │   └── f1tenth_race
│               │       ├── f1tenth_competition_race_starter .......... Race starter
│               │       └── f1tenth_race_mgr .......................... Race manager
│               ├── my_race
│               │   ├── launch
│               │   │   ├── crsm_slam_real.launch .......... CRSM SLAM testing file
│               │   │   └── demo_gmapping.launch .............. Gmapping testing file
│               │   └── src
│               │       └── experiment1.py ....... Powertrain identification experiment
│               └── trgen_local_planner ............. Trajectory following controller
├── Thesis.pdf ................................ Electronic version of this thesis
└── Videos
    ├── DoD.mp4 ........................................... Doors Open Day event
    ├── checkpoint_traversal.mp4 .............. Traversal of ordered checkpoints
    └── trajectory_following.mp4 ......................... Trajectory following
```

Appendix C
User guide

In order to use the outcomes of this thesis, it is necessary to perform the following steps.

- install all the required ROS and/or Gazebo packages (details can be found in [10])
- clone git repository of our project

```
git clone ssh://git@rtime.felk.cvut.cz/f1tenth
```

- compile the catkin workspace (details can be found in [10])
- source the “devel/setup.bash” file

In order to create a map using Hector 2D SLAM, execute on the TK1:

```
roslaunch competition demonstrations_and_mapping.launch
```

In order to create waypoints for the race, execute on PC:

```
roslaunch competition create_waypoints.launch
```

In order to launch the race infrastructure and use AMCL with the previously constructed map, execute on TK1:

```
roslaunch competition competition_known_map.launch
```

In order to record data captured by the LiDAR, ZED camera, IMU or the Teensy board, execute on TK1:

```
roslaunch competition record.launch
```