Using SLAM in Formula Student

Stefan Bjørn Gunnarsson
Janus Nørgaard Syrak Hansen
Supervisor: Dirk Kraft

March 7, 2022

Abstract

Formula Student is an engineering competition in which more than 100 engineering teams from all over the world compete in various racing events. One of the categories is the Driverless Vehicle Category. In one of the Driverless Vehicle events, an autonomous vehicle must navigate a previously unknown course for time.

To navigate in such an event, the vehicle needs to be able to create a map of the course, which is demarcated by cones. The vehicle additionally needs to localize itself within the environment to ensure that it stays within the map and that the created map does not drift. This is also known as the Simultaneous Localization and Mapping (SLAM) problem.

In order to perform this mapping, the vehicle needs to be equipped with one or several exteroceptive sensors and integrate the information received from these in the decision-making system. There are three important pieces of information necessary about the track; the location, color, and size of the cones.

In this thesis, an autonomous system for solving the SLAM problem in a Formula Student Environment is developed. The system uses LiDAR and camera data to determine the information about the cones.

The system uses the FastSLAM 2.0 algorithm, using the estimated centers of the cones as input.

The results show that the system is scalable by altering the number of particles. Furthermore, the presented approach, which relies on semantic segmentation on image data to filter a point cloud, is deemed valid.

The system is able to correctly build a map of the environment; an RMSE of 0.0913 is achieved using 200 FastSLAM 2.0 particles on a custom course.

The code developed in this project can be found in `https://github.com/stebbibg/MSc_Fstudent_SLAM`.

Keywords: SLAM, Formula Student, Driverless Vehicle, Semantic Segmentation, FastSLAM, Sensor Fusion

Preface

The authors collaborated for the entire project. They are both accountable for the entirety of the thesis.

Table of Contents

Acknowledgements

First, we would like to thank Anders Glent Buch for helping us form the original idea of the approach used in this project and providing invaluable insight into the theory and approaches of neural networks in general.

A big thank you to Gunnar Stefansson for helping us interpret the statistical descriptions of the various algorithms used in this project.

We would also like to thank Andreas Calov for providing excellent technical support in using the vehicle.

Next, we would also like to thank Karsten Holm Andersen for providing us access to both the vehicle and the LiDAR used in this project and helping us come up with the initial idea of the project.

We extend our thanks to Jakob Wilm for lending us the camera used in this project.

Last but not least, we would like to thank our supervisor, Dirk Kraft, for bearing with our numerous questions about the project and the report. This project would not have been the same without him.

1 Introduction

Formula Student presents a simplified environment for testing and validating different methods in autonomous vehicles, specifically in their Driverless category. Formula Student is a competition where engineering students from more than 100 teams worldwide compete in racing events with a self-developed race car [34]. It is the most well-established educational engineering competition in Europe [34], with the first competition being held in 1998 [16].

The goal of the competition is to inspire and motivate young engineering students, as well as inspire more young people to take up a career in engineering [34].

Each of these competitions consists of static and dynamic events. The dynamic events are where the actual racing takes place. The most challenging event in the Driverless Category is the Trackdrive, where the autonomous vehicle needs to complete a timed race on an unknown racetrack [17]. The limitations of the trackdrive event can be seen in fig. 1 [17].

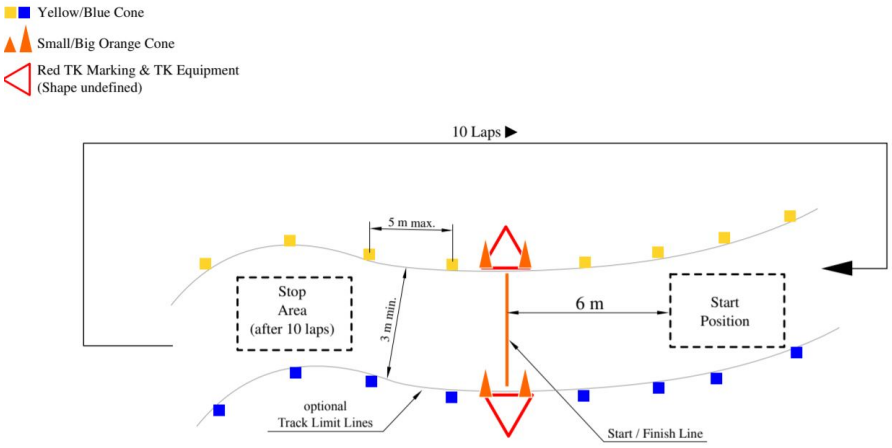


Figure 1: The constraints of the Driverless Trackdrive event. Figure taken from [17].

To navigate such a course, the autonomous system needs to accomplish several tasks:

1. Construct a map of the track while driving
2. Localize itself within the map
3. Determine the speed and direction to drive
4. Determine which control outputs leads to the desired direction and speed

1.1 Problem Statement

This project aims to develop a system that can construct a map of an unknown Formula Student track while driving. The car needs to be able to localize itself during driving within the constructed map. This is also referred to as the Simultaneous Localization and Mapping (SLAM) problem. The system should be able to function in an autonomous vehicle setup for Formula Student. The primary goal is that the developed system can perform with the accuracy needed for completing a Driverless competition. Furthermore, several constraints to the system are defined:

- The system should be easy to understand and modify
- The system should be scalable for the trade-off between runtime and accuracy
- The system should function with a variety of interoceptive sensors

Developing a fully autonomous Formula Student Driverless system is a complicated task. This can be concluded simply by looking at the scores of previous competitions, where only the best and the second-best teams have consistently performed well in the competition, making at least 18 teams unable to perform the task to a satisfactory degree [18]. Therefore the focus is making an easy-to-understand and straightforward system that can serve as a starting point for new or struggling teams, SDU-Vikings [2] being one of them.

The system needs to be scalable to fit different implementations and microcontroller setups.

Lastly, it needs to function with a variety of interoceptive sensors. The system should not inhibit the engineers in the competition, and therefore no choices about internal sensor setup are imposed.

Due to the size of the problem of constructing a fully functioning autonomous vehicle, only the parts of the problem required to solve the SLAM problem are considered. Since the map is not known in advance, the vehicle needs to perform path planning while driving. Therefore, only SLAM approaches that can work in real-time are considered.

The approach suggested in this thesis has not been performed before, to our knowledge, in a Formula Student environment. The project developed for this thesis is intended to be a proof-of-concept of the method. As such, the system's runtime is taken into consideration, but real-time operation is not the goal of this project.

One of the underlying assumptions of this project is that the strategy in an actual competition would be to drive one lap slowly to estimate the map with relatively high accuracy. After the initial mapping round, the vehicle can use faster localization algorithms within the constructed map. The project's focus is the first lap; therefore, relatively low velocities of the vehicle are assumed.

1.2 Thesis Outline

Section 2 contains the review of the literature that was investigated when choosing methods for this thesis. It is divided into three sub-sections, where the first two (section 2.1 and section 2.2) are components to the overall system. The final section, section 2.3, describes a complete state-of-the-art driverless system used in Formula Student.

Section 3 includes a summary and description of important notations used in the SLAM problem. This is a prerequisite for understanding the background material in section 4.

Section 4 introduces information and description of the methods used in this project found in the literature. This section covers background information on SLAM, Semantic Segmentation, and the Formula Student competition.

Section 5 describes the choice of the system, as well as considerations taken into account.

Section 6 describes the methods that were used in the detection and classification of cones.

In section 7, the methods developed and used for the SLAM part of the project are described.

Section 8 describes the preliminary experiments that were performed to develop a fully functioning system.

Section 9 details the experiments that were performed to assess the validity of the approach.

Finally, in section 10, the final conclusions of the project are presented, as well as a discussion on the different results obtained from the experiments.

2 Literature review

In this chapter, the literature for the areas that are considered in this project is reviewed. In section 2.1, a review of some original papers related to the solution of the SLAM problem, as well as recent solutions, are presented. In section 2.2, various semantic segmentation approaches from the literature are reviewed. Finally, section 2.3 presents an official paper that describes a solution to the SLAM problem in a Formula Student context by the most successful Driverless Vehicle team.

2.1 SLAM

SLAM is the problem of estimating a robot's pose while simultaneously building a map of the environment. It is generally considered to be one of the primary keys to developing genuinely autonomous robots and has numerous applications [11, 4]. It has been an important research area within robotics for more than 25 years [4].

The concept of SLAM is a broad field and includes several topics such as sensor fusion and state estimation [4].

Only approaches involving at least a LiDAR are considered in this thesis since that is a requirement of the proposed approach.

The SLAM problem can generally be divided into two different approaches; full SLAM and online SLAM. In full SLAM, the goal is to estimate the complete path of the robot from the time $t = 0$ to the time $t = T$, where T is the last time step [4]. The full SLAM problem will not be considered in this thesis, as it does not apply to applications involving real-time operation. In online SLAM, the primary goal is to estimate the current position of the vehicle and the current observable map, typically based on the most recent sensor information [4]. The online SLAM problem is typically solved using filter-based methods [4].

Eliazar and Parr [12] presented an algorithm based on pure LiDAR input in 2003, named DP-SLAM. Their algorithm uses a particle filter to represent both robot poses and possible map configurations. They were able to maintain and update hundreds or thousands of candidate maps, and robot poses in real-time by using a new map representation, which they referred to as distributed particle mapping. Their algorithm has a worst-case complexity per laser sweep that is log-quadratic in the number of particles and linear in the area covered by the LiDAR. They were able to avoid the data association problem by storing multiple detailed maps instead of sparse landmarks. This means that they combine the association with the localization. DP-mapping maintains a single occupancy grid, where the particles are distributed over the map. Each grid square stores a balanced tree. They were able to construct a map of a simple office environment rather well. In 2004, they presented a new and improved version of DP-SLAM named DP-SLAM 2. This new algorithm provided an improved map representation and an improvement in the asymptotic efficiency of the algorithm.

In 2002, Montemerlo et al. [29] presented a particle filter-based algorithm

that they named FastSLAM. The algorithm used particle filters to represent different guesses about the robot's state. Each particle contained its own map representation, which was represented using an Extended Kalman Filter (EKF) for each landmark. In the sampling step, the state of the particle was updated using a motion model based on odometry. They conclude that their method works well for solving the SLAM problem by only using 0.3% of the parameters required of a similar EKF-SLAM approach. The following year, the same group presented a new and improved FastSLAM method, named FastSLAM 2.0 [30]. The algorithm was similar to the previous, except it integrates the most recent measurements in the sampling of the particle's state. They show significantly improved results compared to FastSLAM 1.0, with a relatively low pose error, even with a significant reduction in the number of particles.

Zhang and Singh [42] presented a method for low-drift and low-computational complexity, which they named LOAM. Their method did not necessitate high accuracy ranging or inertial measurements. Their key idea was to divide the solution to the SLAM problem into two different algorithms. One algorithm performs odometry at a high frequency but low fidelity. The other algorithm runs at a lower frequency to match and register the point cloud finely. They were able to achieve 0.55% translational error and 0.0013 $[deg \cdot m^{-1}]$ rotational error, which is the third-best result ever obtained on the KITTI data set [14].

Karimi et al. [23] recently presented a low-latency LiDAR SLAM framework. This framework, named LoLa-SLAM, was based on LiDAR scan slicing and concurrent matching. They proposed a two-dimensional roughness model to extract feature points for fine matching and registration of the point cloud. Furthermore, they utilized a pose estimator that acts as a temporal motion predictor, which assists in finding the feature correspondences in the map. This leads the utilized non-linear optimizer to have a fast convergence. The final pose fusion is calculated using an EKF. The results are compared to LOAM on a manual data set taken in a sports hall, where they exhibit slightly worse results with an RMSE of 0.039 [m], but with an update rate of 160Hz and 19ms latency. LOAM achieved 0.034 RMSE [m] but with a 20Hz update rate and 93ms latency.

Lenac et al. [24] presented a fast 3D pose-based SLAM system to estimate the trajectory of a vehicle. It did so by registering sets of planar surface segments extracted from point clouds provided by a 3D LiDAR. To process the point cloud efficiently, they project it to three two-dimensional images and apply image-based techniques to these. The proposed method was tested and compared to several state-of-the-art methods on two publicly available datasets, as well as an indoor and outdoor experiment. The authors suggest that the SLAM algorithm manages to improve the accuracy of vehicle trajectories significantly by providing an RMSE of 4.48m. However, it is difficult to assess what this number means without access to the specific data set.

Su et al. [38] proposed a novel method for solving the SLAM problem which they named GR-LOAM. The goal was to develop a method to estimate robot ego-motion by fusing LiDAR, inertial, and encoder measurements. They derived an odometer increment model that fused the measurements from the IMU and

encoder. This allowed them to estimate the robot pose variation on a manifold. Next, they applied point cloud registration and feature extraction to obtain distinct planar features. They also proposed an evaluation algorithm that could detect abnormal measurements and reduce their weights during optimization. They demonstrated high accuracy and robustness for state estimation of ground robots, even in complex environments.

Zhou et al. [43] presented a paper, which detailed their method for solving the SLAM problem. Their proposed method fused LiDAR and ultra-wideband sensor measurements. They estimated the sensor states by minimizing the sum of the Mahalanobis norm of all the measurement residuals. They performed two experiments to evaluate the performance of their fusion algorithm. They believe that these results "show that their algorithm can adjust the trade-off parameter according to the degeneration degrees of the sensor measurements" [43].

2.2 Semantic Segmentation

Image segmentation is an integral part of many computer vision systems. Image segmentation deals with dividing images into multiple objects or labeling each pixel as belonging to a specific class. Image segmentation has seen many applications, among others medical image analysis, autonomous vehicles, and augmented reality [28].

Image segmentation has been around for many years, dating back to classical computer vision systems. These days, almost all image segmentation algorithms rely on Deep Learning, as these models have shown remarkable performance on popular benchmarks, often reaching superhuman levels of performance [28].

In this literature review, only algorithms that are based on Deep Learning are considered, as these generally show superior performance when compared to classical vision techniques (techniques not involving neural networks).

Noh et al. [33] created a neural network built on top of the convolutional layers of the VGG-16 neural network. Their deconvolution network was composed of deconvolution and unpooling layers to identify pixel-wise class labels, as well as predict segmentation masks. The network first detected instance-level segmentation. These were then grouped into the final semantic segmentation. They believe that this approach frees them from scale issues present in the original FCN-based methods on which their work was based. The network demonstrated outstanding performance in the PASCAL VOC 2012 dataset, with an mIoU of 72.5% using a network that was not pre-trained.

In 2015, Ronneberger et al. [36] presented a neural network architecture named U-Net, so named because of its u-like structure. It was initially developed for image segmentation of medical images. The network was composed of two different parts; a contracting path and an expansive path. Each step of both the contractive and the expansive paths consisted of two convolutional layers. The output from each layer in the contractive path was concatenated with the input to the corresponding layer in the expansive path. They achieved excellent

results in the ISBI Cell Tracking Challenge 2015 leading to their victory in the challenge [36].

Visin et al. [41] proposed a structured prediction architecture, ReSeg, that was built on top of ReNet. The aim of the network is to exploit the local generic features extracted by traditional CNN's. Their model was also based on RNNs, whose properties enabled them to retrieve distance dependencies. The recurrent layer was implemented as a GRU unit, as the authors feel that these "strike a good balance between memory usage and computational power" [41]. Each RNN-layer is composed of 4 RNNs coupled together to capture both the local and the global spatial structure of the input data. They first processed the input image through the first layers of VGG-16 pre-trained on ImageNet. The output feature maps were then fed into one or more ReNet layers that sweep over the whole image. Lastly, they employed one or several upsampling layers to resize the final feature maps to the same resolution as the input image. They applied the softmax non-linearity to the output to predict the probability distribution over the classes for each pixel. They achieved 91.6% average mIoU on the Weizmann Horses data set.

Luc et al. [27] proposed an adversarial training approach for semantic segmentation. The architecture is built on VGG-16, excluding the last two max-pooling layers. These were excluded to maintain a higher resolution. Following the last convolutional layer, they included a "context module". This module was composed of eight convolutional layers with increasing dilation factors and was used to expand the field-of-view of the model while maintaining the resolution of the feature maps. They trained both a network for convolutional semantic segmentation and an adversarial network. The aim of the adversarial network was to discriminate between segmentation maps coming from either ground truth or from the semantic segmentation network. They achieved improved accuracy on both the Stanford Background and the PASCAL VOC 2012 data sets [27].

Shelhamer et al. [26] built a fully convolutional network that was able to take input of arbitrary size and produce an output of equal size. They attempted to adapt several contemporary classification networks, such as AlexNet, VGG, and GoogLeNet, into fully convolutional networks. They then transferred the representations learned by these networks through fine-tuning to the task of segmentation. They then defined a "skip architecture that combined the semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations" [26]. The networks achieved improved results on the segmentation task of PASCAL VOC 2012, with an IOU of 67.2%. Furthermore, their inference took one-tenth of a second for a typical image, but provide no further details to support this claim.

In 2016, Chen et al. presented a novel solution to the problem of segmentation, which they named DeepLab [5]. Their proposed network provided a solution to two of the technical hurdles present in applying DCNNs to the image labeling task. The first technical hurdle is signal downsampling. When you repeatedly combine max-pooling and downsampling operations (such as stride), the resolution of the network suffers. The second technical hurdle is spatial insensitivity. When repeatedly applying convolutions, you transform

low-level features into higher-level abstractions. This is one of the reasons deep CNNs (DCNNs) have consistently performed extremely well on classification tasks, but as more abstract representations of the input data are computed, you lose more and more spatial resolution. The problem of signal downsampling is bypassed through atrous convolutions, allowing efficient dense computation of DCNN responses. It even does so in a scheme that is substantially simpler than many of the earlier solutions to the same problem. They boosted the ability of the model to capture fine details by employing a fully connected Conditional Random Field. The proposed solutions showed significant improvements on the PASCAL VOC 2012 Test set, achieving an mIoU of 71.6%. The same group improved this network in 2016, where they presented their new and improved DeepLabv2 [6]. The network was almost identical to the DeepLabv1, with one important improvement; Atrous Spatial Pyramid Pooling (ASPP). In ASPP, you probe the incoming convolutional feature layers with filters at multiple sampling rates. This enabled the network to capture both objects and image context at multiple scales. Using a ResNet-101 as the backbone, this network reached a 79.7% mIoU score on the 2012 PASCAL VOC challenge and a 70.4% mIoU score on the Cityscapes challenge. They later added image-level features encoding global context to the ASPP module to boost performance in their DeepLabv3 model [7]. To handle segmenting objects at multiple scales, they employed atrous convolutions either in cascade or in parallel. This allowed them to achieve an mIoU of 81.3% on the Cityscapes data set while achieving 77.21% mIoU on the PASCAL VOC 2012 test set. The most recent version, DeepLabv3+, further adds to the architecture [8]. The addition is a decoder module that helps refine the segmentation results, particularly in the object boundaries. They explore the Xception model as the backbone, where they apply their idea of ASPP. This model achieved a performance of 82.1% on the Cityscapes data set and 89.0% mIoU on the PASCAL VOC 2012 test set.

2.3 SLAM in Formula Student

The SLAM problem was solved with success in a Formula Student environment by a team in the Swiss Federal Institute of Technology in Zurich (ETH Zürich) in 2018 [20].

They used Deep Learning to detect cones using both LiDAR and a stereo camera set-up. To estimate the cone positions, they used triangulation methods with stereo vision along with the LiDAR observations. For the Localization and Mapping, FastSLAM 2.0 was used, with a measurement model that included data from various sensors (Motor torque, Steering, GSS, IMU, GNSS, and WSS).

As a result, they managed to achieve an RMSE accuracy of 0.25 m for the cone positions, driving a track with a length of 230m.

3 Notations and Task Description

This section describes the notation and task description for the SLAM problem in general, focusing on the technical aspect. The notations used can be seen in table 1. Note that a subscript (e.g., s_t) refers to a single instance of a variable, while the superscript (e.g., s^t) typically refers to the full set of instances. The exception to this is Θ, which is used to refer to the full map. This notation is adopted from [31].

Variable	**Description**
s_t	pose of the robot at time t
s^t	complete path of the robot up to time t
θ_n	position of the n-th landmark
Θ	map, set of all n landmark positions
z_t	sensor observation at time t
z^t	set of all observations up to time t
u_t	robot control input at time t
u^t	set of all controls input up to time t
n_t	data association at time t
n^t	set of all data associations up to time t

Table 1: The notations used in this thesis, from [31].

3.1 The SLAM Problem

The SLAM problem has received enormous attention within the field of robotics research within recent years [29, 31]. The practical applications have continued to increase in impressiveness during the years, with algorithms covering increasingly large and more challenging environments [11]. The problem has seen several different applications, including underwater systems, outdoor systems and airborne systems [11]. Some consider SLAM a solved problem, both at a theoretical and conceptual level [11]. Nevertheless, several issues in practical realizations of SLAM solutions that can work in more general environments remain [11]. The solution of the SLAM problem is generally considered to be a fundamental prerequisite of genuinely autonomous robots [29, 31].

In general, SLAM addresses the problem of trying to construct a map of the environment of a mobile robot while simultaneously estimating its own pose relative to this map [31, 11, 30].

If no global position information is available, both the pose of the robot and the constructed map will drift over time [30]. The map may contain thousands of landmarks, meaning that acquiring said map can be formulated as a challenging statistical estimation problem [30]. As typically required of autonomous vehicles, a real-time constraint only adds to the difficulty of this problem.

If the robot had access to the actual map of the environment, the estimation of the robot path would be a trivial odometry problem. In the same way, if the

actual path of the robot was known, building a map of the environment would not provide as much difficulty [31].

3.1.1 Probabilistic Formulation

In probabilistic terms, the SLAM problem is expressed by the posterior [29]:

$$p(s_t, \theta | z^t, u^t, n^t) \tag{1}$$

This probability distribution is also referred to as the SLAM posterior [31]. It describes the joint posterior density of the robot pose, as well as the landmark locations given the observations and control inputs from time 0 to time t.

3.1.2 Motion and Measurement Model

According to Bresson et al. [4], filter-based SLAM methods are suited for online SLAM, which would be required in a Formula Student competition. They are an iterative process that consists of two steps: a prediction step and a correction step.

In order to calculate the posterior distribution (eq. (1)), the robot is provided with a probabilistic motion model that takes the form of a conditional probability distribution [29]:

$$p(s_t | u_t, s_{t-1}) \tag{2}$$

This distribution describes how a control input, u_t, provided at time $t-1$ affects the resulting pose [11, 29, 30]. This is the prediction step mentioned previously. In practice, different sensors may be implemented to approximate this model, such as an IMU. The motion model is usually a time-invariant probabilistic generalization of the robot kinematics [29]. Finding a good representation of this motion model is one challenge when implementing a filter-based SLAM algorithm.

The measurements made by the robot are governed by a probability distribution referred to as the observation model [29]:

$$p(z_t | s_t, \theta_{n_t}, n_t) \tag{3}$$

The measurement model describes the probability of making a specific observation, z_t, given both the current robot pose, the actual position of the landmark, and the correspondences [11]. This also requires knowledge of the uncertainty of the measurement model.

Both the motion model and the measurement model can be modeled by nonlinear functions with independent Gaussian noise [30]:

$$p(s_t | u_t, s_{t-1}) = h(u_t, s_{t-1}) + \gamma_t \tag{4}$$

$$p(z_t | s_t, \Theta, n_t) = g(s_t, \theta_{n_t}) + \epsilon_t \tag{5}$$

ϵ_t and γ_t are variables representing Gaussian noise with covariances R_t and P_t, respectively [30].

3.1.3 Landmark Estimates

Since neither the correct map nor the actual pose of the robot is known with certainty, the landmarks are estimated. The correlations between landmark estimates increase monotonically as more observations are performed. Therefore, the estimate of the relative location of landmarks continuously improves, regardless of robot motion [11].

The SLAM problem exhibits important conditional independence. In particular, if the path of the robot is known, the individual landmark measurements are independent of each other [29].

This conditional independence property allows us to factor the posterior in the following way [31]:

$$p(s^t, \Theta | z^t, u^t, n^t) = p(s^t | z^t, u^t, n^t) \prod_k p(\theta_k | s^t, z^t, u^t, n^t) \tag{6}$$

This factorization is precise and is always applicable to the SLAM problem [29].

Depending on the sensor setup, a robot typically senses more than one landmark at a time. The background material [31, 29, 30] generally assumes that only a single landmark is observed at each time-step.

4 Background

In this section, the background information necessary for the reader to understand the main points of this thesis is provided.

In section 4.1, the Formula Student competition is described.

Section 4.2 and section 4.3 present the theory of the Kalman, Extended Kalman, and Particle Filters, and are prerequisites to understanding the theory of the FastSLAM algorithms described in section 4.4.

Section 4.5 provides the background information regarding image segmentation required to understand the methods developed in the following section(s).

4.1 Formula Student Driverless

Formula Student (FS) is one of Europe's most well-established educational engineering competitions [34]. It is backed by several high-profile engineers and prides itself on the real-world engineering experience it provides [34]. It combines practical engineering and soft skills. The practical engineering manifests itself as a racing competition, while the soft events include business planning and project management. More than 100 university teams from all over the world participate in the competition each year [16].

The cars entering the competition are divided into three different categories, each of which has its own competition: Combustion Vehicle (CV), Electrical Vehicle (EV), and Driverless Vehicle (DV) [15].

Each of these competitions are divided into static and dynamic events. The dynamic events are [15]:

- Skid Pad (Figure of 8)
- Sprint
- Acceleration
- Endurance
- Fuel Economy

The first driverless competition took place in 2017 [16]. The driverless track is demarcated using cones, which can be seen in fig. 2.

The DV competition includes each of the dynamic events above, but includes an additional event: trackdrive. In the trackdrive event, the DV is expected to navigate an unknown course. The vehicle has two runs, and the time of the best run is the recorded attempt. Each run consists of ten laps around the course [15]. The vehicle must be able to localize itself within the track and map out the track while driving to successfully navigate. Both of these have to be performed simultaneously; this is a classic example of the SLAM problem mentioned previously.

Figure 2: The four cones used in Formula Student Driverless. From the left: Start-cone, left-cone, right-cone and end-cone.

The trackdrive layout is a circuit including a full loop, and adheres to the following guidelines [15]:

- Straight segments must not exceed 80m in length
- Constant turns must have a diameter of no more than 50m
- The outside diameter of a hairpin turn must be at least 9m
- The width of the track must be at least 3m
- A single lap is approximately between 200m and 500m

Time is not the only factor affecting the score of a trackdrive. Penalties are given for knocking over cones [15], making it essential to correctly map the course and localize the vehicle with high accuracy during the race. The penalty also needs to be considered for the control system, as the car needs to have a safe margin to the cones since the effect of hitting a demarcating cone could potentially be catastrophic. The localization algorithm might not be able to correct for cones that are out-of-place or missing from the course.

4.2 Kalman Filter

The Kalman Filter (KF) is a technique used for filtering and prediction in continuous linear systems [39].

The KF is one of the most valuable tools for estimating state [9]. It can estimate the state of a dynamical system recursively, and can function even when noise is present [9]. The KF consists of two steps; a prediction step and a correction step. In the prediction step, the next state probability $p(x_t \mid u_t, x_{t-1})$ is calculated. In the correction step, the measurement probability $p(z_t \mid x_t)$ is calculated [39].

The Kalman Filter has one important assumption; that both the next state probability (eq. (7)) and the measurement probability (eq. (8)) are linear functions with Gaussian noise [39]:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t \tag{7}$$

$$z_t = C_t x_t + \delta_t \tag{8}$$

A Kalman filter maintains an estimate of both the state vector and error covariance matrix, meaning that the output of a Kalman filter is a Gaussian probability density function [9].

Applied to the problem of localization, the output of a Kalman filter is a distribution of likely robot states, rather than a single estimate of the state [9]. At each moment in time, the KF represents this belief of a state by a mean, μ_t, and a covariance, Σ_t.

A pseudo code of the algorithm is shown in algorithm 1 [39]:

Algorithm 1 Kalman Filter Algorithm from [39]

input : $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$
$\bar{\mu_t} = A_t \mu_{t-1} + B_t u_t$
$\bar{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
$K_t = \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1}$
$\mu_t = \bar{\mu_t} + K_t (z_t - C_t \bar{\mu_t})$
$\Sigma_t = (I - K_t C_t) \bar{\Sigma}_t$
return μ_t, Σ_t

The variable K_t is referred to as the Kalman gain and specifies the importance of the most recent measurement when calculating the estimate of the new state [39].

In the last step, the covariance of the posterior belief is calculated. This requires adjusting for the information gain that is obtained from incorporating the measurement [39].

4.2.1 Extended Kalman Filter

The standard Kalman filter cannot be used on non-linear systems, but a simple solution is to linearize the non-linear equations of the dynamic system. This

technique is referred to as the Extended Kalman Filter (EKF) [9]. The next state probability, x_t, and the measurement probability, z_t, are governed by non-linear functions g and h [39]:

$$x_t = g(u_t, x_{t-1}) + \epsilon_t \tag{9}$$

$$z_t = h(x_t) + \delta_t \tag{10}$$

This linearization can be performed using a first-order Taylor approximation [39]:

$$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + \underbrace{g'(u_t, \mu_{t-1})}_{G_t}(x_{t-1} - \mu_{t-1}) \tag{11}$$

$$h(t) \approx (\bar{\mu}_t) + \underbrace{h'(\bar{\mu}_t)}_{H_t}(x_t - \bar{\mu}_t) \tag{12}$$

The algorithm for EKF can be seen in algorithm 2 [39]:

Algorithm 2 Extended Kalman Filter Algorithm from [39]

input : $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$
$\bar{\mu}_t = g(u_t, \mu_{t-1})$
$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
$K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1}$
$\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
$\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t$
return μ_t, Σ_t

4.3 Particle Filters

The Particle Filter algorithm, introduced in 1993, uses a numerical approximation to the non-linear filtering problem [21]. They differ from many other filtering methods (e.g., Kalman Filter). Instead of the distribution being represented in parametric form, it is represented by random samples, known as particles, drawn from the distribution. Since this representation is non-parametric, it has the advantage that it is not limited by Gaussians [39], as opposed to e.g. the Kalman Filter. Still, it allows approximation of any probability distribution, even in the case of nonlinearity and non-stationarity, which is particularly useful when the true posterior is difficult or impossible to calculate [10]. Each particle is assigned a weight, which describes this probability of the particle being sampled from the probability density function [10, 39, 22]. Resampling of the particles can then be performed between timesteps [39].

Particle Filters have many differing applications. One of these applications is Monte Carlo Localization (MCL), in which a particle filter is applied to the problem of robot pose estimation [29]. The primary idea of MCL is to evolve

the set of particles, where each particle represents a guess of the state, so that it accurately reflects the actual state [29].

The basic algorithm of the particle filter can be seen in algorithm 3 [39]:

Algorithm 3 Particle Filter algorithm from [39]

input : $\mathcal{X}_{t-1}, u_t, z_t$
$\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
for *m = 1 to M* **do**
  sample $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$
  $w_t^{[m]} = p(z_t|x_t^{[m]})$
  $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
end
for *m = 1 to M* **do**
  draw i with probability $\propto w_t^{[i]}$
  add $x_t^{[i]}$ to $\mathcal{X}_t$
end
return $\mathcal{X}_t$

The PF constructs the belief of the actual state recursively from the belief about the true step at the previous time step [39].

4.3.1 Resampling

The resampling step in Particle Filters is crucial. If the particles to be resampled are chosen randomly by the weights assigned to them, a significant loss of diversity can be introduced to the particle set [39]. This error is critical when Particle Filters are used in localization. An extreme example is described in [39], where there comes a time interval where it cannot localize (e.g., due to missing observation or sensor failure). In that case, all the particles will have very similar weights assigned to them. If chosen randomly multiple times, particle deprivation can occur. Eventually, a single particle will take over, causing the localization algorithm to fail since there is no guarantee that the chosen particle represents the correct state [39].

A solution to this resampling problem is the Low Variance Resampling method described in algorithm 4 [39].

Algorithm 4 Low_variance_sampler from [39]

input : $\mathcal{X}_t, w_t$
$\bar{\mathcal{X}}_t = \emptyset$
$r = rand(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
for *m = 1 to M* **do**
  $u = r + (m-1) \cdot M^{-1}$
  while $u > c$ **do**
    $i = i + 1$
    $c = c + w_t^{[i]}$
  end
  add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
end
return $\bar{\mathcal{X}}_t$

Another important factor is the choice of when to resample the particles. That choice is not trivial and requires practical experience [39]. If the resampling is performed too often, there is a risk that the particles will lose their diversity. On the other hand, if the resampling is not done frequently enough, a great portion of the particles will likely end up in regions with a low probability [39]. The effective particle number, n_{eff}, seen in eq. (13) is a measure of how many particles in the set contribute to the solution of the problem [22].

$$n_{eff}(t) = \left(\sum_{i=1}^{M} (w_t^{(i)})^2 \right)^{-1} \tag{13}$$

A good rule of thumb is to resample when $n_{eff} < \frac{2N}{3}$ [22].

4.4 FastSLAM

In this thesis, FastSLAM 2.0 is used to solve the SLAM problem. For more details, we refer to the work by Thrun et al. [31], where the information, along with the equations and notations in this section (section 4.4 and subsections) is from.

The FastSLAM algorithms are particle filter-based algorithms where each particle has its own pose and representation of the map environment. The representation of the map is represented with EKF's, one for each landmark. A single particle in the set takes the following form:

$$S_t^{[m]} = \left\langle s^{t,[m]}, \theta_1^{[m]}, \theta_2^{[m]}, ..., \theta_N^{[m]} \right\rangle \tag{14}$$

Where $S_t^{[m]}$ is a particle with an index of m, $s^{t,[m]}$ is the path estimate of the particle, and $\theta_n^{[m]}$ is a single landmark estimate of the particle, defined by a mean and a covariance matrix.

The critical idea of FastSLAM depends on the fact that the position estimates of the landmarks are conditionally independent of each other, given the robot pose. If the probability distributions are considered independent, the combined probability can be treated as the product of all of their individual probabilities (eq. (6)). If conditioned on the robot path, the landmarks can be treated independently of each other (see section 3.1.3).

The FastSLAM algorithm consists of four steps:

1. Update the pose of each particle by sampling from the proposal distribution
2. Update the mean and covariance of each landmark
3. Calculate the importance weights for the particles
4. Perform particle resampling

Each step will be described in further detail below.

4.4.1 Sampling the pose

The first step of the FastSLAM algorithm is to sample a new pose for each particle. In FastSLAM 1.0 the motion model described in section 3.1.2 is used directly as the sampling distribution:

$$s_t^{[m]} \sim p(s_t \mid u_t, s_{t-1}^{[m]}) \tag{15}$$

The FastSLAM 2.0 algorithm extends this sampling distribution, which is described in section 4.4.5.

4.4.2 Update the Landmarks

In each iteration the landmarks for each particle are updated, where the landmark estimate can be modelled as a probability distribution:

$$p(\theta_{n_t} \mid s^t, z^t, u^t, n^t) \tag{16}$$

Determining this probability distribution can be a challenging task since the landmark associations are not necessarily known. The data association problem can be addressed in various ways depending on the problem. All data associations are assumed known in this section.

To assess the quality of an observation it is compared to an observation estimate (eq. (17)) which is computed given a state and a previous landmark estimate:

$$\hat{z}_t = g(s_t^{[m]}, \mu_{n_t,t-1}) \tag{17}$$

This non-linear measurement model can be estimated using a first-order Taylor approximation, as described in section 4.2.1. Since the robot path is

fixed, this Taylor expansion is only affected by the landmark estimates (θ_{n_t}). This approximation can be seen in eq. (18).

$$g(s_t^{[m]}, \theta_{n_t}) \approx \hat{z}_t + G_{\theta_{n_t}}(\theta_{n_t} - \mu_{n_t,t-1}^{[m]}) \tag{18}$$

Where $G_{\theta_{n_t}}$ is the derivative of the measurement model with respect to the landmark:

$$G_{\theta_{n_t}} = \nabla_{\mu_{n_t,t-1}^{[m]}} g(s_t^{[m]}, \mu_{n_t,t-1}^{[m]}) \tag{19}$$

Equation (20) to 23 are used to update the mean and the covariance for landmarks. This is identical to the update step in conventional EKF's described in section 4.2.1.

$$Z_{n,t} = G_{\theta_{n_t}} \Sigma_{n_t,t-1}^{[m]} G_{\theta_{n_t}}^T + R_t \tag{20}$$

$$K_t = \Sigma_{n_t,t-1}^{[m]} G_{\theta_{n_t}}^T Z_{n,t}^{-1} \tag{21}$$

$$\mu_{n_t,t}^{[m]} = \mu_{n_{t,t-1}}^{[m]} + K_t(z_t - \hat{z}_t) \tag{22}$$

$$\Sigma_{n_t,t}^{[m]} = (I - K_t G_{\theta_{n_t}}) \Sigma_{n_t,t-1}^{[m]} \tag{23}$$

4.4.3 Calculating the importance weights

When estimating the pose of a particle, the algorithm samples from the proposal distribution $p(s^t \mid z^{t-1}, u^t, n^{t-1})$, while the desired posterior is the target distribution given by $p(s^t \mid z^t, u^t, n^t)$. This can be seen in fig. 3.

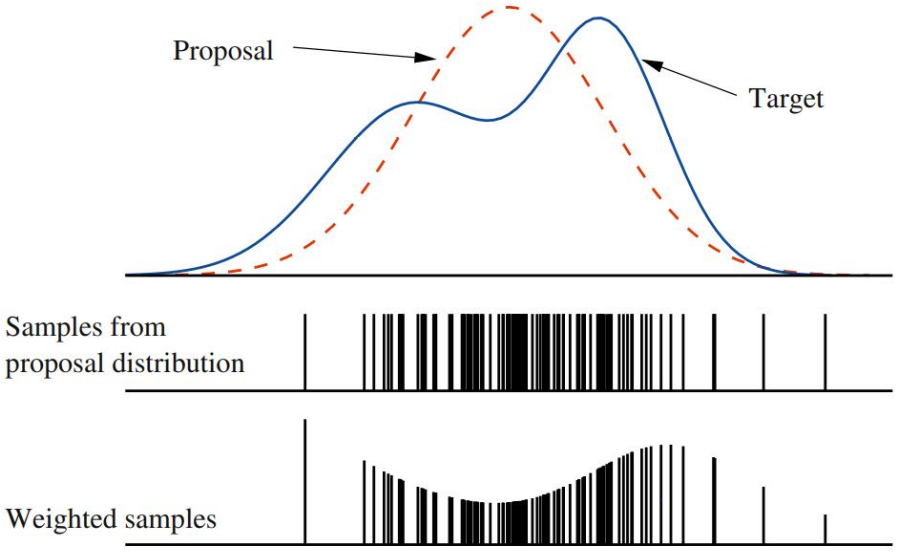


Figure 3: A visual representation of the particle weights. The samples are drawn from the proposal distribution, and assigned weights calculated using the target distribution. Figure taken from [31].

The difference between these two distributions are corrected for by calculating an importance weight for each particle. Since the landmark estimates are represented as EKF's, the importance weights can be computed in closed form:

$$w_t^{[m]} = \frac{1}{\sqrt{|2\pi Z_{n_t,t}|}} \exp\left(-\frac{1}{2}(z_t - \hat{z}_{n_t,t})^T [Z_{n_t,t}]^{-1}(z_t - \hat{z}_{n_t,t})\right) \tag{24}$$

The importance weight of a particle is the combined probability of observing each of the landmarks given a particular state estimate.

4.4.4 Resampling

When each particle has been assigned a weight, new samples are drawn from the proposal distribution. This step is no different from the resampling step in conventional particle filters, as described in section 4.3.1.

4.4.5 FastSLAM 2.0 Proposal Sampling

In many practical cases, such as when using a LiDAR, the observation model is typically more accurate than the motion model. This causes FastSLAM 1.0 to diverge given a limited number of particles. Figure 4 shows an example of this problem. There is a mismatch between the proposal sampling distribution and the target distribution. Having a noisy motion model requires a lot of particles, and only a small amount of them will be used in the resampling step.

Figure 4: Figure showing a mismatch between the proposal distribution (all the particles) and the target distribution (the blue ellipse). Only the particles within the blue ellipse will be used in the resampling step. Figure taken from [31].

FastSLAM 2.0 solves this problem by incorporating the current observation into the proposal distribution. Instead of drawing a sample from the standard motion model in eq. (15), the new samples will be drawn from:

$$s_t^{[m]} \sim p(s_t \mid u_t, s_{t-1}^{[m]}, n_{t-1}, z_t) \tag{25}$$

where the sensor measurements z_t along with the data associations n_t are incorporated. In order to determine this distribution, a new motion model needs to be defined:

$$g(s_t, \theta_{n_t}) \approx \hat{z}_t + G_\theta(\theta_{n_t} - \mu_{n_t,t-1}^{[m]}) + G_s(s_t - \hat{s}_t) \tag{26}$$

where G_s is the derivative of the measurement model with regards to the state of the particle, $\hat{s}$ is the predicted state for a given particle, and $\hat{z}$ is the predicted location of the particular landmark as seen from that particle.

Now the mean (eq. (28)) and the co-variance (eq. (27)) of the new sampling distribution can be computed.

$$\Sigma_{s_t}^{[m]} = \left[G_s^T Z_t^{-1} G_s + P_t^{-1} \right]^{-1} \tag{27}$$

$$\mu_{s_t}^{[m]} = \Sigma_{s_t}^{[m]} G_s^T Z_t^{-1} (z_t - \hat{z}_t) + \hat{s}_t^{[m]} \tag{28}$$

4.4.6 Data association

In most practical applications of SLAM, the data associations are unknown. The uncertainty of an observation z_t corresponding to a specific landmark θ_n is not only dependent on the uncertainty of the measurement model but also the motion noise.

Wrong data associations can significantly impact the performance of a SLAM algorithm. Wrong data associations due to measurement noise will over-estimate the covariance of specific landmarks while under-estimating the covariance of others without significantly affecting the robot path. Wrong data associations due to motion noise may have a more severe impact. In cases where multiple measurements are made per timestep, making a single wrong data association will risk the other observations being associated with the wrong landmarks with a high probability.

One of the advantages of FastSLAM, compared to e.g. EKF-SLAM, is that a single data association hypothesis is made for each particle. This means that in theory, the particles that have the wrong data associations should receive lower weights and thus vanish in the resampling step.

Since the landmarks are represented by EKFs, an observation z_t has the maximum likelihood of belonging to the landmark estimate $\hat{z}_{n,t}$ that yields the highest probability using eq. (29).

$$p(z_t \mid s^{t,[m]}, z^{t-1}, u^t, n^{t-1}) = \frac{1}{\sqrt{|2\pi Z_{n,t}|}} \exp\Big(-\frac{1}{2}(z_t - \hat{z}_{n,t})[Z_{n,t}]^{-1}(z_t - \hat{z}_{n,t})\Big) \tag{29}$$

4.4.7 Time Complexity of FastSLAM

Processing each landmark in a particle is a constant time operation since the landmarks are represented by EKFs, typically represented in low dimensions. This is an advantage when having a rather large amount of landmarks over other types of SLAM algorithms, e.g., EKF SLAM, which has a quadratic complexity in increasing landmarks.

The second factor that affects the time complexity of FastSLAM is the number of particles. The time complexity grown linearly with the number of particles. This factor cannot be reduced since all particles need to be processed in each timestep.

The time complexity of the FastSLAM algorithm is $\mathcal{O}(M \cdot N)$ where M is the total number of particles and N is the total number of landmarks. The time complexity can be reduced to $\mathcal{O}(M \cdot \log N)$ by saving the landmarks in a tree structure. This runtime after this optimization can be seen in fig. 5.

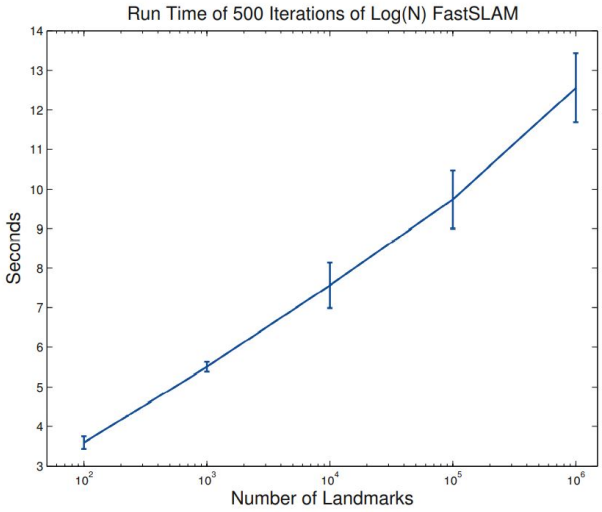


Figure 5: The runtime of the FastSLAM 2.0 algorithm where the landmarks are saved in a tree structure. The graph shows the results of the algorithm running in a simulated environment. Figure taken from [31].

4.4.8 Adding new landmarks

When the data associations are unknown, all new observations need to either be associated with an existing landmark or added as a new one. This can be done by using eq. (29), after defining a threshold p_0, that can be used to distinguish between these two cases.

When adding a new landmark, the mean and covariance can be computed as follows:

$$\mu_{n_t,t}^{[m]} = g^{-1}\left(s_t^{[m]}, z_t\right) \tag{30}$$

$$\Sigma_{n_t,t}^{[m]} = \left(G_{\theta_{n_t},t} R^{-1} G_{\theta_{n_t},t}\right)^{-1} \tag{31}$$

4.5 Semantic segmentation

Image segmentation generally takes on one of two forms. The first is to formulate it as a classification problem, where a class/category is assigned to each pixel. This is also known as semantic segmentation. The second is to divide an image into individual object classes, which is known as instance segmentation [28].

Today, almost all image segmentation algorithms rely on Deep Learning, as these models have shown remarkable performance on popular benchmarks, frequently reaching superhuman levels of performance [28].

Deep Learning relies on training on generally large data sets. If such data sets are not available, training a neural network from scratch might cause problems

of overfitting and convergence [25]. To solve this problem, transfer learning can be utilized.

4.5.1 Transfer Learning

Transfer learning is a technique where you attempt to improve a learning algorithm for one domain, the target domain, by transferring information from another related domain, known as the source domain [25].

Deep Learning, in general, consists of multiple network layers, which can learn representations of data at different abstraction hierarchies. In the field of computer vision, these layers are typically convolutional neural network layers. To train such a network, especially if the depth is reasonably high, requires significant data [25]. Transfer learning is helpful if the source domain contains significantly more data than the target domain. The idea is then that the good representations for the source domain are also helpful for the target domain. This idea is plausible since many visual categories share low-level abstractions, such as edges, geometric shapes, and changes in lighting [19].

There are generally two fundamentally different approaches to transfer learning. The first is to use a pre-trained model directly as a feature extractor. Several studies demonstrate that the generic descriptors mentioned previously extracted from pre-trained CNN models are pretty general and can be used when recognizing and locating objects from raw images [25]. The second approach is to use fine-tuning, where you train several or all network layers that have been pre-trained on one data set, on a different, smaller data set in a supervised manner [25].

4.5.2 U-Net

U-Net is a popular choice in image segmentation and is described in [36]. As the name suggests, it has a U-like shape, where all the layers include 3x3 double convolutions, each followed by a ReLU activation function. The contracting path of the network (left side of the network in fig. 6) uses 2x2 max pooling for downsampling, with a stride of 2, where the number of feature channels are doubled at each step. The expansive path (right side of the network in fig. 6) consists of upsampling of the feature map, followed by 2x2 up-convolution for halving the number of feature channels, followed by a concatenation of the feature map in the corresponding layer in the contracting path. The architecture of U-Net is straightforward and easy to both implement and modify, compared to more complicated networks (e.g., DeepLabv3 section 4.5.3)

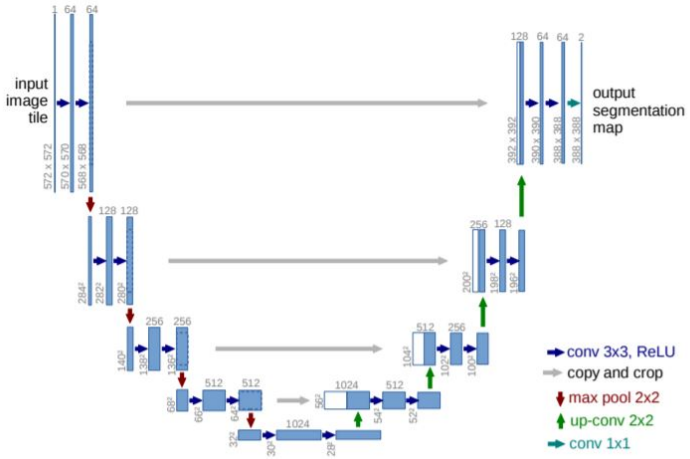


Figure 6: The U-Net architecture presented by Ronneberger et al. Figure taken from [36].

4.5.3 Deeplabv3

DeepLabv3 (henceforth referred to as DeepLab), developed by Chen et al., is a state-of-the-art Neural Network used for image segmentation [7]. In neural networks, performing consecutive pooling operations or convolution striding is the norm. While this allows the network to learn increasingly abstract feature representations, it also has the downside of reducing the feature resolution [7]. DeepLab employs the concept of atrous convolutions to partially solve this problem, which has previously proved effective in semantic image segmentation [7]. Atrous convolution allows DeepLab to be repurposed from networks pre-trained on ImageNet to extract denser feature maps. This is done by removing the downsampling operations from the last few layers while upsampling the corresponding filter kernels [7]. One is thereby able to control the resolution at which feature responses are computed within the network, but without learning any extra parameters [7].

Atrous convolution, also known as dilated convolution, introduces the dilation rate as an additional parameter to convolutional layers [28].

Minaee et al. describe that: "the dilated convolution of a signal, $x(i)$, is defined as" [28]:

$$y_i = \sum_{k=1}^{K} x[i + rk]w[k] \tag{32}$$

r corresponds to the dilation rate, which defines a spacing between the weights of the kernel (w) [7, 28]. To use dilation is equivalent to convoluting the input filters having $r - 1$ zeroes between any two consecutive filter values along each spatial dimension [7]. This can be seen in fig. 7.

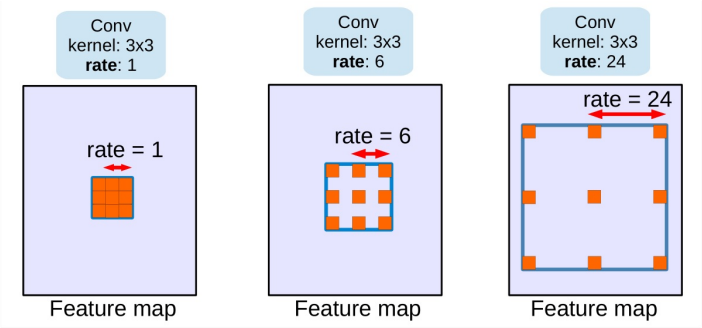


Figure 7: Atrous convolution with kernel size 3 x 3 and different atrous rates. Figure taken from [7].

Performing dilated convolution with a rate of $r = 1$ is equivalent to standard convolution. The field-of-view of the model increases monotonically with the atrous rate, enabling object encoding at multiple scales [7].

Considering a $3x3$ kernel with a dilation rate of 2, the size of the receptive field will equal that of a $5x5$ kernel. It does so by using only 9, as opposed to 25, parameters. The receptive field can thereby be increased without an increase in computational cost.

5 System Design

This section details the system's overall design and the considerations that went into choosing the system's building blocks. It is intended as a high-level view of the pipeline and does not include detailed information. More detail will be provided in the following sections.

5.1 Choice of range and color sensor

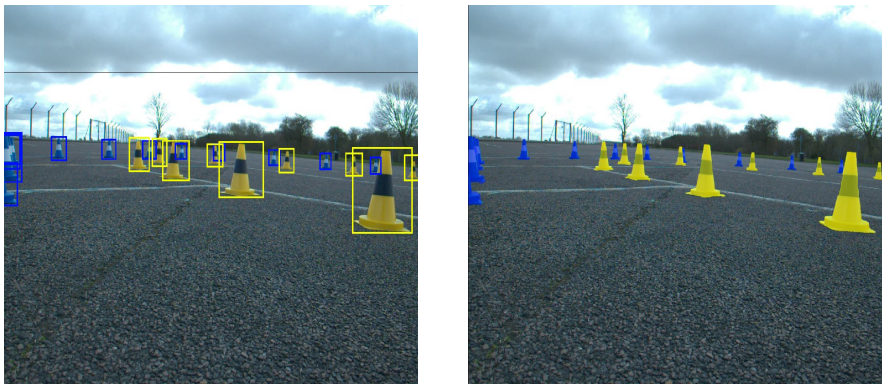
As stated previously, a LiDAR scanner and a camera are available for this project.

A LiDAR scanner can estimate 3D location much more accurately than most modern cameras, which is a clear advantage. However, distinguishing the cones from each other may be more challenging using the LiDAR points since they are sparse, and their reflection values are not as descriptive as the RGB values from the camera. For these reasons, a camera is chosen to distinguish the cones, and a LiDAR scanner is chosen to estimate the 3D location of the cones.

5.2 Choice of sub-systems

There are several challenges when combining information from point clouds and images. Some of these challenges were addressed in [20], where the authors combined several classical computer vision algorithms, as well as deep learning, to estimate the 3D location of the cones in a stereo camera setup. The central technique of the approach was to extract bounding boxes of the cones in the image and then perform multiple post-processing steps to determine both the 3D location and the color of the cones.

In fig. 8a it can be seen that when performing object detection in a Formula Student environment, large regions of the classified objects belong to the background or overlap with other classes. This means that one or several post-processing steps are needed in order to determine the actual 3D location of the cones. In this thesis, an entirely different and arguably much simpler solution is proposed, that requires only a single pass over the image. The solution relies on the fact that it is relatively simple to compute the transformation from a LiDAR to the image plane. With the LiDAR coordinate system as a reference, only the camera parameters (extrinsics and intrinsics) are needed. Knowing these parameters the 3D points can be projected from the point cloud to the image plane. This makes it possible to do a fast and efficient point-wise classification of the point cloud by seeing which pixel a point corresponds to. A semantic segmentation method is needed to perform this pixel-level classification, as seen in fig. 8b. All points labeled as background can then be removed, and only the points belonging to a cone remains, given that the transformation is accurate.



(a) (b)

Figure 8: When using classical object detection (fig. 8a), large regions of the classified objects are either belonging to background, or overlap with other classes. Using a pixel-wise prediction (Semantic Segmentation) solves this challenge (fig. 8b).

When choosing the preprocessing of the LiDAR points belonging to the cones, two options were considered. Either estimate the center of the cone as a single point for the SLAM algorithm to process, or input all the points belonging to a cone into a SLAM method designed to build an occupancy grid map. The advantage of using an occupancy grid map is that it does not require any predefinition of landmarks [31]. Even though clear information about data association is not available, the cones are always clearly separated on a Formula Student track. Therefore, the cone-center estimation is performed separately to reduce the amount of data for the SLAM algorithm to process

To summarize, fig. 9 shows the division of the system into three separate sub-systems.

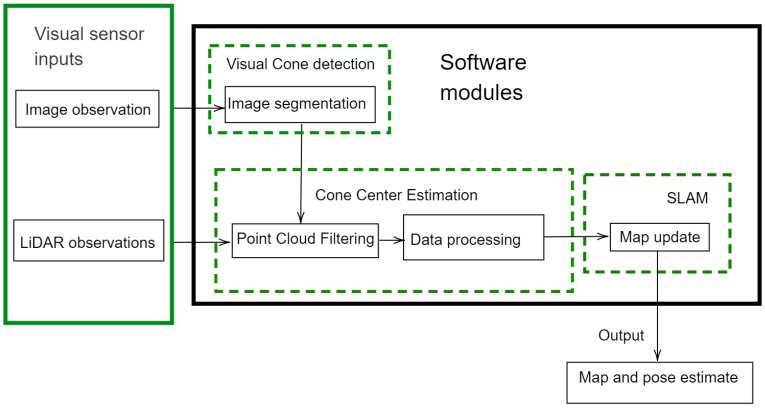


Figure 9: An overview of the system design. The green solid rectangle indicate sensor inputs and the green dashed rectangles indicate the sub-systems designed in this project.

5.3 Hardware

The hardware chosen for this project was a VLP-16 LiDAR scanner and a Basler acA1440-220um camera. The standard sampling rate for each is 10Hz, which is kept throughout the project.

The lens on the camera is a Basler Lens C125-0418-5M-P F4mm.

The car used is Traxxas X-maxx2 [3]. The top plate of the vehicle proved not to be an optimal position for the LiDAR scanner due to the size of the cones. Therefore, an extension plate was designed for the LiDAR (fig. 10).

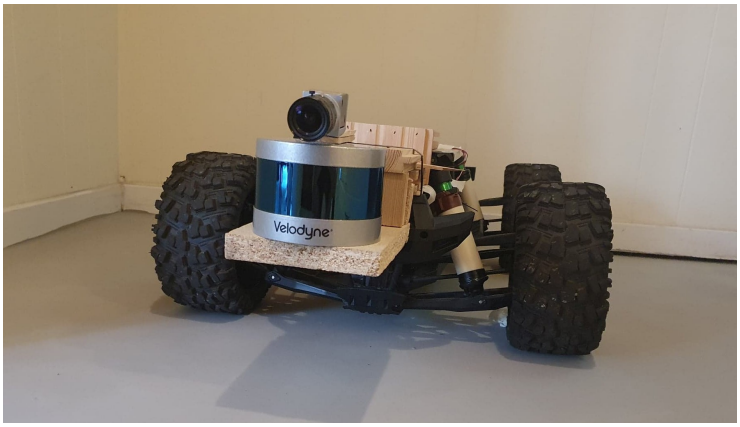


Figure 10: The Traxxas X-Maxx2 vehicle. The VLP-16 LiDAR and the Basler camera were mounted on top of an extension plate.

5.4 Complete System Design

This project aims to design a system capable of solving the SLAM problem in a Formula Student environment. That is, however, just a sub-task of the Driverless Formula Student competition since it also requires path planning and robot control. A high-level proposal of a system design is shown in fig. 11.

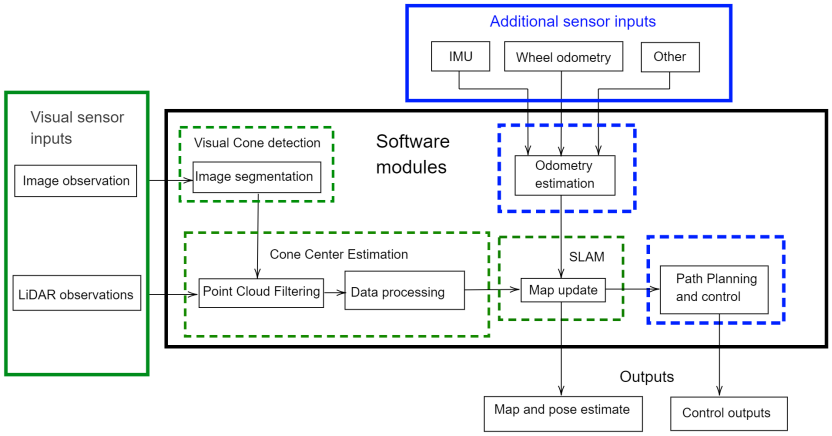


Figure 11: The complete pipeline for the autonomous driving system incorporating the SLAM system developed in this project. The green rectangles indicate the modules included in the project, while the blue rectangles indicate what is needed for the final design. The solid rectangles indicate sensor inputs, while the dotted rectangles indicate software modules.

6 Cone Detection and Classification

This section describes how a cone center and its predicted class were estimated given a camera and a LiDAR observation. It includes the modules marked in orange in fig. 12.

In section 6.1, the process of fusing the camera and the LiDAR is presented.

Section 6.2 describes the data collection and augmentation methods used for training the neural networks described in section 6.3.

Finally, section 6.4 describes the method used to estimate the center of the cones given the point cloud and segmented mask. This is represented by the lower orange block in fig. 12.

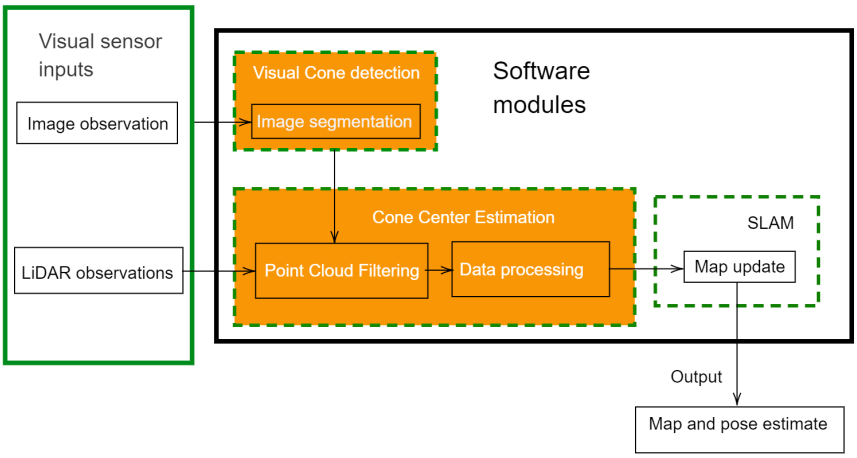


Figure 12: The overall system design, with the modules explained in this section highlighted in orange.

6.1 Sensor Fusion

To warp the 3D points obtained from the LiDAR to the camera's image plane, the homogeneous transformation from the LiDAR to the camera, the camera's intrinsic parameters, and the distortion coefficients of the lens need to be estimated.

In fig. 13 the different coordinate systems of the LiDAR and the camera can be seen.

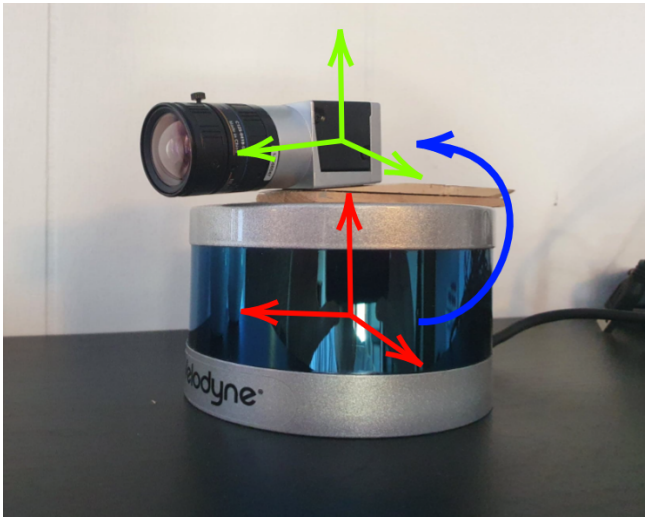


Figure 13: The coordinate systems of the LiDAR and the camera. The blue arrow indicates a transformation between the two coordinate systems.

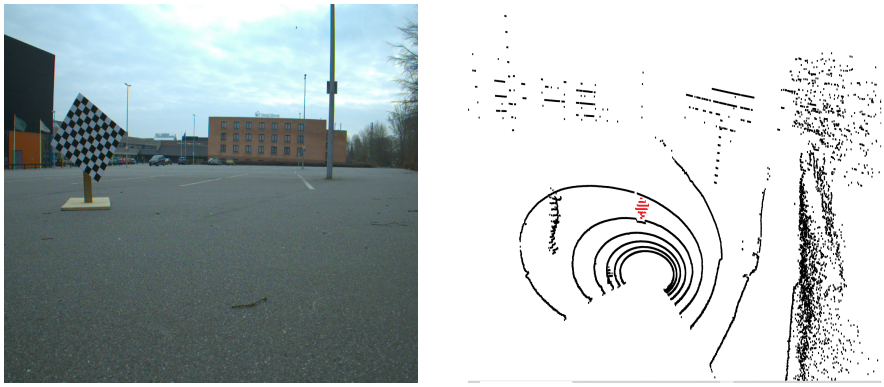
Camera calibration is performed to find the camera's intrinsics, along with the distortion coefficients. The intrinsics are calculated using sixteen images by using the MatLab camera calibration toolbox. The images used can be seen in fig. 14.

Figure 14: The sixteen images used for the camera calibration, to find the camera's intrinsic parameters and distortion coefficients.

The Camera 3D points $\begin{bmatrix} X_c & Y_c & Z_c \end{bmatrix}^T$ can be computed using the LiDAR observations $\begin{bmatrix} X_L & Y_L & Z_L \end{bmatrix}^T$ (eq. (33)).

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \tag{33}$$

This transformation is calculated by recording 45 images and corresponding point clouds, each containing a checkerboard. The checkerboard is 50x45 cm, with a square size of 5x5 cm. Then the corners of the checkerboard are estimated in both the point cloud and the image. The parameters that most accurately describe the transformation seen in eq. (33) are then calculated. The calculation itself was performed using MatLab with the Lidar Toolbox. See appendix B for the specific parameters. A raw image of the checkerboard can be seen in fig. 15a, and a raw point cloud observation can be seen in fig. 15b.



(a) (b)

Figure 15: A single measurement including both an image (fig. 15a) and a point cloud (fig. 15b), captured simultaneously.

The projection matrix can then be calculated using the extrinsic homogeneous transformation and the camera intrinsics, making it possible to estimate a pixel value corresponding to a 3D LiDAR observation. This requires the image to be rectified using the lens distortion coefficients. The specific parameters can be found in appendix B. This projection can be seen in eq. (34).

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ 1 \end{bmatrix} \tag{34}$$

Where s is a scaling factor.

6.2 Data Acquisition

To train the neural network used for semantic segmentation, data and corresponding labels are needed.

In data collection, 885 images of cones were collected and manually labeled with the four classes shown in fig. 2.

Sixteen images, along with their hand-labeled masks, chosen randomly from the dataset, are shown in fig. 16.

Figure 16: Sixteen images from the data chosen at random, along with their manually labelled masks. The colors are randomly generated, and do not correspond to the actual color of the cone.

Since the data set is relatively small, with regards to the number of parameters most state-of-the-art networks use, different data augmentation methods were used. Flipping images is a standard data augmentation method, but in this case, it could hurt the training process since the yellow cones are always on the right side of the blue cones when aligned on a track. Therefore, only the images that were not taken on a track were flipped, resulting in 1080 images. The data augmentations performed were randomly chosen from:

- Random cropping
- Gaussian noise

- Color jitter noise
- Random affine transformation
- Speckle noise
- Erasing small regions of the image
- Salt and pepper noise

Specific parameters and the probabilities of choosing a specific augmentation can be seen in the code [1]. Eighteen different augmentations chosen using these parameters on the same image can be seen in fig. 17.

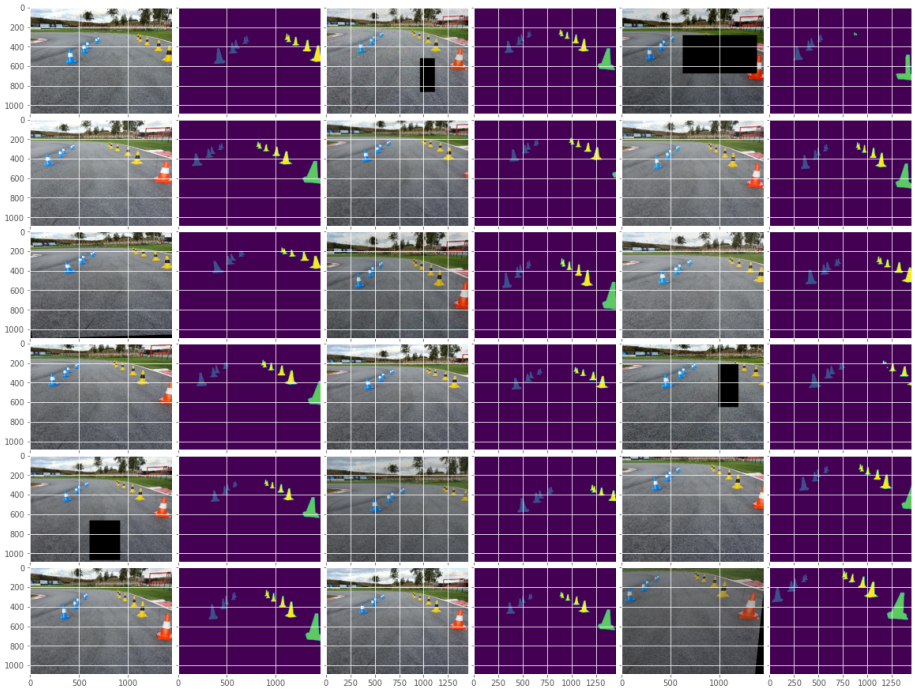


Figure 17: Different types of data augmentation methods were implemented to prevent over-fitting. This figure shows the result of augmenting the same image 18 times with different parameters, as described in [1].

6.3 Semantic Segmentation Methods

When choosing a semantic segmentation algorithm, there are more things to consider than just the accuracy. For example, when designing a fully autonomous driving system in a real-world environment, such as a Formula Student race track, runtime is an essential factor.

Logically, locating cones in an image is a simple problem compared to the pre-existing benchmarks used to validate other segmentation networks, such as Coco, Cityscapes, or Pascal VOC. However, it is difficult to say how much more straightforward the problem is and use this information to inform the choice of network architecture. For this reason, several network designs are implemented and compared (section 8.1). All of the neural networks were implemented using PyTorch. U-Net was implemented along with two modified versions, where some of the layers were removed. The network architectures can be seen in appendix A.

To compare these networks with a current state-of-the-art network, DeepLabv3 was used. The DeepLabv3 used is the official PyTorch implementation [35], constructed with a ResNet-101 backbone and pre-trained on the COCO dataset. The output layer of the DeepLab module was replaced with a convolutional layer using a kernel size of 5 (one for each class, along with the background). The model was then fine-tuned on the data set.

The data set was split into three subsets: training, validation and test as shown below:

Training	850
Validation	133
Test	100
Total	1083

Each network was trained using the training set and validated on the validation set using the Cross Entropy loss function [35]:

$$loss(x, class) = -log\left(\frac{exp(x[class])}{\sum_j exp(x[j]}\right) \tag{35}$$

The Adam optimizer was used with default parameters to update the weights of the network. More details can be found in the code [1].

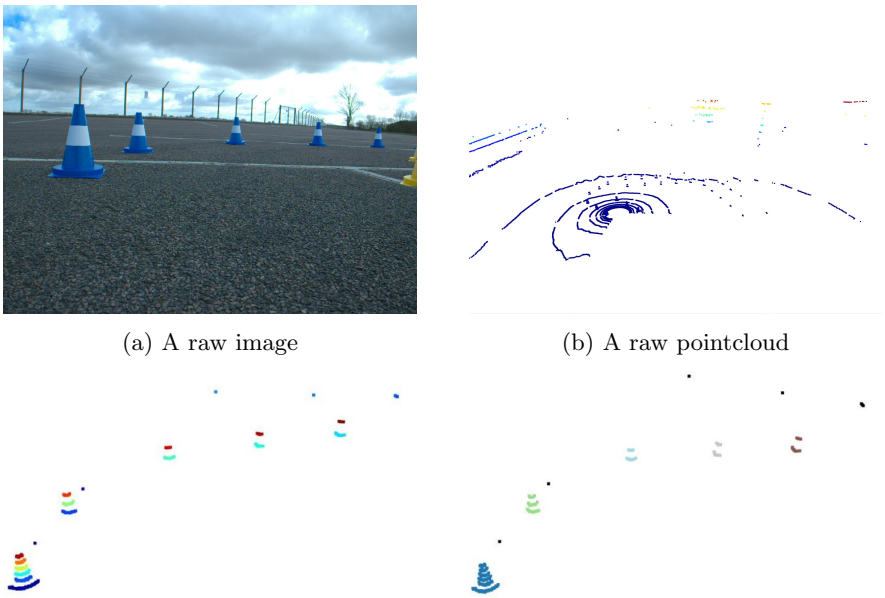
6.4 Cone Center Estimation

This subsection describes how the cone centers are estimated from a point cloud and a segmented mask. The cone center estimation method is comprised of two parts, as seen in fig. 12.

Each of the 3D points from the point cloud is projected to the image plane using eq. (34), and the label of the closest pixel in the mask is assigned to the 3D point. If the point is assigned as background, it is removed from the point cloud. The result is a point cloud, where each point contains a semantic label corresponding to one of the four classes.

To perform a successful center estimation for each cone, the cone each point belongs to needs to be known. This is also referred to as instance segmentation. To distinguish the cones from each other, a clustering algorithm was implemented. Since the distance between two cones is always larger than the

distance between points on the same cone, a Euclidean clustering algorithm was chosen and implemented in Open3D ([44]). Further details about the clustering algorithm can be found in [13]. This clustering method takes two parameters: the maximum Euclidean distance from a point to another in the cluster and the minimum number of points in a single cluster. Figure 18 shows the results of applying the point cloud filtering followed by clustering, where the points in fig. 18d are filtered with a maximum distance of 30cm and a minimum of 7 points in each cluster. Each cluster is assigned a color in that figure, while the black points do not belong to a cluster.



(a) A raw image

(b) A raw pointcloud

(c) Pointcloud filtered by masks

(d) Pointcloud after clustering

Figure 18: The process of point cloud filtering. From a segmented mask, the raw point cloud (fig. 18b) is filtered (fig. 18c), and the points clustered with Euclidean clustering (fig. 18d).

Since all LiDAR points not belonging to a cone are filtered, it is essential that the sensor fusion is as accurate as possible (eq. (34)). However, even if this fusion is perfect, some sources of error remain:

1. Synchronization error between the LiDAR and the camera
2. Poor definition of edges in the image
3. The cone is obstructed

The synchronization error between the camera and the LiDAR is apparent in fig. 19, where the vehicle is turning. Since the sampling rate is $10Hz$, the worst-case synchronization offset is $50ms$.

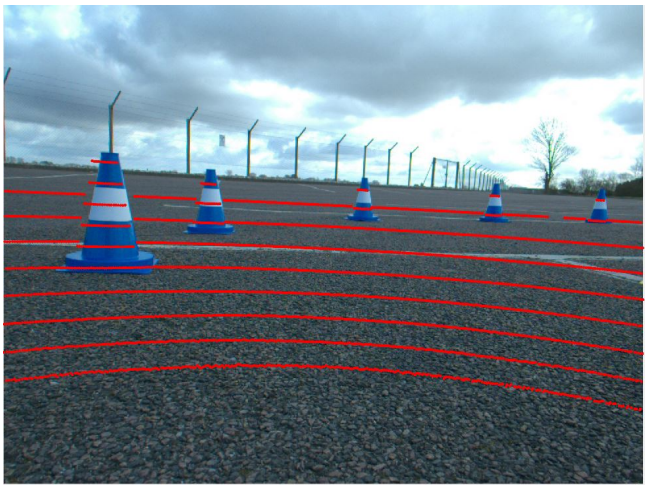


Figure 19: Although the vehicle is moving slowly, the offset due to poor synchronization between the LiDAR and camera can be clearly seen when rotating. This results in a slight bias of the observed points on the cone when projected to the image.

The upper scanline of each cluster was reconstructed to compensate for these errors. This requires at least two scanlines on a cone. A single scanline on a cone can indicate several things:

- The scanline is on the bottom of the cone
- The cone is too far away to be correctly observed
- The cone is only partially observed in the image

These reasons could cause the algorithm to fail, and so, if a cone had only one scanline, it was not included as an observation. The reconstruction was performed by iteratively adding previously removed points to the scanline if they fall within some threshold. The reconstruction is deemed necessary, as 25% of the points of the upper scanline on the cones were incorrectly removed. This could be explained by one of the error sources described above. This number was calculated by analyzing the results of the experiment explained in section 9.1. The distance threshold was initially 10cm but could easily work for lower values. This is simple and efficient when the LiDAR points are sorted by the vertical and horizontal angles. Since this is only done on the topmost scanline, no background points are close to these points.

To estimate the center of each cone, a vector, v_F was defined from the origin to furthest point on the top-most scanline. A unit vector is then defined, v_M, pointing from the origin to the midpoint of the line connecting the left-most and right-most points of the same scanline. v_M is then scaled by $|v_F|$. The end-point of v_M is then the estimated center of the cone. These vectors are

all defined in the (x, y) plane. The reason this method is deemed valid is that the diameter of the upper scanline of a cone is quite small, and the projected center will lie within the actual cone with a very high probability. One could argue that this proposed method is less sensitive to noise compared to regular circle-fitting methods when only a few points are observed.

7 SLAM Implementation

This section describes the choice of the SLAM method, as well as the specific representations and implementations that were used in this project for solving the SLAM problem. A figure of the overall system design, with the module described in this section marked in orange, can be seen in fig. 20.

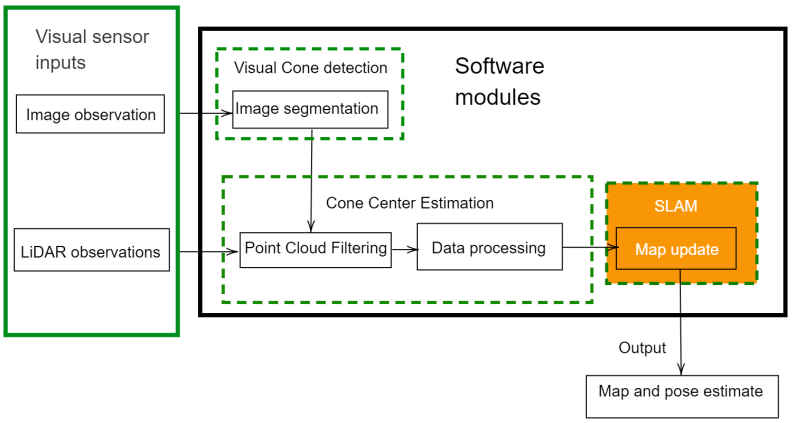


Figure 20: The overall system design, with the modules explained in this section highlighted in orange.

7.1 Choice of SLAM Method

The core goal of this project is to develop a SLAM method capable of navigating in a Formula Student trackdrive event. Several constraints are defined for the method:

- Easy to scale to the number of landmarks
- Adjustable to time-complexity
- The motion model should be easy to implement and extend

These constraints are derivatives of the constraints defined for the complete system in section 1.1.

If the motion model was not easy to implement and extend, it would not work for various interoceptive sensors. It would also not be possible to implement several combinations of these.

The number of cones on a trackdrive is not given in advance, and the number might be huge. For this reason, the SLAM system needs to be scalable in the number of landmarks.

Furthermore, the system should be adjustable to time complexity since it could be run on various microprocessors.

For these reasons, the FastSLAM family of methods was chosen. FastSLAM 2.0 was used since it does not constrain the user to an accurate motion model [31].

A downside of FastSLAM is that it is complicated to implement a loop-closing algorithm for optimization, similar to the one presented in [32]. However, previous systems (e.g., [20]) obtained precision at such a high level that loop-closing optimization was not necessary.

7.2 State Space Representation

To fully represent the state of the vehicle requires six dimensions (x, y, z, roll, pitch, and yaw). Having a 6D state-space representation for the vehicle would require a massive number of particles since there should be a decent variation along all dimensions. The required amount of particles grows exponentially when adding an extra dimension to the state space.

The Formula Student competition is held on a relatively flat surface. Therefore, not all degrees of freedom provide the same value to the system. While using all degrees of freedom would technically yield the highest accuracy, the trade-off in time complexity is most likely not worth it.

With that in mind, the state space of the vehicle is described with two translational coordinates (x, y) and an angle θ:

$$s_t^{[m]} = \begin{bmatrix} x\ [m] \\ y\ [m] \\ \theta\ [rad] \end{bmatrix} \tag{36}$$

The system is designed to take a single coordinate as input for the cone. Since the cone lies on the same plane as the vehicle, and the angle of the cone is not relevant, the absolute position of the cone is chosen to be represented with two coordinates (x, y):

$$\theta_n = \begin{bmatrix} x\ [m] \\ y\ [m] \end{bmatrix} \tag{37}$$

7.3 Motion Model and Observation Model design

In this thesis, a motion model is not implemented on the vehicle since the hardware needed has not yet been developed, and to develop it is considered outside the scope of the project. The idea is that the project should be open to

modification, as mentioned previously. This means that the project should not impose any constraints on the motion model, such as linearity. In theory, any motion model could be implemented as an extension to the system.

Since FastSLAM 2.0 requires a motion model, a simple odometry estimate is created using the cone center estimation. The transformation between two states is estimated using point-pair correspondences of the same cones in consecutive frames. Since the vehicle is driving slowly in the mapping phase, these correspondences can be found with a Euclidean distance threshold of $0.4m$. The transformation of the points from time t to $t+1$ can be described as:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \end{bmatrix} \tag{38}$$

This transformation was found by using the Total Least Squares method from [40].

A motion model completely independent of the observation model would be optimal. However, this motion model can very easily be replaced; this is one of the reasons FastSLAM was chosen in the first place.

The observation model takes the estimated cone centers as input. Since it is assumed that the race track is on a flat surface, the observation model can be described in 2D relative to the vehicle's state. A simple observation model, as the one suggested in [31], was designed to estimate the relative range and bearing of the landmark:

$$z_t = g(s_t, \theta_{n_t}) = \begin{bmatrix} r(s_t, \theta_{n_t}) \\ \phi(s_t, \theta_{n_t}) \end{bmatrix} = \begin{bmatrix} \sqrt{(\theta_{n_t,x} - s_{t,x})^2 + (\theta_{n_t,y} - s_{t,y})^2} \\ \arctan\left(\frac{\theta_{n_t,y} - s_{t,y}}{\theta_{n_t,x} - s_{t,x}}\right) - s_{t,\theta} \end{bmatrix} \tag{39}$$

7.4 Derivation of Jacobians

To calculate the Kalman Gain (eq. (21)) needed for updating the landmarks, the Jacobian G_θ was computed and can be seen in eq. (40).

$$G_{\theta_{n_t}} = \begin{bmatrix} \frac{\theta_{n_t,x} - s_{t,x}}{d} & \frac{\theta_{n_t,y} - s_{t,y}}{d} \\ -\frac{\theta_{n_t,y} - s_{t,y}}{d^2} & \frac{\theta_{n_t,x} - s_{t,x}}{d^2} \end{bmatrix} \tag{40}$$

where d is the Euclidean distance from the vehicle to the observed landmark.

In FastSLAM 2.0, the proposal distribution also takes the measurement model into account as described in section 3.1.2

The Jacobian G_s can be computed as:

$$G_s = \begin{bmatrix} -\frac{\theta_{n_t,x} - s_{t,x}}{d} & -\frac{\theta_{n_t,y} - s_{t,y}}{d} & 0 \\ \frac{\theta_{n_t,y} - s_{t,y}}{d^2} & -\frac{\theta_{n_t,x} - s_{t,x}}{d^2} & -1 \end{bmatrix} \tag{41}$$

7.5 Modifications to the FastSLAM 2.0 Algorithm

The FastSLAM 2.0 Algorithm was implemented using Python. In the following subsections, the modifications that were performed to the original algorithm as described in section 4.4 are explained.

7.5.1 Landmark observations

Multiple sensor measurements were made in each timestep, where the centers of the cones were estimated. Traditionally, eq. (29) could be used to decide whether or not to associate the current observation to an existing landmark. However, since the cones are always clearly separated in a Formula Student competition, a simple Euclidean distance threshold of 0.7m was used. If an observation is within that distance of an existing landmark, a data association should be assigned. Otherwise, a new landmark was created.

Along with an estimated (x, y) position of the landmarks, the number each classification is stored. The current class estimate is then updated in each iteration by choosing the class that has the most predictions.

7.5.2 Weight update and resampling

After the observation step, the weight for each particle was computed. In the background literature [31, 29, 30], it is assumed that a single observation is made per timestep. If several observations are made, the weight is simply the product of the individual probabilities for each of the observations [37]:

$$w_t^{[m]} = \prod_{i=1}^{N} \frac{1}{\sqrt{|2\pi Z_{n_i,t}|}} \exp\left(-\frac{1}{2}(z_{i,t} - \hat{z}_{n_{i,t},t})^T [Z_{n_{i,t},t}]^{-1} (z_{i,t} - \hat{z}_{n_{i,t},t})\right) \quad (42)$$

The resampling method chosen for this project is Low variance resampling described in section 4.3.1

7.6 Overview

This section shows the FastSLAM 2.0 algorithm used in this project. This is a modified version from the one described [31] and partly inspired by the implementation described in [37]. It follows the notations described in table 1, where a landmark class estimate, y is added to the particle set, which includes the amount of each classification for each landmark.

Algorithm 5 A single timestep in the FastSLAM 2.0 algorithm,

// Sampling step
for *m = 1 to M* **do**
  Retrieve particle m from the state S_{t-1}
    $\left\langle s_{t-1}^{[m]}, N_{t-1}^{[m]}, w_{t-1}^{[m]}, \mu_{1,t-1}^{[m]} \Sigma_{1,t-1}^{[m]}, y_{1,t-1}^{[m]}, .., \mu_{N_{t-1}^{[m]},t-1}^{[m]}, \Sigma_{N_{t-1}^{[m]},t-1}^{[m]}, y_{N_{t-1}^{[m]},t-1}^{[m]} \right\rangle$
  $w_t^{[m]} = \left(N_{t-1}^{[m]}\right)^{-1}$
  $s_t^{[m]} \sim p(s_t \mid u_t, s_{t-1}^{[m]})$
end
// Loop through all the observations
for *i=1 to K* **do**
  // retrieve observation z_i and classification y_i
  // Loop through all the particles
  for *m = 1 to M* **do**
    Retrieve particle m from the state S_{t-1}
    $\left\langle s_{t-1}^{[m]}, N_{t-1}^{[m]}, w_{t-1}^{[m]}, \mu_{1,t-1}^{[m]} \Sigma_{1,t-1}^{[m]}, y_{1,t-1}^{[m]}, .., \mu_{N_{t-1}^{[m]},t-1}^{[m]}, \Sigma_{N_{t-1}^{[m]},t-1}^{[m]}, y_{N_{t-1}^{[m]},t-1}^{[m]} \right\rangle$
    // The absolute position of the observation z_i
    $a = g^{-1}(s_t^{[m]}, z_i)$
    // Loop over all possible data associations
    for *n=1 to N* **do**
      // Distance from the landmark to the possible data association
      $d_n = \sqrt{(\theta_{t,n,x} - a_x)^2 + (\theta_{t,n,y} - a_y)^2}$
    end
    $d_{min} = min(d)$
    $n = argmin(d)$
  end
  if $d_{min} < 0.7$ **then**
    updateLandmark(m, z_i, y_i, n) // algorithm 6
    proposalSampling(m, n, z_i) // algorithm 8
    // Update weight
    $w^{[m]} = w^{[m]} \cdot \mid 2\pi Z_{n,t}^{[i]} \mid^{-\frac{1}{2}} exp\big[-\frac{1}{2}(z_t - \hat{z}_{n,t})^T Z_{n,t}^{-1}(z_t - \hat{z}_{n,t})\big]$
  else
    addNewLandmark(m, z_i, y_i) // algorithm 7
  end
end
Normalize the weights of all particles and resample using algorithm 4

Algorithm 6 updateLandmark(m, z, y, n)

Input: particle index m, observation z, classification y, data association n

$G_{\theta_{n_t}} = \Delta_{\theta_n} g(\theta_n, s_t) \mid_{s_t = s_t^{[m]}; \theta_{n_t} = \mu_{n_t,t-1}^{[m]}}$

$\hat{z}_t = g(s_t^{[m]}, \mu_{n_t,t-1}^{[m]})$

$Z_{n,t} = R_t + G_{\theta_{n_t}} \Sigma_{n_t,t-1}^{[m]} G_{\theta_{n_t}}^T$

$K_t = \Sigma_{n_t,t-1}^{[m]} G_{\theta_{n_t}}^T Z_{n,t}^{-1}$

$\mu_{n_t.t}^{[m]} = \mu_{n_t.t-1}^{[m]} + K_t(z_t - \hat{z}_t)$

$\Sigma_{n_t,t}^{[m]} = (I - K_t G_{\theta_{n_t}}) \Sigma_{n_t,t-1}^{[m]}$

$y_{n,t}^{[m]} = y$

Algorithm 7 addNewLandmark(m, z, y)

Input: particle index m, observation z, classification y

$N_t^{[m]} = N_{t-1}^{[m]} + 1$

$n_t = N_t^{[m]}$

$G_{\theta_{n_t},t} = \Delta_{\theta_n} g(\theta_n, s_t) \mid_{s_t = s_t^{[m]}; \theta_{n_t} = \mu_{n_t,t-1}^{[m]}}$

$\mu_{n_t,t}^{[m]} = g^{-1}(s_t^{[m]}, z_t)$

$\Sigma_{n_t,t}^{[m]} = \left(G_{\theta_{n_t},t}^T R^{-1} G_{\theta_{n_t},t} \right)^{-1}$

Algorithm 8 proposalSampling(m, n, z)

Input: particle index m, data association n, observation z

$G_{\theta_{n_t}} = \Delta_{\theta_n} g(\theta_n, s_t) \mid_{s_t = s_t^{[m]}; \theta_{n_t} = \mu_{n_t,t-1}^{[m]}}$

$Z_{n,t} = R_t + G_{\theta_{n_t}} \Sigma_{n,t-1}^{[m]} G_{\theta_{n_t}}^T$

$G_s = \nabla_{s_t} g(s_t, \theta_{n_t}) \mid_{s_t = s_t^{[m]}; \theta_n = \mu_{n,t-1}^{[m]}}$

$\Sigma_{s_t}^{[m]} = \left[G_s^T Z_{n,t}^{-1} G_s + P_t^{-1} \right]^{-1}$

$\mu_{s_t}^{[m]} = \Sigma_{s_t}^{[m]} G_s^T Z_{n,t}^{-1} (z_t - \hat{z}_t) + s_t^{[m]}$

8 Preliminary Experiments

This section describes two different experiments that were carried out prior to the final experiment. Section 8.1 describes the experiment that was performed to determine the accuracy and runtime of the various neural networks used in this project, while section 8.2 describes the experiment with the goal to determine the uncertainty of the measurement model.

8.1 Evaluation of the Neural Networks

In this experiment, different segmentation networks were analyzed, where both runtime and accuracy were assessed. The experiment was performed using PyTorch in Python, using the data described in section 6.2. This comparison between runtime and accuracy should aid future implementation since the most significant limitation is hardware and resources. The implementation should be the fastest one that has satisfactory accuracy. The networks described in section 6 were used in this experiment. The type of data augmentation chosen for each model was the augmentation type that yielded the highest mIoU for the test data.

When choosing a model, the gap between the training loss and validation loss, sometimes referred to as the validation gap, was assessed. A common indication of over-fitting is when the training loss decreases after the validation loss has converged, or worse, starts to increase again. After choosing the model with that in mind, mIoU was calculated on the never seen before test set.

8.1.1 Results and Part Conclusion

The results of the different types of models can be seen in table 2

Net	**Augmentation**	**Parameters**	**mIoU**	**Time** [s]
Deeplab V3	None	60.992.090	91.02	0.88
UNet	None	17.267.653	90.79	0.47
UNet*	Full	4.284.613	83.97	0.39
UNet**	Crop, rotate and affine	1.037.125	56.70	0.31

Table 2: Comparison of the neural networks tested in this project. *: 7 double convolutional layers. **: 5 double convolutional layers

Given the limited variation of the training data, which was collected manually, it is difficult to say whether different augmentation methods would give better results on other data. Although detecting cones sounds like a simple classification problem, this experiment indicates that a neural network with a rather large number of parameters is needed to solve the task accurately.

When analyzing the runtime of the different networks, the average inference time was measured when predicting an RGB image with the size of 1456×1088. The test was performed using an Nvidia GeForce RTX2060 6GB GPU.

8.2 Uncertainty of the Measurement Model

This section describes the experiment that was carried out to determine the uncertainty of the measurement model. The goal is to estimate a covariance matrix that represents the uncertainties as required in the FastSLAM algorithms:

$$R = \begin{bmatrix} \sigma_r^2 & \sigma_{r\theta} \\ \sigma_{\theta r} & \sigma_\theta^2 \end{bmatrix} \tag{43}$$

Where r indicates the observed range and θ indicates the angle in radians.

Many factors can affect the uncertainty of a measurement model. In this case, it mainly depends on the following factors:

1. The accuracy of the VLP-16 LiDAR sensor
2. The classification accuracy of the segmentation algorithm (section 6.3)
3. Imperfect transformation between the LiDAR and the image, either due to the sensor fusion (section 6.1) or bad synchronization
4. The accuracy of the cone center estimation algorithm (section 6.4)
5. The track is uneven

After filtering the point cloud by using the segmented masks, the cones are reconstructed (section 6.4). As long as a single point of the top scanline will be included in the segmented mask, the reconstruction will succeed. In this case, factors 2 and 3 can safely be ignored. The most important factors remain, which are the uncertainty of the LiDAR and the cone center estimation method.

There is no information in the VLP-16 datasheet that may be used to assess the uncertainty analytically, and to our knowledge, no information is available from the manufacturer. This uncertainty is most likely complicated to model since it may depend on many factors (e.g., lighting condition, material color and texture, distance, temperature, and humidity). For this reason, an experiment was conducted in order to determine this uncertainty.

8.2.1 Experimental Setup

The experiment was performed in Odense, Denmark, on a cloudy day (2. May 2021 09:30) at 6 degrees.

Multiple measurements were made of different types of cones with varying ranges to assess the uncertainty of the measurement model. Since nothing indicates that the uncertainty depends on the angle of the measurement, where the LiDAR makes measurements with a 360° field of view, the cone measurements were done in a straight line with the same reference angle. The measurements were made with a 0.5 meter interval. The ground truth was measured using a measuring tape, and the reference angle was estimated from a LiDAR measurement of a cone on the same line.

The images were segmented with the DeepLabv3 model described in section 6.3, and the cone centers were estimated as described in section 6.4.

8.2.2 Results and Part Conclusion

The estimated cone positions, along with the ground truth estimates can be seen in fig. 21.

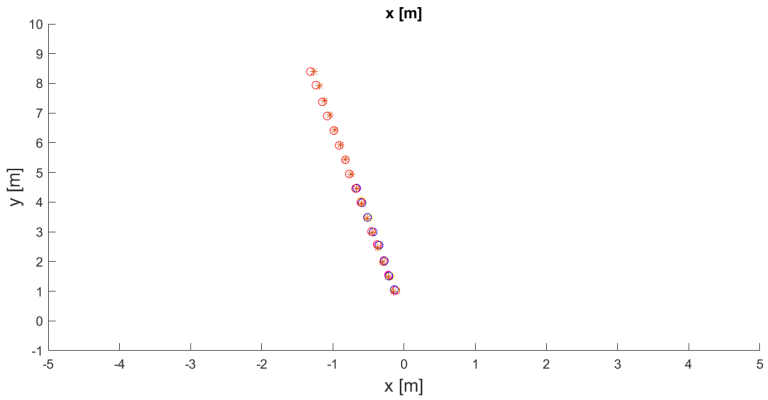
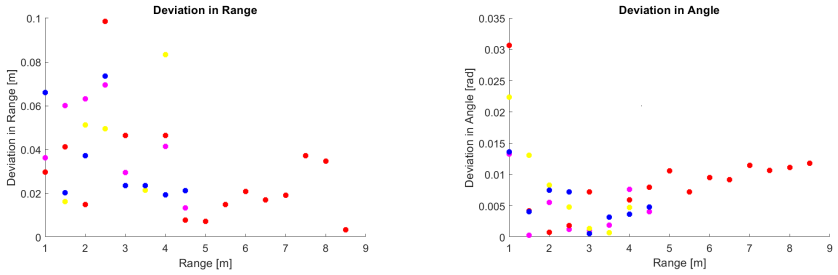


Figure 21: The cones were placed in a straight line with 0.5m interval. The circles indicate the estimated cone centers, and the stars indicate the ground truths.

In fig. 22 it can be seen that there is no immediate correlation between the range of the measurement and the error in range and bearing.

The deviation in range and angle was analyzed for different ranges to determine whether the range affects the accuracy of the LiDAR measurements (fig. 22).

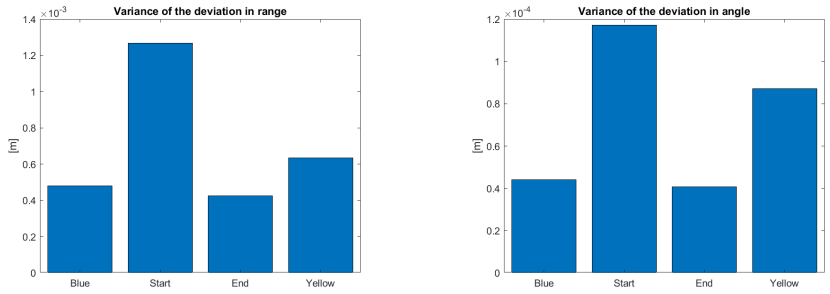


(a) (b)

Figure 22: The deviation of the measurements taken at different ranges. Figure 22a shows the deviation in the range, while fig. 22b shows the deviation in the angle.

The range does not seem to be a significant factor in the uncertainty. A constant covariance matrix for the cones is therefore sufficient.

The cone class could impact the accuracy of the LiDAR points since they all have different colors and heights. Figure 23 shows the individual deviation of each class, both in range and angle.



(a) (b)

Figure 23: The variance of the deviation of the measurements taken at different ranges, for each of the cones. Figure 23a shows the variance in the range, while fig. 23b shows the variance in the angle.

The uncertainty of the deviation in the range is similar for each of the small cones (blue, end, and yellow), while the start cone could fit into its own group.

The deviations do not seem to imply any logical grouping of the cones. The deviations are dissimilar; therefore, separate covariance matrices are calculated and used for each cone.

$$R_{blue} = \begin{bmatrix} 0.4812 & 0.1162 \\ 0.1162 & 0.0440 \end{bmatrix} \cdot 10^{-3}$$

$$R_{start} = \begin{bmatrix} 1.3 & 0.134 \\ 0.134 & 0.117 \end{bmatrix} \cdot 10^{-3}$$

$$R_{end} = \begin{bmatrix} 0.4242 & 0.0365 \\ 0.0365 & 0.0408 \end{bmatrix} \cdot 10^{-3}$$

$$R_{yellow} = \begin{bmatrix} 0.6358 & 0.0031 \\ 0.0031 & 0.087 \end{bmatrix} \cdot 10^{-3}$$

9 Evaluation of the System

This section describes the experiments that were carried out in order to evaluate the implementation of the full pipeline of this project.

For these experiments, a single non-trivial track was designed and constructed. Following the construction, the track was navigated manually with the vehicle using a human-held remote controller. The data was recorded from the two sensors during this test drive.

The experiment is divided into different parts, depending on the overall goal of the specific test. They all share the experimental setup, as well as the data collected from the experiment.

This section is divided into several subsections. In section 9.1, the experimental setup used for collecting the data is described.

In section 9.2 the experiment that uncovers the rate of misclassification of the cones is presented.

In section 9.3, the experiment and results of the test that was performed to evaluate the performance of the algorithm when using a different number of particles is presented.

In section 9.4 the experiment in which the odometry estimates of the car were disabled is demonstrated.

In section 9.5 the effect of an increased Field of View of the camera is demonstrated.

Finally, section 9.6 measures the runtime of the complete system.

To assess the performance of the SLAM algorithm, RMSE was used for the best particle, m, in the particle set where N is the total number of observed landmarks in particle m:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} \left(\theta_x - s_{t,\theta_i,x}^{[m]}\right)^2 + \left(\theta_y - s_{t,\theta_i,y}^{[m]}\right)^2}{N}} \tag{44}$$

9.1 Experimental Setup

The experiment was performed in the parking lot at Beldringe Airport, Denmark, on a cloudy day (7. April 2021 09:30) 5°C.

A track was constructed to assess the validity of the approach. There were several factors that the track must live up to:

- At least one full turn
- The track must end where it started (loop closure)
- The track should include at least one left and one right turn

These requirements were set up to more thoroughly cover the situations most likely to be encountered in an actual driving situation. The track was constructed by manually designing a grid on the ground by triangulating the distance to two previously measured cones. The measurements were performed using a measuring tape. An image of the track can be seen in fig. 24.

Figure 24: Image of the setup used in the final experiment. The experiment was conducted in Beldringe Airport.

The experiment was conducted using the hardware outlined in section 5.3, with no further modifications. The data was collected using VeloView and Basler Video Recording Software. The data was synchronized manually by placing an object simultaneously in front of both sensors.

The motion noise used for this experiment was:

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_\phi \end{bmatrix} = \begin{bmatrix} 0.057 \\ 0.057 \\ 0.006 \end{bmatrix} \tag{45}$$

Using the Least Squares method described in section 3.1.2 to simulate a motion model failed in certain frames in this experiment due to lack of correspondences between frames while turning. This occurred due to the low FOV of the camera. Correspondences were added manually to frames that would otherwise fail, in order to provide an estimate of the odometry model. These extra correspondences were not used in the SLAM algorithm.

In all experiments, the DeepLabv3 model as described in section 4.5.3 was used for the segmentation. The training of the model was performed as detailed in section 6.3. The code can be seen in [1].

In all applications of the SLAM algorithm, the algorithm was implemented in Python. The code can be seen in [1].

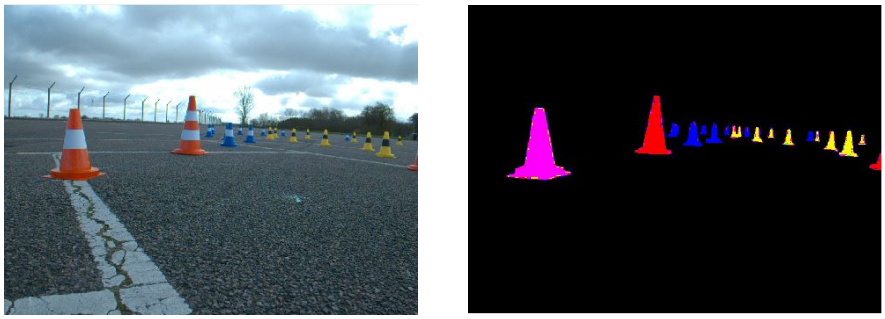
9.2 Verification of Cone Classification

As stated previously, correctly classifying the cones is a crucial step in the Formula Student competition. Misclassification may result in the path planning algorithm failing. Many factors could lead to misclassification of a cone in this approach, e.g., misclassification in the segmentation and poor synchronization between the LiDAR and the camera. The latter would erroneously remove points from the point cloud or provide them with an incorrect label.

The goal of this experiment is to determine the degree of correct cone classifications in the experimental setup.

9.2.1 Results and Part conclusion

The first step in the cone detection algorithm is to locate the cones in the image. An example of an image, along with the segmented mask can be seen in fig. 25.



(a) (b)

Figure 25: A raw image from the final test (fig. 25a), along with the segmented image (fig. 25b) visualizing the predicted classes. The segmented image was predicted using the DeepLabv3 network described in section 6.3.

The results from this experiment can be seen in fig. 26. The classification accuracy below $2m$ is 100%, and decreases to 84.6% after $7m$. Although some temporary misclassification occurs, the final result is always correct classification since every cone that is observed at least once is observed in multiple frames.

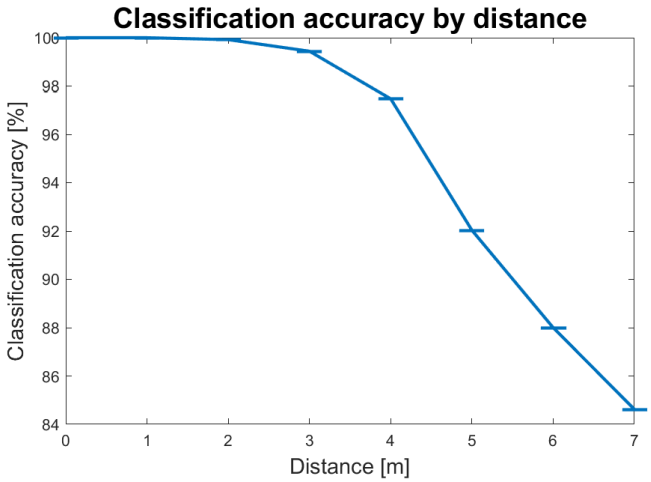


Figure 26: The accuracy of the cone classification from all the measurements in section 9.1.

9.3 Number of Particles

An essential parameter in the FastSLAM algorithms is the number of particles. The value of this parameter depends on the characteristics of the specific application. Two factors have a significant effect on the choice of the parameter; the uncertainty of the motion model and the degrees of freedom of the vehicle. As either of these increases, more particles are needed to achieve the same accuracy.

The FastSLAM algorithms' time complexities grow linearly with the number of particles (section 4.4.7). In theory, the algorithm's accuracy will never decrease, on average, with an increased number of particles. Like in many other cases, there is a trade-off between runtime and accuracy. The number of particles can be reduced to decrease the runtime but at the risk of a decrease in accuracy.

The aim of this experiment is to determine the effect of the number of particles on the RMSE.

9.3.1 Results and Part Conclusion

The RMSE with varying amounts of particles can be seen in fig. 27.

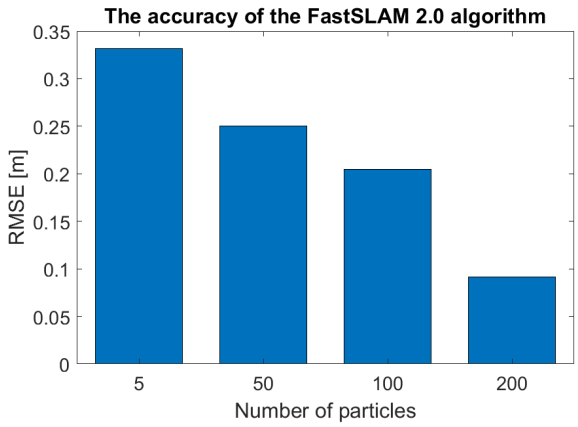
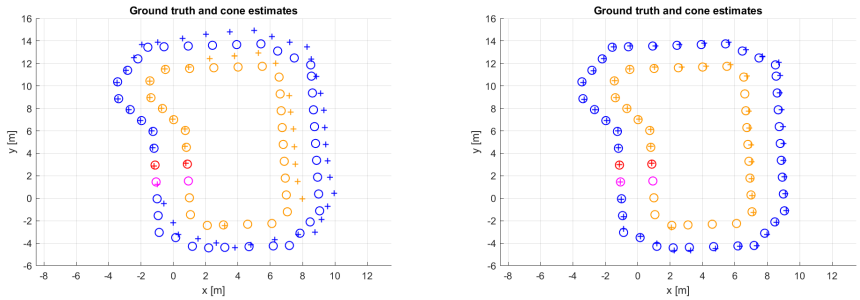


Figure 27: RMSE as a function of number of particles. The RMSE drops when more particles are included.

The estimated map of the best particle, $\Theta^{[m]}$, along with the ground truth map, Θ can be seen in fig. 28.



(a) 5 particles

(b) 200 particles

Figure 28: The estimated map for the experiments performed with 5 and 200 particles. The crosses indicate estimated cones and the circles are the ground truth.

The results clearly show that the RMSE decreases monotonically with the number of particles. The best result was an RMSE of $0.0913m$ using 200 particles. The decrease in RMSE means that the parameter can be tuned, depending on whether there is a desired increase in accuracy or a desired decrease in runtime.

Note that seven cones are missed. This is due to the limited FOV of the camera and not due to errors in the system.

9.4 Performance with no odometry estimate

Like stated previously, filter-based SLAM methods, such as FastSLAM 2.0, consist of two steps: a prediction step and an update step. The prediction step gives a rough estimate of the current state of the robot; this is given by the odometry model, while the correction step is performed using the measurement model. As explained in section 7.3, a motion model was not developed for the vehicle. The method in this project instead estimates the odometry using the measurements.

In this experiment, the algorithm was run where the estimated state at time $t = i$ was equal to the state at time $t = i - 1$ to evaluate the algorithm without an odometry estimate. Since FastSLAM is a particle filter-based algorithm, this is predicted to work as long as the step size is not too large relative to the variance of the proposal distribution.

Running the FastSLAM algorithm in practice without a motion model is not optimal since it requires more particles, which in turn requires more computational power. It also prevents the possibility for the system to run on pure odometry estimates given uncertain sensor inputs. However, testing this will indicate if the solution will work with other motion models since any motion model should perform better than none.

9.4.1 Results and Part Conclusion

Given enough particles, a decent map estimate can be obtained as seen in fig. 29 with an accuracy of $RMSE = 0.152m$.

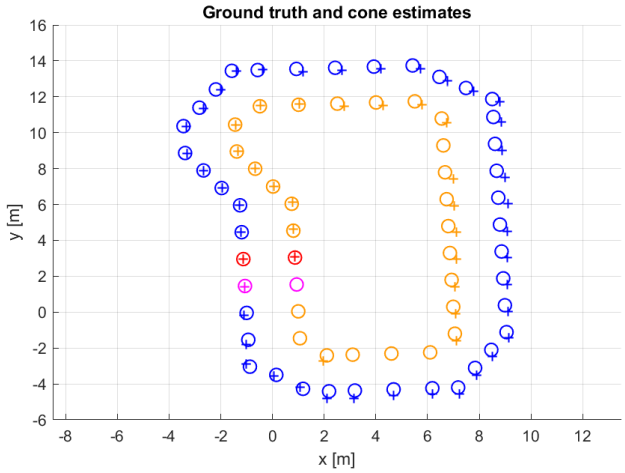


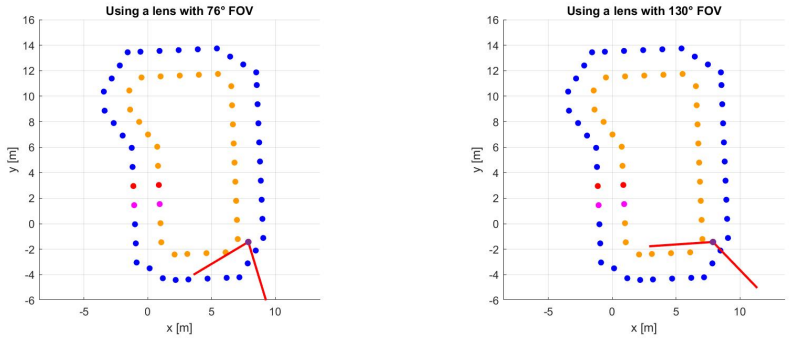
Figure 29: Using 200 particles, a decent map estimate was obtained with no odometry estimate as an input ($RMSE = 0.152m$).

The noise in the sampling step needs to be increased if the motion model is inaccurate or the step size is large. However, the results demonstrate that

even without a motion model, the algorithm provides a reasonable estimate of the map. The RMSE roughly corresponds to the same test performed with odometry, with twice as many particles.

9.5 Improvements using increased FOV

The camera lens used in this project was a C125-0418-5M-P F4mm, with a 76° horizontal FOV. As seen in fig. 28b, 7 of the 68 cones fail to be observed. The reason is the small FOV of the camera lens. This can be confirmed by analyzing the images taken in the test. Due to time constraints, it was not possible to repeat these experiments with another lens. However, by visualizing the path driven by the robot, along with the ground truth estimates, it can be seen that another lens with a larger FOV, e.g., "M13B02118IR F1.8" with a 130° horizontal FOV, would fix this problem. This can be seen in fig. 30.



(a) (b)

Figure 30: Some of the cones failed to be observed due to the small horizontal FOV of the camera lens. This could be fixed by choosing another camera lens with a larger horizontal FOV. Figure 30a shows the FOV of the camera lens used while fig. 30b shows the FOV of the proposed camera lens.

9.6 Runtime Feasibility Analysis

As stated previously, implementing the system in real-time is not the focus of this project. However, it should be feasible to optimize it to the point that it could run in real-time. This experiment was performed to indicate whether it is plausible to assume that the system could be optimized to run in real-time.

Each software module described in fig. 9 was run independently for all the data in this experiment. The average runtime for each module was then calculated for a single time step.

9.6.1 Results and Part Conclusion

The runtime analysis of the overall system can be seen in table 3.

Sub-system	**Runtime** [s]
Segmentation	0.88
Cone center estimation	0.65
FastSLAM 2.0 iteration	0.42
Total	1.95

Table 3: The average runtime for a single time-step for all the sub-systems in the pipeline.

Currently, the system is not able to run in real-time applications using the current sensor setup. The sensors both sample at 10Hz, which means that the speed needs to be improved by 1950% to be run as-is.

A vital factor to note is that the parameters of the networks used in these tests were not optimized. Furthermore, sampling at a lower rate could be feasible, reducing the required increase in speed.

In fig. 31 it is shown that the runtime increases approximately linearly with an increased number of landmarks. This can be optimized so that the runtime increases logarithmically (see section 4.4.7), which would ensure a more stable runtime when driving longer tracks.

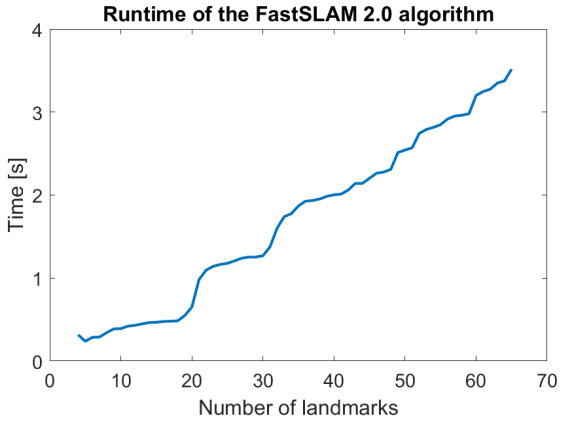


Figure 31: The FastSLAM 2.0 algorithm run with 200 particles. The runtime for a single iteration is analyzed using a different number of landmarks.

10 Discussion and Conclusion

The original goal was to integrate state-of-the-art techniques within Visual Odometry, SLAM, and Object Detection to generate an accurate Formula Student trackdrive course map. A system was developed with the aim of solving this problem. This system received images from a camera and point clouds from a LiDAR as input. The original approach of using Object Detection was modified to using Semantic Segmentation for the localization of the cones in the image to simplify the overall pipeline.

The approach proposed in this thesis has several upsides, one of them being the simplicity of the data processing step. The system only needs to perform the detection and classification of cones in the collected images. The primary alternative approach is to perform the detection and classification directly in the point cloud obtained from the LiDAR. The second upside is that the approach works even for quite sparde point clouds, as the needed information about the cones' location is provided by the image.

One downside of the approach is the incredibly high mIoU requirements for the segmentation algorithm, as false positives of cones can prove disastrous for the system. These false positives could result in ground points incorrectly being classified as cones. If ground removal was implemented, this would pose a much smaller problem. The same outcome would result from an error in synchronization between the LiDAR and the camera since the projected LiDAR points could be projected with a significant offset. The system is intended to be run on low velocities, and therefore this does not impose problems.

The current system is not able to process the data at a rate that would enable real-time operation. The system could be optimized; in the following paragraphs, the optimizations believed to cause the most significant improvements in runtime are presented.

The results from fig. 26 indicate that the image resolution could be decreased significantly without affecting the accuracy since the image resolution is considered high relative to the maximum distance of observable cones. This trade-off could be investigated, but is not the focus of this thesis. Different segmentation algorithms provide different results, and as stated previously, correct labels are crucial to the method to map the course successfully.

As seen in section 8.1, reducing the network size does not improve the inference time significantly compared to the decrease in mIoU when comparing DeepLabv3 and the reduced UNet architectures. A reduction of the parameters by 98.3% reduces the inference time by 61.7%, while the mIoU decreases by 37.7%. It isn't easy to compare the models directly since the reduced UNet was not pre-trained, while DeepLabv3 was.

The switch from Python to a more low-level programming language, such as C++, could also present a significant improvement in runtime. In the actual competition the code would need to be implemented on a microprocessor. Depending on the specific microprocessor, the programming language switch might be required for compatibility reasons.

The current implementation is simple and quite naïve. Modern processors

typically support multi-threading. A prominent place to implement multi-threading is the image segmentation step, where a single thread could communicate with an external GPU, which would then process the image. Another possible optimization using threads could be in the particle filter, where the particles could be divided over several threads. This would effectively reduce the runtime by a factor of the number of threads, with a slight deviation.

The final improvement would come from storing the landmarks in a tree structure as explained in section 4.4.7. This would prevent the runtime from increasing linearly with landmarks, enabling a more stable runtime driving a larger map.

A simplified motion model was implemented that used the cone center estimates to determine the movement of the vehicle. It was demonstrated that even without any information about the vehicle's movement, the system could predict the actual path with reasonable accuracy.

Although the tests indicate good results without having a motion model, having an accurate motion model is essential to the final product, mainly for two reasons. First of all, when having no motion model, the system requires more particles, and the sampling needs to be performed very frequently, which would result in the vehicle driving so slowly in the mapping round that it would significantly affect the score. Second, having no motion model requires the system to be very stable, not allowing the measurement model to fail temporarily, e.g., due to lack of cone observations. Having the possibility to run purely on the motion model temporarily in these cases can be beneficial.

The RMSE of the approach can be adjusted by adjusting the number of particles, effectively allowing the user to trade accuracy for runtime. The RMSE when using 200 particles was $0.0913m$, which should prove sufficient for the task of performing the path planning step of the competition correctly. However, this is difficult to determine without having implemented the path planning and ideally running the whole pipeline as seen in fig. 11 in an environment resembling the Formula Student trackdrive event.

10.1 Future work

This section presents our recommendations for any team who wishes to adopt the work we have performed and apply it to a Driverless Vehicle.

The primary recommendation is to implement the software system onto an embedded platform, including a microcontroller and GPU. The software modules described in fig. 9 all work independently and would need to be implemented in an embedded environment such as ROS. This embedding would also require synchronization between the camera and the LiDAR.

Additional improvements may be made to the system by adding interoceptive sensors, such as an IMU, that could be used to estimate the odometry. This would require designing a motion model for the particular sensor setup, allowing for a more stable and efficient system. Implementing a stable motion model would also enable a significant decrease in the number of particles, further decreasing the runtime.

Since runtime optimization has not been the focus of this project, there are several possibilities for improvements. Making the switch from an interpreted programming language (Python) to a compiled programming language (C++ or Rust) would present a significant increase in speed. The implementation itself also presents several opportunities for optimization, such as the tree structure implementation presented in section 4.4.7 and multi-threading.

Finally, the field of Deep Learning-based semantic segmentation of cones is an area that has not yet been thoroughly investigated. Arguably, further improvements to both the results and the number of parameters, and in turn, the runtime, could be significantly improved given the correct architecture. Four different ones were explored in this project, and users are encouraged to explore possibilities for increases in accuracy and speed.

References

[1] MSc_Fstudent_SLAM GitHub repository. `https://github.com/stebbibg/MSc_Fstudent_SLAM`. Accessed: 2021-05-29.

[2] SDU Vikings Racing Team. `https://sdu-vikings.dk`. Accessed: 2021-05-29.

[3] Traxxas x-maxx rc car. `"https://traxxas.com/products/landing/x-maxx/"`. Accessed: May 23 2021.

[4] Guillaume Bresson, Zayed Alsayed, Li Yu, and Sébastien Glaser. Simultaneous localization and mapping: A survey of current trends in autonomous driving. *IEEE Transactions on Intelligent Vehicles*, 2(3):194–220, 2017.

[5] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs, 2016.

[6] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs, 2017.

[7] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. `https://arxiv.org/abs/1706.05587`, 2017.

[8] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation, 2018.

[9] Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun. *Principles of Robot Motion*. The MIT Press, 2005.

[10] Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. Rao-blackwellised particle filtering for dynamic bayesian networks, 2013.

[11] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics Automation Magazine*, 13(2):99–110, 2006.

[12] Austin Eliazar and Ronald Parr. DP-SLAM: Fast, Robust Simultaneous Localization and Mapping without Predetermined Landmarks. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, page 1135–1142, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[14] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[15] Formula Student Germany. Formula Student Germany Rules 2021, Version 1. `https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf`. Accessed: May 13 2021.

[16] Formula Student Germany. Formula Student Germany website. `https://www.formulastudent.de/fsg/`. Accessed: May 13 2021.

[17] Formula Student Germany. *FSG Competition Handbook 2021.* Accessed: May 06 2021.

[18] Formula Student Germany. FSG: Results FSG 2019. `https://www.formulastudent.de/fsg/results/2019/`. Accessed: May 20 2021.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[20] N. Gosala, A. Bühler, M. Prajapat, C. Ehmke, M. Gupta, R. Sivanesan, A. Gawel, M. Pfeiffer, M. Bürki, I. Sa, R. Dubé, and R. Siegwart. Redundant perception and state estimation for reliable autonomous racing. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6561–6567, 2019.

[21] Fredrik Gustafsson. Particle filter theory and practice with positioning applications. *IEEE Aerospace and Electronic Systems Magazine*, 25(7):53–82, 2010.

[22] Fredrik Gustafsson. *Particle Filters*, pages 1037–1044. Springer London, London, 2015.

[23] Mojtaba Karimi, Martin Oelsch, Oliver Stengel, Edwin Babaians, and Eckehard Steinbach. Lola-slam: Low-latency lidar slam using continuous scan slicing. *IEEE Robotics and Automation Letters*, 6(2):2248–2255, 2021.

[24] Kruno Lenac, Andrej Kitanov, Robert Cupec, and Ivan Petrović. Fast planar surface 3d slam using lidar. *Robotics and Autonomous Systems*, 92:197–220, 2017.

[25] Xiaogang Li, Tiantian Pang, Biao Xiong, Weixiang Liu, Ping Liang, and Tianfu Wang. Convolutional neural networks based transfer learning for diabetic retinopathy fundus image classification. In *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, pages 1–11, 2017.

[26] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[27] Pauline Luc, Camille Couprie, Soumith Chintala, and Jakob Verbeek. Semantic segmentation using adversarial networks, 2016.

[28] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.

[29] Michael Montemerlo, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem. 11 2002.

[30] Michael Montemerlo, Daphne Koller, and Ben Wegbreit. FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges. *Proc. IJCAI Int. Joint Conf. Artif. Intell.*, 06 2003.

[31] Michael Montemerlo and Sebastian Thrun. *FastSLAM: A Scalable Method for the Simultaneous Localization and Mapping Problem in Robotics*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[32] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[33] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation, 2015.

[34] Institution of Mechanical Engineers. Formula Student - Institution of Mechanical Engineers. `https://www.imeche.org/events/formula-student`. Accessed: May 13 2021.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[37] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, and Alexis Paques. Pythonrobotics: a python code collection of robotics algorithms, 2018.

[38] Yun Su, Ting Wang, Shiliang Shao, Chen Yao, and Zhidong Wang. GR-LOAM: LiDAR-based sensor fusion SLAM for ground robots on complex terrain. *Robotics and Autonomous Systems*, 140:103759, 2021.

[39] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, Cambridge, Mass., 2005.

[40] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[41] Francesco Visin, Marco Ciccone, Adriana Romero, Kyle Kastner, Kyunghyun Cho, Yoshua Bengio, Matteo Matteucci, and Aaron Courville. Reseg: A recurrent neural network-based model for semantic segmentation, 2016.

[42] Ji Zhang and Sanjiv Singh. LOAM : Lidar Odometry and Mapping in real-time. *Robotics: Science and Systems Conference (RSS)*, pages 109–111, 01 2014.

[43] Haoyu Zhou, Zheng Yao, and Mingquan Lu. Lidar/UWB Fusion Based SLAM With Anti-Degeneration Capability. *IEEE Transactions on Vehicular Technology*, 70(1):820–830, 2021.

[44] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

Appendices

A U-Net diagrams

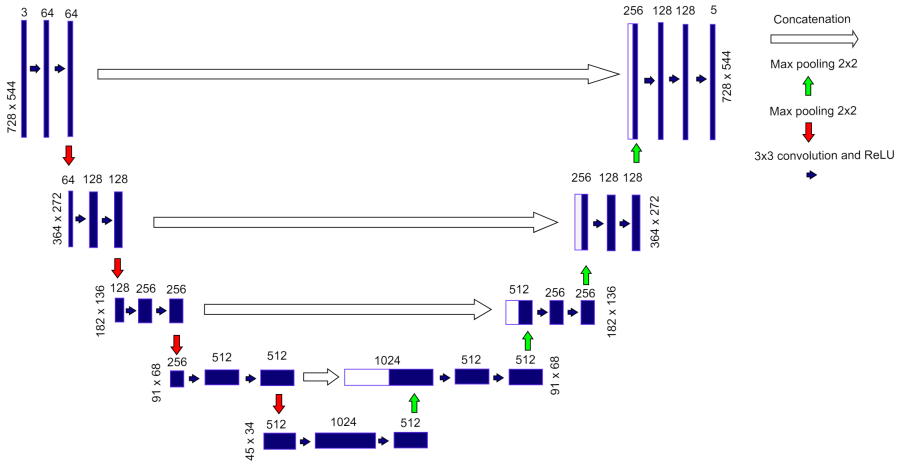


Figure 32: The UNet achitecture used in this experiments as described in [36]

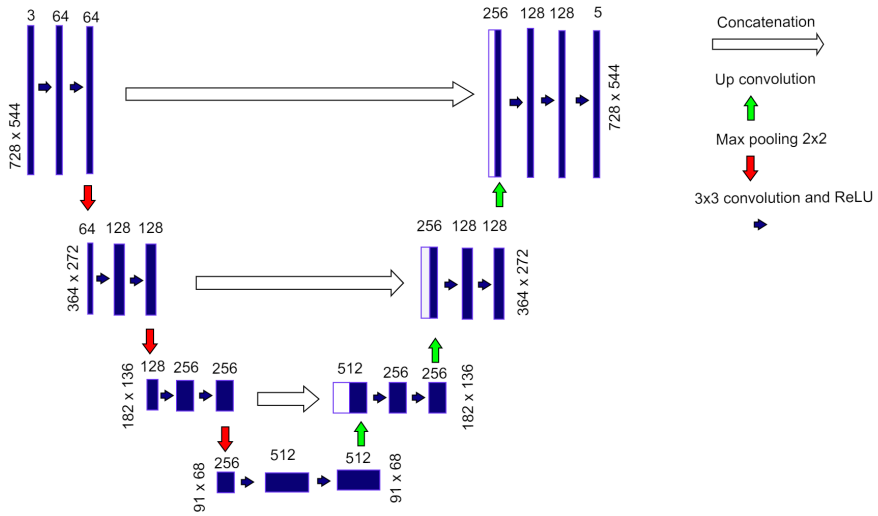


Figure 33: The custom U-Net architecture with seven double convolution layers

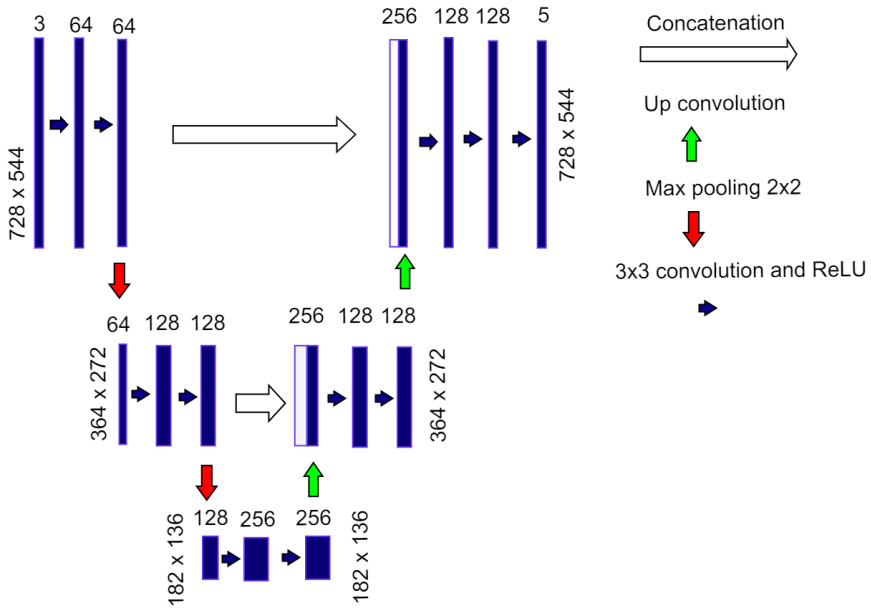


Figure 34: The custom U-Net architecture with five double convolutional layers

B Camera parameters

The intrinsic camera matrix was found:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1166.8 & 0 & 746.9 \\ 0 & 1170.9 & 586.7 \\ 0 & 0 & 1 \end{bmatrix}$$

Two radial distortion coefficients were estimated:

$$\begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} -0.237 \\ 0.091 \end{bmatrix}$$

along with two tangential distortion coefficients:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} 0.002 \\ -0.0007 \end{bmatrix}$$

The homogenous transformation between the LiDAR and the camera:

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.9955 & 0.0774 & 0.0539 & 0.0046 \\ 0.0635 & -0.1272 & -0.9898 & 0.0435 \\ -0.0698 & 0.9889 & -0.1315 & -0.0979 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$