Philipp Reitter, B.Sc.

# Improving robot navigation through semantic image segmentation

**Master's Thesis**

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

**Graz University of Technology**

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Friedrich Fraundorfer

Institute of Computer Graphics and Vision

Graz, January 2020

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

# Affidavit

# Abstract

Robust obstacle detection and avoidance is one of the most important functions for robotic applications, especially when moving in human shared environments. Existing solutions commonly use various ranging techniques such as LIDAR or depth from stereo. These special purpose built sensors are expensive or large in size. This thesis aims to provide a collision avoidance system for mobile robots using monocular color images and neural networks without any preexisting training data. We show a pipeline for automatically generating semantic segmentation training data for segmenting flat ground from obstacles using a stereo camera. Our method can convert the semantic segmentation output of a then-trained neural network to 3d point clouds to be used by a robot for avoiding obstacles. The proposed automatic data generation is implemented and evaluated against manually labeled ground truth data. Finally, 3d point cloud data from the neural network is compared against existing sensors on a mobile robot.

# Contents

# List of Figures

# 1 Introduction

With further development in robotic applications, more robots will be used to assist or replace humans not only in industrial areas but for everyday tasks (Buchegger et al., 2018). For example, a robot platform can carry heavy loads and follow a human operator. For such applications, good obstacle detection and avoidance is important to not harm any people in the area or not damage any obstacles in its way. For the use in human shared spaces, robots cannot simply drive the same pre-planned route or move in a certain programmed way every time, as it is often done by industrial robots. Changes and unknown obstacles are inherently present in our daily environments.

For obstacle detection and avoidance, LIDAR range sensors and depth from stereo are commonly used (Pérez et al., 2016). The cost of these specialised sensors is high and yet they have certain drawbacks or situations in which they do not provide the best results. With recent advances in machine learning it has become possible to easily segment monocular color images into classes, such as ground, sidewalk, humans, etc. By detecting objects in images, it is possible to assist existing sensors or replace them all together for certain applications. A regular color camera and a compute unit is comparably cheap, especially nowadays as computational devices, particularly for neural network inference, are becoming increasingly cheap.

In order to provide the robot with data where it is allowed to drive, the color image is semantically segmented per-pixel into either flat ground or obstacle. This information can be used by the robot to avoid detected objects and stay on track. To train a neural network on semantic segmentation tasks, a considerable amount of training data is required. In the training data, the class for each pixel in the color image is specified. When done manually, generating these per-pixel labels takes a considerable amount of human effort. Such datasets are already publicly available, but feature mostly urban road scenes for the use with autonomous vehicles. This data is not ideal for the use in small mobile robots which have a relatively low mounted camera and can operate in indoor environments.

Thus, in this thesis we aim to create a framework for providing collision avoidance for robotic applications by segmenting flat ground from a monocular RGB image using convolutional neural networks. In order to reduce human effort, we automatically label training data by using depth information from a stereo camera during data recording. Our method only requires the user to reject incorrect or keep correctly labeled frames. Computer aided labeling of training data is an ongoing topic of research as the abundance of training data is crucial for many deep learning applications. Similar tools have already been created (Dias et al., 2019; Jäger et al., 2019; Russell et al., 2008) which speed up the labeling process by providing assistance to the user for certain domain specific tasks.

The quick generation of training data allows us to produce a working system for any perspective of the camera and any environment without relying on existing datasets of the exact same setup. We evaluated the training data which was generated automatically, with minor human interaction, for an urban and an indoor environment, and found that the quality to be good with up to 98% IoU for the ground class. This works so well, because in such environments the ground can be approximated by a plane. We use the generated training data to train a neural network for the task of ground plane segmentation from a monocular RGB image. At runtime the output mask of the neural network is converted into a point cloud under the assumption that the robot is standing on flat ground. The point cloud is sent to the robot which can incorporate the information into its movement planning and avoid potential obstacles.

# 2 Prerequisites

In this chapter we give a brief overview of techniques and hardware components used. The neural network architecture we used, and our changes to it, are described in detail.

## 2.1 ICNet

Due to the real time nature of the collision avoidance task, a low inference time is favorable. Semantic segmentation via convolutional neural networks is traditionally a computationally expensive task. ICNet (Zhao, Qi, et al., 2017) provides high resolution output with good accuracy while using comparatively little computational resources. The authors achieved this by analyzing the computation time of existing networks. The introduced image cascade network (ICNet) operates on three different input scales in order to avoid costly computation of high resolution feature maps on high resolution input, as the computation increases squarely with regards to image resolution. Additionally the computation and weights in the two lowest resolution branches are shared. Figure 2.1 shows a simplified view of the architecture.

### 2.1.1 Cascade feature fusion

The individual branches are combined using the proposed cascade feature fusion (CFF) units (Figure 2.2). The authors propose this function to combine the differently scaled featuremaps from two branches together. The lower resolution feature map is first upsampled by a factor of two. After upsampling, a dilated convolution with a $3x3$ kernel is applied to combine

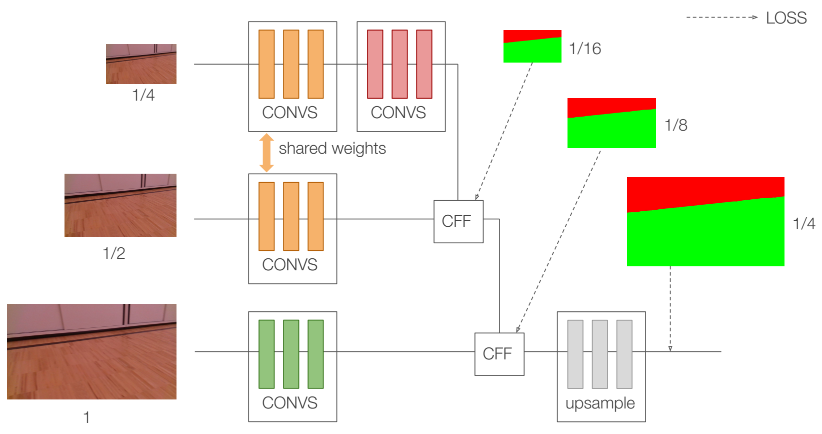


Figure 2.1: Simplified architecture of ICNet. Left shows the network input with three different input scales. The CFF blocks are he Cascade feature fusion units. The loss is calculated on three different scales to achieve label guidance for lower resolution branches.

features of neighboring pixels. This is computationally cheaper, but has similar performance as a deconvolution, which would require a larger $7x7$ kernel to achieve the same receptive field the authors claim. The higher resolution featuremap is processed by a $1 \times 1$ projection convolution before being summed with the upscaled featuremap.

### 2.1.2 Cascade label guidance

To prevent one branch from dominating the output and enhance learning, cascade label guidance is applied. For each branch, a classifier followed by a softmax cross entropy loss is appended to the final featuremap before it is fused in the CFF units (see Figure 2.2). The ground truth label $y$ is resampled to the respective resolution ($y_i$) ($1/4$, $1/8$, $1/16$) and used as the ground truth for the loss function against the classifier output $x_t$. For $T = 3$ branches, the final loss $L$ is given by the sum of the individual cross entropy losses $H_t$.

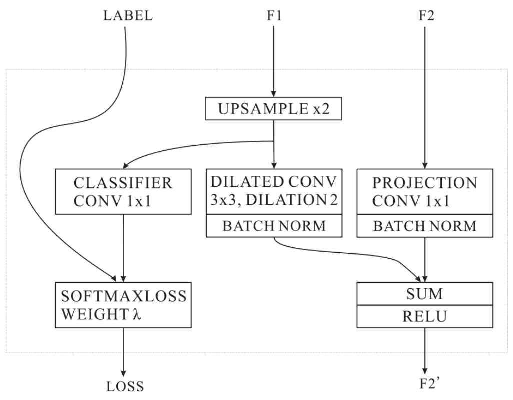


Figure 2.2: Cascade feature fusion unit used to combine two differently scaled featuremaps. Picture taken from (Zhao, Qi, et al., 2017)

$$L = \sum_{t}^{T} \lambda_t * H_t(x_t, y_t) \tag{2.1}$$

### 2.1.3 Model changes

To use the published network architecture for our project, some adaptations had to be made. Only the evaluation model and code was released by the authors. Thus, the necessary layers for cascade label guidance loss during training had to be added to the network. Contrary to the original paper, the auxiliary loss weights $\lambda_1$, $\lambda_2$, $\lambda_2$ were set to 1 as this yielded the best validation loss.

The release framework used a modified version of caffe originally developed for the use with PSPNet (Zhao, Shi, et al., 2017). Since then, the necessary layers have been added to caffe (Jia et al., 2014) and it is possible to build the same network architecture using the standard caffe framework. The PSPNet specific interpolation layers were replaced with convolution and deconvolution layers respectively and other minor syntax differences were solved.

In order to reach our performance goal of 10 frames per second, the input resolution was reduced to 512x768.

## 2.2 Robot operating system (ROS)

The robot operating system (ROS) (Stanford Artificial Intelligence Laboratory et al., 2018) is a open source framework for developing and operating robot systems. Although named ”operating system” it is not a operating system itself, but a collection of frameworks, libraries and well defined interfaces that allow the numerous individual components that make up a robot to communicate and work together.

Computation in ROS is split into individual nodes. A ROS node is a logical unit that does computation and communicates to other nodes via TCP/IP networking. One master node must exist to help other nodes connect and find each other. There are three concepts with which nodes can communicate: publish/subscribe message passing (topics), remote procedure calls (services) and a shared key/value storage (parameter server). Important concepts and parts used in this thesis are the publish/subscribe message passing and the transformation package.

### ROS topics

The publish/subscribe message passing (ROS topics) allows nodes to broadcast and receive messages for named topics. It is not important which node has published the information, but on what topic the information is sent. For each topic there can only be one type of message, which is defined by a message definition. There can be multiple publishers and subscribers per topic which all communicate peer-to-peer. Thus, heavy publish/subscribe traffic between two nodes does not influence the other parts of the system.

### Transformation system (tf)

The transformation package is a system in ROS that allows users to keep track of coordinate frames. The transformation for each coordinate frame and the time it changed is recorded and buffered for a configurable amount of time. It enables users to query the position of certain parts at a specific time in the past. This is important to compensate for latency introduced during networking or computation. The transformation of a coordinate frame can also be interpolated between two recorded time steps. Moreover the relation between coordinate frames is stored in a tree structure. For example the position of the left infrared lens is stored in relation to the mounting point of a stereo camera. If the stereo camera is to be mounted on the robot, only the position of the mounting point needs to be specified. Afterwards the position of the lenses can be obtained in relation to any other part of the robot.

## 2.3 Intel RealSense D435

The Intel RealSense D435 camera was used to record stereo images for the generation of the training data. It is a compact stereo camera specifically designed for computer vision applications. It includes two infrared cameras which provide images up to 90 times per second at 1280x720 pixels resolution. Additionally a infrared pattern projector is included which improves stereo matching, especially for plain white walls in indoor environments. A RGB camera provides additional color information at 30 frames per second with 1980x1080 pixel resolution. The infrared cameras operate with a global shutter which reduces artifacts for moving applications. The image streams can be synchronized which is essential for stereo matching.

The device houses an on-board vision processing unit which directly provides a depth output stream that is calculated from the infrared stereo images. This greatly reduces the computational resources needed on the host device. The camera connects to the host device via a USB3 interface. For our application, the depth output stream was not used, as offline computation of the depth map provided better results and allowed for parameter adjustments after recording.

Intel provides software development kits, examples and a ROS node for the camera.

## 2.4 NVIDIA Jetson

The NVIDIA Jetson is a small form factor compute unit with an on-board NVIDIA Pascal GPU. The compute unit allows normal Linux distributions. The onboard GPU's driver support CUDA and OpenCL for computational tasks. Neural network libraries and frameworks such as caffe or tensorflow can be used without limitations.

The small form factor and energy efficiency is ideal for mobile robots and reduces complexity which comes with housing comparatively large consumer graphics processing units in a mobile platform. Due to this it is an attractive compute platform for mobile robots and it has been heavily used in this field (Piemngam, Nilkhamhang, and Bunnun, 2019; Tang, Ren, and Liu, 2017; Bokovoy, Muravyev, and Yakovlev, 2019).

# 3 Training data generation

The training data generation is split into five steps. First the recorded data is reduced by removing visually similar color images, as they do not add significant information for later training. Then the point cloud is generated from the infrared stereo images. In the third step, the ground plane is segmented from the point cloud. In step four, the identified ground plane and obstacles are reprojected into the color image frame and the generated semantic mask is refined. The final step involves the only human interaction as every frame is manually reviewed and either removed or kept.

## 3.1 Data Reduction

The camera is recording infrared and color images at 30 frames per second. This yields 9000 Frames for just a 5 minute recording. In comparison, the widely used Cityscapes dataset (Cordts et al., 2016) has 5000 frames in total, but features a much greater variety of different scenes. It is preferable to reduce the dataset size and sort out similar frames as there is usually no significant change from two consecutive frames at 30 frames per second.

To automatically sort out similar color frames, a dissimilarity measure was used. If the measure exceeds a threshold, the next frame is chosen. The dissimilarity ($SAD$) is based on the sum of absolute difference of all pixels between the reference frame ($I_R$) and the current frame ($I_C$).

$$SAD = \frac{1}{UV} \sum_{u}^{U} \sum_{v}^{V} |I_R(u,v) - I_c(u,v)| \tag{3.1}$$

After the similarity threshold is exceeded, a batch of up to $N_{batch}$ following images is collected. Of this batch, the sharpest frame is chosen, using a sharpness measurement. The sharpness $S$ of an image is measured using the variance of its laplacian as described by Pech-Pacheco et al., 2000.

$$S = \sum_{u}^{U} \sum_{v}^{V} (\Delta I(u,v) - \overline{\Delta I})^2 \tag{3.2}$$

Where $\Delta I$ is the laplacian of the image and $\overline{\Delta I}$ its average value. The absolute values of the sharpness measure is not important as the frame with the highest sharpness value is chosen from the batch.

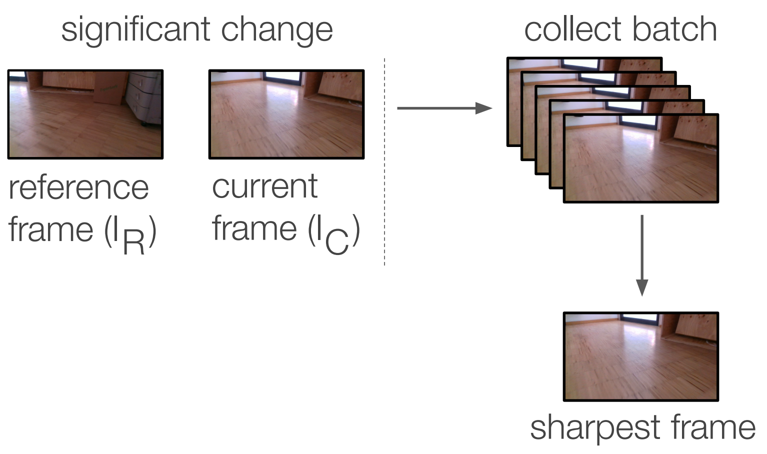


Figure 3.1: Schematic view of the data reduction method

## 3.2 Point cloud calculation

The disparity map was calculated from the recorded left and right infrared images using the Semi Global Block Matching algorithm (Hirschmüller, 2005). The real-time depth information form the camera was not used, as it did not allow for any parameter changes after recording. For each pixel at position $(u, v)$, a point is projected into 3d space from the disparity $(d_{u,v})$ and the disparity to depth matrix $(Q)$. This assumes the images are rectified.

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = Q \times \begin{bmatrix} u \\ v \\ d_{u,v} \\ 1 \end{bmatrix} \tag{3.3}$$

For two identical, parallel cameras, the disparity to depth matrix $Q$ can be constructed from the camera's focal point $(c_x, c_y)$, it's focal distance $(f)$ and the baseline $(t)$

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/t & 0 \end{bmatrix} \tag{3.4}$$

Any points, which are further than 10 meters away from the camera, are removed. These points are usually erroneous. The used camera specifies it's depth range up to 10 meters and for the selected use case of collision avoidance for small mobile robots, obstacles beyond 10 meters are not important

## 3.3 Point cloud segmentation

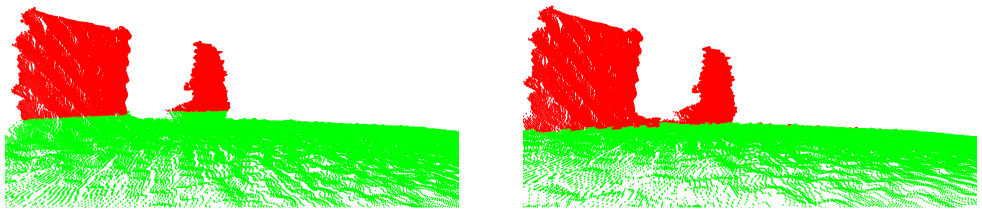
The point cloud is segmented using the random sample consensus algorithm (Fischler and Bolles, 1981) (RANSAC). The advantage of using RANSAC over for example fitting a plane through all points using linear regression, is its ability to ignore outliers. An outlier in this case is any point which is not part of the ground plane, e.g. an obstacle.

RANSAC iteratively chooses a set of minimal points to generate a hypothesis. In our case three points are randomly chosen and for the hypothesis a plane is fitted through these points. For each guess, the number of points that fall within a certain error margin around the plane is calculated. After a number of tries the hypothesis with the most amount of inliers is chosen. Finally one plane is fitted through all inliers of the best hypothesis. The output of the algorithm is a plane model which fits the most points. By applying a threshold to the distance from the plane, points that are considered part of the ground plane can be calculated.

### Improving segmentation with surface normals

Using only the point distance from the ground plane as the inlier criteria has a downside. Points of obstacles which are near to the ground (where an obstacle is touching the ground) are also considered part of the ground (see Figure 3.2 (a)).

Simply reducing the distance threshold to a suitable value results in false negative matches for actual ground points. By using the surface normal as an additional inlier criteria, the distance threshold can be higher while still creating sharp edges around obstacles (see Figure 3.2 (b)).

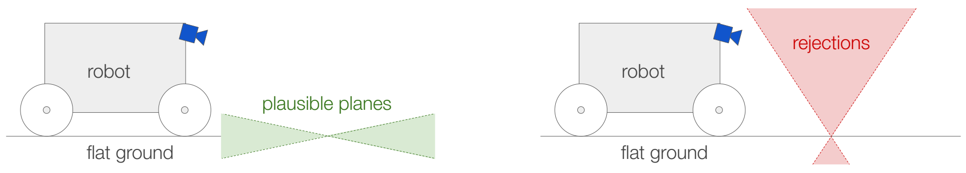


(a) only using the thresholded distance (b) with normal vector constraint

Figure 3.2: Examples of the two different methods for calculating a plane's inliers

### Additional constraints

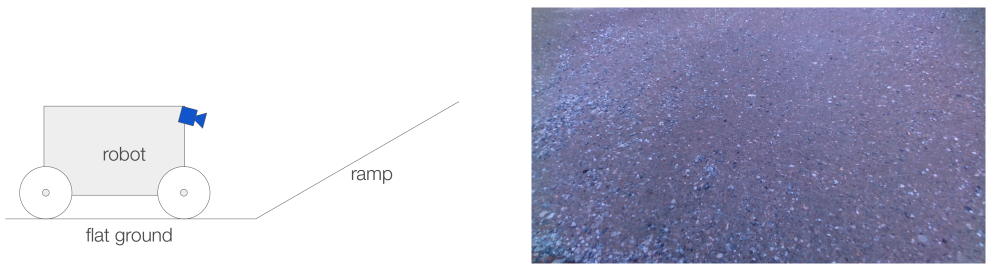
To further improve results, plane hypotheses were rejected based on their orientation. Because the robots frame is rigid and its wheels form a relatively rigid connection to the (assumed) flat ground, planes can be rejected based on this information. By knowing the orientation and position of the camera on the robot, all planes can be rejected on which the robot could not plausibly stand on (see Figure 3.3).



(a) plausible angles (b) implausible angles

Figure 3.3: Rejection of ground plane hypothesis based on the angle relative to the robot

This naive approach also rejected planes from ramps, when the robot was not yet on a ramp (see Figure 3.4). In this case, in combination with a slight downward tilt of the camera, the plane appears to be too steep for the robot to stand on. Thus, the threshold for rejecting a plane had to be chosen relatively high and only planes of walls directly facing the camera could be reliably rejected. Due to the high threshold it was not possible to identify and reject poorly matched ground planes, as setting a low threshold would reject many correct ground hypotheses as well.



(a) schematic view (b) camera view

Figure 3.4: Robot in front of a ramp

## 3.4 Mask generation

To generate the semantic mask, the points are reprojected into the color image frame. By using the color camera's camera matrix and its extrinsic parameters with respect to the depth image, the individual points from the point cloud can be projected into the color image using the pinhole camera formula.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A[R|t] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

Where $[R|t]$ are the extrinsic camera parameters, composed of the $3 \times 3$ rotation matrix $R$ and a $3 \times 1$ translation vector $t$. The pinhole model assumes that the image is not distorted and can only be used, because the whole pipeline assumes all used images are rectified.

After projecting the point cloud points into the color image frame, holes are present, due to rounding. Because a point is assigned to exactly one pixel and not every pixel in the color image is has a matching a point in the point cloud. To close the holes, a morphological close operation is applied on the image (see Figure 3.5 (a) and (b)).

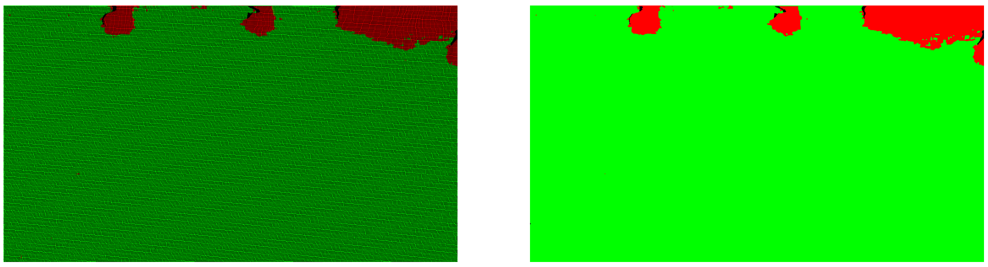

(a) projected mask before closing (b) projected mask after closing

Figure 3.5: Projected mask from 3d point cloud

## 3.5 Mask refinement

The naively generated mask can further be improved by removing known artifacts from bad disparity feature matching and by removing unlikely results. Furthermore, the color image is used to refine edges by using its contextual information and conditional random fields.

### 3.5.1 Conditional Random Fields

Conditional random fields (Lafferty, Mccallum, and Pereira, 2001) are commonly used as a postprocessing step for image segmentation tasks (Yu and Fan, 2019). They improve the segmentation edges by taking into account the contextual color information of the image. Neighbouring pixels with similar colors are more likely to be of the same class.

Figure 3.6 (a) shows a mask before and (b) shows a mask after CRF refinement. After CRF refinement, the ignore mask, which contains pixels where no depth information was able to be calculated by the stereo matching algorithm, is applied back to the mask. Thus, CRF is not used to infill areas where no depth information is available as this introduced more errors and did not improve the overall output quality (see section 6.2.2).

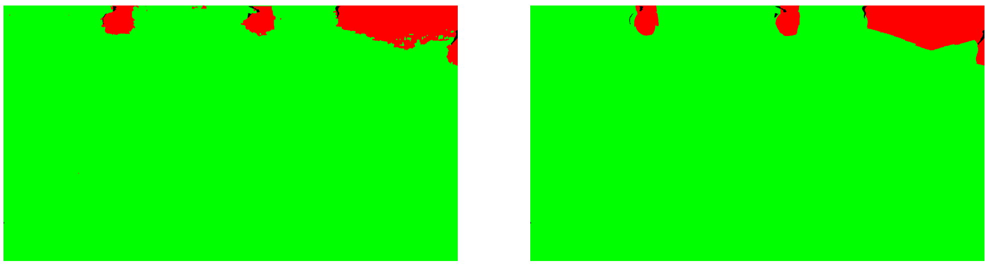
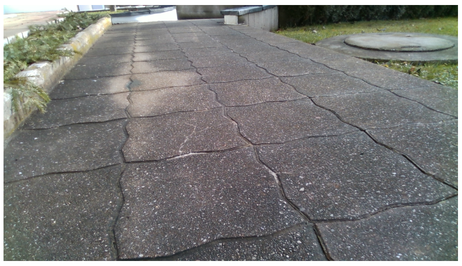

(a) mask before crf refinement (b) mask after crf refinement

Figure 3.6: Mask CRF refinement

### 3.5.2 Not Connected Components

We assert that all ground which should be classified as ground is connected together. This assertion allows us to further assume that the ground plane is one connected component as seen from the perspective of the camera. Thus, we remove all pixels from the ground plane mask which are not part of the biggest connected component.

Figure 3.7 (a) shows an image where a disconnected round part in the top right corner is on the same height as the ground plane. This part would have been labeled as ground, but is clearly not accessible by the robot. The final mask seen in (b) shows that the round part was marked black, which indicates the ignore class. The ignore class is used for pixels which we cannot assign to either ground or obstacle for sure. This class will later not have an affect on the loss function during training.


(a) the color image showing a same-height, round surface on the top right

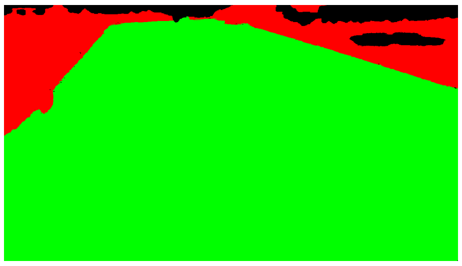

(b) final mask with the not-connected component set to ignore (black)

Figure 3.7: Example for removing non-connected ground components

Furthermore, we remove tiny connected components which are part of the obstacle mask, since those were found to be part of noise from bad stereo matching or artifacts from the CRF refinement. Figure 3.8 (a) shows small speckles from stereo matching artifacts at a corner and (b) shows the cleaned mask where small components were removed from the obstacle mask. When removing components, the new value for the mask is set to the the ignore class.

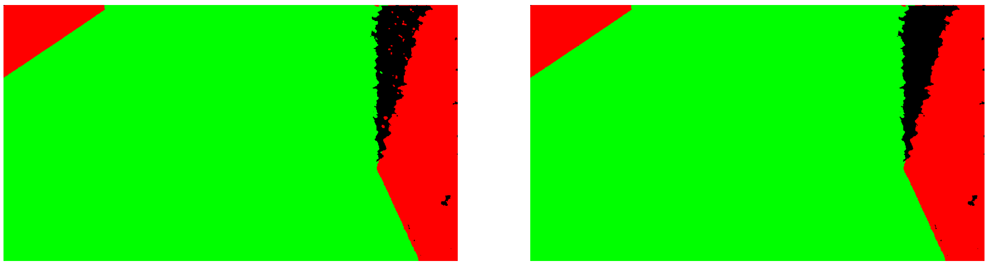

(a) example of stereo matching artifacts on a corner (b) cleaned mask through the removeal of small obstacle components

Figure 3.8: Example for removing small connected obstacle components

## 3.6 Manual review

After generating the mask proposals, all images are once reviewed by a human. The reviewer only has the possibility to remove the mask from the generated dataset or keep it. Generating new data is comparatively cheap with respect to the reviewer correcting misclassified pixels in the image. Thus, we only provide the option to remove the mask entirely or accept it as is.

# 4 Inference

The training data from Chapter 3 is used to train the neural network. At runtime, only a monocular RGB camera is used. The output of the neural network is a segmentation mask with two classes. One class denotes the ground plane, and the other class denotes obstacles. This chapter describes how this segmentation mask is processed.

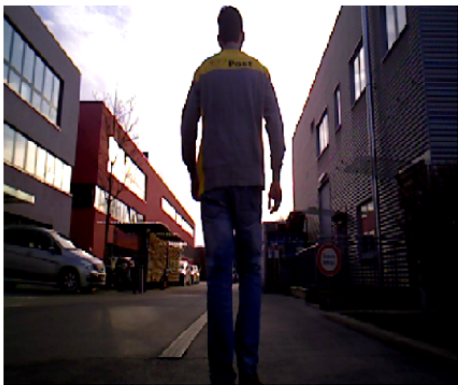
## 4.1 Point cloud from mask

The output of the neural network is a semantic segmentation mask. This mask cannot be used directly by ROS and its movement planning software. The best data format for ROS and other nodes to work with, are 3d point clouds. Thus, the segmentation mask is converted into a point cloud. The conversion happens in two steps: first, the boundary pixels of the mask are extracted. Secondly each boundary pixel is projected into 3d space assuming the robot rests on flat ground. Since no depth information will be present at runtime, we had to make this assumption about the surrounding ground in order to calculate 3d position solely from the boundary pixel's position.

### 4.1.1 Boundary pixel extraction

The most important points in the generated point cloud are the nearest obstacles. For every column in the image, the lowest pixel is searched, which is classified as an obstacle. All other pixels above this pixel are further away from the robot. These may be important for macro movement plans, but for the task of collision avoidance they are not useful. In addition, the further away points experience more error and, thus, may not even be useful for macro planning.

Figure 4.1 (a) shows the color image and (b) the output of the neural network in white/black. The calculated boundary pixels are colored red. The white dot in the center shows how only the lower object boundary is calculated. The white dot itself has no influence on the resulting point cloud since it is behind an obstacle.


(a) color image

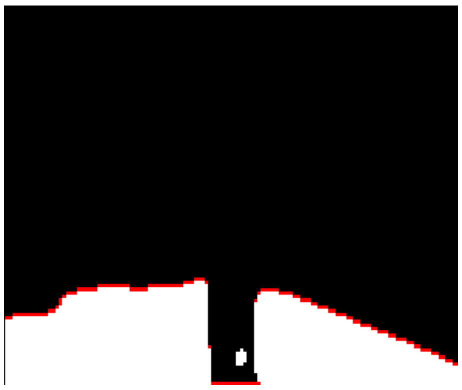

(b) black/white: segmentation result, red: calculated boundary pixel

Figure 4.1: Result of the inference at runtime and boundary pixel extraction

### 4.1.2 Projection into 3d

For each boundary pixel, a 3D ray is constructed. By knowing the cameras transformation frame, i.e. its rotation and translation relative to the robot base and the ground plane, the ray can be constructed in the robot's coordinate system. This is important as elevation changes (in the z-axis) of the robot should not impact the results.

The ground plane is assumed to be parallel to the robot base transformation, offset in the z-axis by the predefined height of the wheels ($h_{wheel}$). The ground plane is defined as a point $p_0$ through which it passes with its normal vector $p_n$.

$$p_n = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T, p_0 = \begin{bmatrix} 0 & 0 & -h_{wheels} \end{bmatrix}^T \tag{4.1}$$

A given camera ray $l$ with origin $l_0$ at the camera position, intersects with the ground plane if the ground plane's normal vector is not parallel to the ray, thus

$$p_n \cdot l \begin{cases} = 0 & miss \\ \neq 0 & hit \end{cases} \tag{4.2}$$

The intersection distance $t$ can be calculated by

$$t = \frac{(p_0 - l_0) \cdot p_n}{n \cdot l} \tag{4.3}$$

Since the calculated ray has no direction, the intersection distance has to be positive ($t > 0$) for a intersection to be in front of the camera. The final point $p$ of the point cloud is then calculated by

$$p = l_0 + l \cdot t \tag{4.4}$$

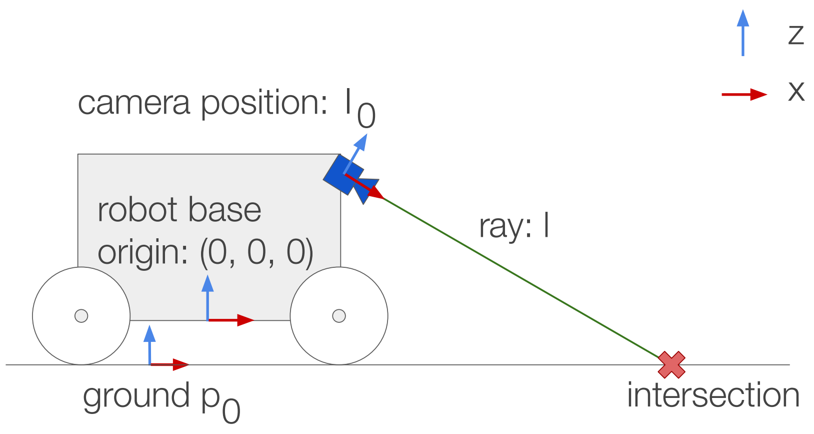
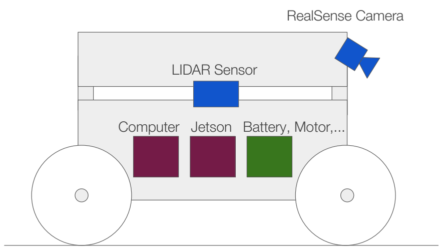


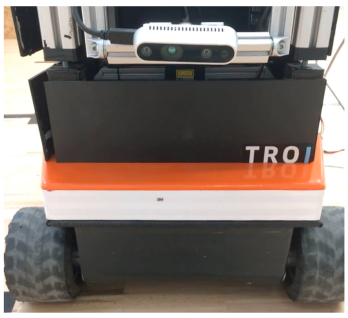
Figure 4.2: X/Z Plane of the camera ray ground intersection

# 5 Implementation

## 5.1 Robot Setup



(a) schematic view of the robot



(b) front view of the robot

Figure 5.1: The used robot platform

The robot used for evaluation is equipped with an on-board computer, a LIDAR range sensor and motors. The robot's software is ROS based, with nodes for planning, collision avoidance and movement already setup. The robot can either be controlled with a controller or be used in follow-mode where it follows a person.

The camera was mounted on the front of the robot and its position and rotation was registered with the robots transform system.

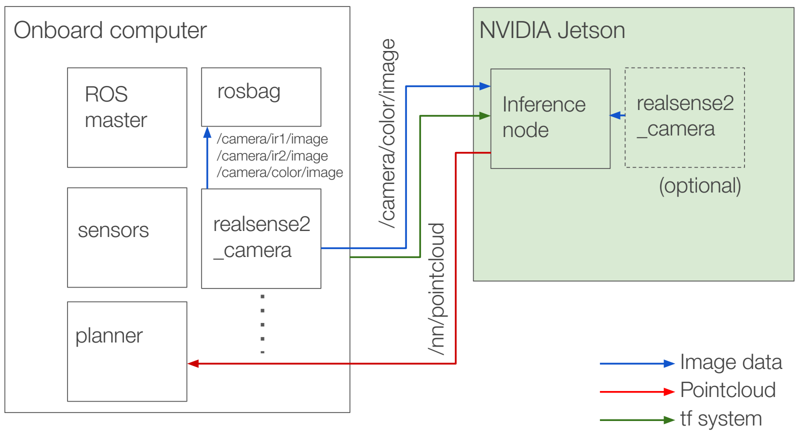


Figure 5.2: Schematic view of the ROS system

### 5.1.1 ROS Setup

#### Intel RealSense Node

Intel already provides a ROS node named realsense2_camera for the RealSense camera lineup. The camera can be configured for different resolutions and framerates via ROS launch parameters. The list below describes the used configuration.

- Color resolution: 1280x720
- Color FPS: 30
- Infrared resolution: 1280x720
- Infrared FPS: 30
- Depth stream: off
- IR projector: on
- Synchronize streams: on

The camera is factory calibrated and the realsense2_camera ROS node publishes all information of the cameras via ROS topics and the ROS transform system. The intrinsic camera parameters are published in the image topic via the camera information parameter. The images are already rectified and thus, the distortion coefficients are zero.

The extrinsic parameters are published through the ROS transform system. Since the camera's body position is defined in the transform graph the position of the individual cameras can be calculated easily with respect to any reference frame, through the use of the ROS transform system. This is important later when the obstacle positions are projected into 3d space from the camera image.

The camera was connected to the on-board computer's USB3 port during recording, testing and evaluation. This was done because the recordings were saved on the storage of the on-board computer. Since the raw image data was in excess of $1GBit/s$ the network connection between the inference node and the on-board computer would have been over-saturated. In our setup, only the RGB stream has to be transferred over the network link. This setup, of course, introduces additional latency.

When recording is not necessary, the setup can be changed. The camera can be connected directly to the inference node as shown with the dashed box in Figure 5.2. In this case, the Jetson can act as a self contained ROS node, providing the finished inference results to other connected nodes.

### Data recording (rosbag)

The data was recorded in ROS with the rosbag package. This package records a list of topics directly to disk. The color image stream as well as the left and right infrared stream were recorded. For all recordings, the uncompressed raw data was recorded to allow for reproducible results. Rosbag was launched with the following parameters:

```
$ rosbag record \
    --split --size 1024 \
    /camera/color/image_raw \
    /camera/color/camera_info \
    /camera/infra1/image_rect_raw \
    /camera/infra2/image_rect_raw \
    /camera/infra1/camera_info \
    /camera/infra2/camera_info \
    /tf
    /tf_static
```

The *–split –size* parameter allows us to split the recorded files into 1*Gb* files called chunks. This eases the post processing as it allows the whole file to be loaded into memory with reasonable hardware requirements.

The recorded topics are the image streams and their respective *camera_info* topic on which the intrinsic camera parameters are published. Lastly the topics of the transformation subsystem *tf*, *tf_static* are recorded. They are used to calculate the the extrinsic camera parameters.

Special care has to be taken in order to keep the recording rate below the write speed of the disk as pent-up data will be dropped. The data throughput is significant since the video streams are recorded as uncompressed raw data to provide reproducible results from the recordings. Early tests showed that the data recording became unstable at high throughput, which caused skipped frames on either one of the infrared cameras. This causes synchronization problems when testing since the stereo matching is very sensitive to time offsets between the stereo image pairs.

#### Other nodes

A number of additional ROS nodes are executed on the on-board computer, which handle movement, sensors, control input and safety features of the robot. These are already setup and configured by the robot platform.

## 5.2 Data generation

The data generation described in Chapter 3 was implemented in Python3 with the help of open source libraries. The source code and examples are also available on the project website (Reitter, n.d.). The pipeline is split into individual steps similar to the description in Chapter 3. Each step is executed individually via separate python files.

Each step is written to only process new data and skip already processed files. This allows you to simply run the whole pipeline again after creating new recordings files. No parameters need to be passed specifically for each recording. To reset or rerun the pipeline for a specific dataset, simply delete its corresponding folder in the configured data output folder and re-run the pipeline.

### 5.2.1 Dependencies

The project depends on certain python libraries to be present, the most important ones are described here. A full list of dependencies can be found in the included requirements.txt file.

The **rosbag** package is used to read the rosbag recording files. It is required for step 1 of the process. This library is automatically present if ros is installed or can be installed via the python package manager.

**OpenCV** (Bradski, 2000) is used for common image manipulation tasks and the point cloud computation. Its implementation of the Stereo Block Matching Algorithm was used to convert the two infrared images into a disparity map. Further, the projection from the disparity map into the 3d point cloud and the projection from the 3d point cloud into the color image frame were sped up with the help of opencv's native implementation.

**python-pcl and the point cloud library (Rusu and Cousins, 2011)**: The calculation of the normal vectors of the point cloud points and the plane segmentation with RANSAC was done using the point cloud library. To access the point cloud library functions in python the python-pcl wrapper was used. It allows us to access some of its functions from python.

**PyDenseCRF** (Beyer, n.d.) is a python wrapper for the method described by Krähenbühl and Koltun (2012). It is used for the mask refinement in step 5.

### 5.2.2 Configuration

The configuration file *config.py* defines the source folder of the recordings (rosbag files) and also the destination folder of the output files. Furthermore, thresholds and parameters for various computation steps can be configured. The configuration file contains comments about what effect the individual parameters have. The default configuration works well for the used Intel RealSense D435.

#### Inverted mode

If the camera was mounted upside-down, the dataset can be processed as-well without any changes to the code by changing the dataset's name to end in *_inv*. This is useful when the camera is mounted upside down on the robot or a second recording is made with the camera upside down to avoid problems with occlusion as discussed in section 6.2.1.

### 5.2.3 Step 1: Data extraction

This script extracts the images and extrinsic and intrinsic camera parameters from the rosbag file. It expects a certain format in which the files have to be in the configured input folder. Each recording session has its own subfolder, in which rosbag records the individual files one by one. The name of the final dataset will be the name of the folder. The project includes a utility script *record.sh* which automatically creates a folder and launches rosbag with the correct parameters.

```
recordings/
  - indoor1/
    - indoor1_2000-01-02-15-42-52_0.bag
    - indoor1_2000-01-02-15-42-52_1.bag
  - indoor2/
    - indoor2_2000-01-02-13-22-12_0.bag
    ...
```

Step 1 produces the following output files per chunk of each recording.

- **color.npy** all color images as numpy matrix of type *uint8* and shape $N \times W_c \times H_c \times 3$.
- **infra1.npy, infra2.npy** the left and right infrared images as a numpy matrix of shape $N \times W_i \times H_i \times 1$.

Where $N$ is the number of output recorded frames, $H_c$, $W_c$ is the resolution of the color image and $H_i$, $W_i$ the resolution of the infrared images.

#### Frame synchronization

The script expects both infrared frames and the color frame to be in sync. This is enforced by only extracting IR and color frames where the timestamp of the message matches exactly.

This was found to be the case for the Intel RealSense D435, as long as the exposure of the color camera is lower than one over the infrared camera's framerate. If the color camera's exposure is set to automatic, frames may be skipped in low lighting conditions. The output of the script shows how many matching and non-matching frames were found.

```
$ python step1_bagfile.py
looking for bagfiles in /home/user/Documents/
extracting /home/user/Documents/demo/2020-01-03-12-02-23_1.bag
found 192 matching pairs of 237 total frames
extracting /home/user/Documents/demo/2020-01-03-12-02-13_0.bag
found 210 matching pairs of 236 total frames
extracting /home/user/Documents/demo/2020-01-03-12-02-33_2.bag
found 72 matching pairs of 97 total frames
```

### 5.2.4 Step 2: Data reduction

This step reduces the recorded frames based on the technique described in section 3.1. The threshold for the dissimilarity measure and the batch size ($N_{batch}$) from which the sharpest image is selected can be set in the configuration file. The data reduction is performed for each chunk of the recording individually. After all chunks of a recording are reduced they are combined into one file and the temporary files from each chunk are deleted.

The batch size, from which the sharpest frame will be drawn, dictates the maximum rate at which new frames will be chosen and saved in the final output. This rate can be calculated by

$$FPS_{max} = FPS_{recording} / N_{batch} \tag{5.1}$$

where $FPS_{recording}$ is the frame rate at which the camera recorded. Additionally, if the camera image was still during parts of the recording, the final output frame rate may be lower.

Step 2 produces the same output files as step 1, but with reduced data and all chunks of a recording will be combined into one output file. The step's output lists all processed steps.

```
$ python step2_reduce_data.py
reduced /data/demo_2020-01-03-12-02-23_1 from 192 frames to 14
reduced /data/demo_2020-01-03-12-02-13_0 from 210 frames to 13
reduced /data/demo_2020-01-03-12-02-33_2 from 72 frames to 3
combining /data/demo
deleting /data/demo_2020-01-03-12-02-23_1
deleting /data/demo_2020-01-03-12-02-13_0
deleting /data/demo_2020-01-03-12-02-33_2
```

### 5.2.5 Step 3: Point cloud calculation

This step takes the previously extracted infrared images from *infra1.npy* and *infra2.npy* and calculates a 3d point cloud without any color information. This point cloud is saved to *coords.npy* as a numpy array with type *float32* and shape $N \times (W_i \cdot H_i) \times 3$. Any points without disparity information or out of plausible range are filled with *NaN* values.

The parameters for the Stereo Block Matching algorithm can be changed in the configuration file. The extrinsic and intrinsic camera parameters for the projection into 3d are taken from the data camera parameters which are published on the ROS camera topic and the ROS transform system. For this step to work properly the camera has to be properly setup in ROS and defined in the transformation graph.

### 5.2.6 Step 4: plane segmentation

Step 4 uses the calculated point cloud from the previous step and extracts the ground plane. The calculated plane's model is saved to *planes.npy* . Since the segmentation is not only based on the distance from the ground plane, but additional constraints, a map of all inliers is saved in *good_idx.npy* as a numpy array with type *bool* and shape $N \times (W_i \cdot H_i) \times 1$ .

### 5.2.7 Step 5: mask generation

In this step, the calculated inliers are projected into the color camera's reference frame. The initial mask is refined as described in section 3.4. The refined mask and the color images are saved into individual folders in the the output folder. The masks are saved as indexed png files. Where index 0 is the ground class, index 1 is the obstacle class and index 255 is the ignore class.

```
output/
  - indoor1/
    - color/
      indoor1_00000.png
      indoor1_00001.png
      ...
    - label/
      indoor1_00000.png
      indoor1_00001.png
      ...
```

### 5.2.8 Step 6: Mask refinement

This step implements the connected component refinement described in section 3.5.2. The cleaned masks are saved analog to the input masks in an adjacent folder named *label_clean*.

### 5.2.9 Step 7: Manual review

The last step in the pipeline is the human review step. A user interface was implemented to assist the reviewer in manually flag each frame as good or bad. The reviewer is presented with a blended image of a mask and the color image. He then has to decide whether to keep the frame or remove it. Additionally, a frame can be skipped to be reviewed later.

Keyboard actions:

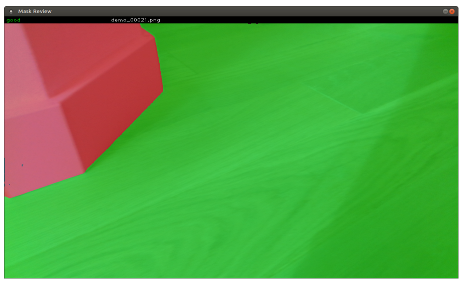
- **Return** mark mask as good
- **Backspace** mark mask as bad
- **Left Arrow / Right Arrow** go one mask backwards / forwards
- **Escape** close the interface and save all changes
- **1, 2, 3** switch between (1) blended, (2) mask only or (3) color only view mode

The user interface outputs two text files in the output folder named *good.txt* and *bad.txt* with the respective file paths of good and bad masks. These files can then be used to collect the data for training the neuronal network.

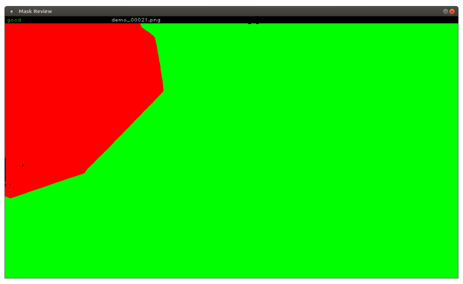
## 5.3 ROS Inference Node

The inference node was written in C++. It is responsible for executing the neural network on the RGB image stream and output the segmentation mask as well as the collision point cloud.

The inference node receives the RGB image data from the master node. Again the data is sent uncompressed to keep result reproducible. The inference process is pipelined and split up into three parts and three threads



(a) blended view mode



(b) mask only view mode

Figure 5.3: Review utility user interface

to allow the GPU to work without interruption and achieve maximum throughput. Figure 5.4 shows a schematic view of the inference node and its threads. Receiving the images and sending the inference results is mostly bound by the i/o speed of the network. Messages between the threads are passed using shared memory and do not add any significant additional overhead.

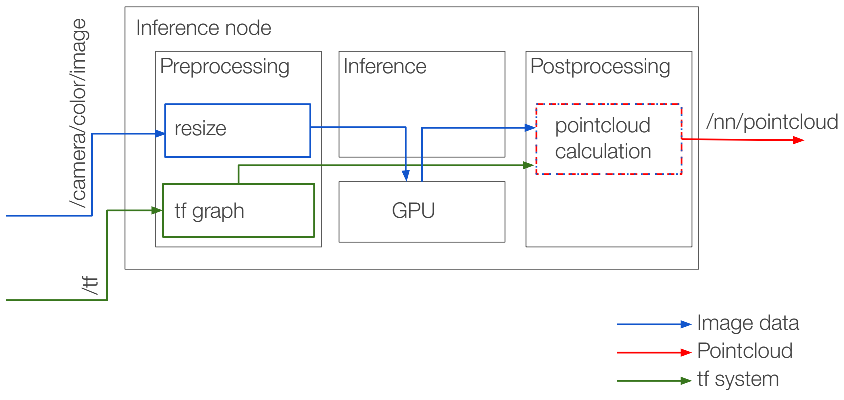


Figure 5.4: Schematic view of the inference node

### 5.3.1 Preprocessing thread

The preprocessing thread receives RGB images from the realsense2_camera image publisher node and resizes them to the correct dimensions for inference. The RGB image is then sent to the inference thread for processing. The receiver thread also listens to transform updates and stores a local copy of the transform graph. This information is later used by the publisher thread.

In order to avoid stale frames between the preprocessing stage and the inference stage, the preprocessing thread constantly replaces the next frame of the inference stage with new frames it receives from ROS until the inference thread picks it up.

### 5.3.2 Inference thread

The inference thread takes the resized images from the Preprocessing thread and processes them with the neural network. The output of the model is a 2d segmentation map with two classes per pixel. One class denotes driveable ground and one class obstacles. The class probabilities are converted into binary where 1 is driveable ground and 0 is obstacle. The result is sent to the postprocessing thread.

### 5.3.3 Postprocessing thread

The publisher thread receives the segmentation output and projects the results into 3d points as described in section 4.1.2. The 3d points are then converted into a ROS point cloud object and made available as a ROS topic to any node which listens on the topic.

### 5.3.4 Performance results

Pipelining a process always introduces some additional latency. The latency of our inter process message passing was measured to be around 10 microseconds and thus is negligible in the following calculations.

| stage | processing time (ms) | frames per second |
|---|---|---|
| ROS package decoding | 0.9 | 142.9 |
| preprocessing | 6.1 | |
| inference | 95.2 | 10.5 |
| postprocess & publish | 0.7 | 1428.6 |
| total latency | 102.9 | 9.7 |
| throughput | | 10.5 |

The greatest benefit in terms of processing time is due to the separation of the preprocessing and the inference stage, as the preprocessing stage takes time to resample the image into fitting dimensions. Other processing times, e.g. the sending/receiving of packages in ROS is outside of our scope and adds a fixed amount of time we can hardly optimize for.

# 6 Evaluation

The evaluation was split into three parts.

- Evaluating the effort and time required to collect a dataset for a specific indoor environment and camera position.
- Evaluating the quality of the generated training segmentation masks and the the neural network output.
- Evaluating the differences between the generated point cloud during normal usage of the robot and the already existing on-board sensors.

The first part shows the difference in the time required for processing recordings manually or via our proposed method. The key difference is the yield of our method in comparison to the manual approach. This is because the some frames are rejected due to bad stereo matching, incorrect ground plane estimation or other artifacts. Contrary to this a person labeling the frames had no problems correctly labeling all frames. In the second part we show quantitative measurements on the quality of the generated masks and subsequently the quality of neural network output. We show that the quality of the generated masks is suitable for successfully training a neural network. The third section shows a qualitative evaluation of the final system and compares the results with a LIDAR sensor.

## 6.1 Evaluating data generation

The goal of this thesis is to provide a pipeline for quickly generating training data for new environments or different camera positions. In order to quantify the human effort required, we compare the time required for recording, the processing time and time for the manual review against the time it took to manually label the same number of images.

### 6.1.1 Data recording

For this evaluation two datasets were created. One outdoor dataset and one indoor dataset of a residential home. For recording the camera was not mounted to the robot but to a movable platform acting as a replica. This platform was moved around the indoor and outdoor environments. The recordings initially consisted of:

- 4.7 minutes indoor recording, resulting in 8453 frames
- 3.3 minutes outdoor recording, resulting in 5946 frames

### 6.1.2 Data reduction

During data extraction only synchronized frames are extracted, thus in the extraction step already 768 frames were lost. During the data reduction step the 13613 in-sync frames were reduced to 825. Finally, during the human review 99 masks rejected.

| Step | # frames indoor | # frames outdoor | yield (%) |
|---|---|---|---|
| recording | 8453 | 5946 | 100 |
| synchronized extraction | 7990 | 5623 | 94.5 |
| data reduction | 468 | 357 | 6.1 |
| review | 451 | 275 | 88.0 |

The total yield is only 5% which is to be expected as the batch size for drawing the sharpest frame ($N_{batch}$) was set to 15. This limits the maximum yield to 6.6% of initially recorded frames. A better measure of the total yield is how many frames of the maximum possible frames (due to the batch size) are selected as good masks. This yield value is 75%. In comparison the manually labeled frames had a effective yield of 100% because the person labeling the images was able to understand and separate each individual frame.

### 6.1.3 Processing Time

For comparing the processing time, data transfer from the robot and the actual time of the recordings was not considered, as both our automatic approach and manual labeling requires these steps.

The total processing time of our pipeline for both recordings was 34 minutes. This includes reading and writing the data and temporary results from and to the system's disk. After processing, the dataset was reviewed by a human. The dataset's 825 images were reviewed in 24 minutes. For generating 726 good masks the entire process took 58 minutes.

Contrary to this the 100 randomly selected images used for the testset, which were manually labeled with the help of photo editing software, required about 200 minutes time.

## 6.2 Evaluating mask quality

From the recorded masks we randomly selected 100 images for a test set. These images were not part of the training or validation set. The images were manually labeled by a human using common photo editing software in section 6.1. For each of the test images the IoU was calculated for the automatic generated mask, and the output of the trained neural network.

In the following, sources describe which segmentation mask was evaluated. The **naive** mask is the output from the ground segmentation without any post processing. **crf refined** is after crf refinement as described in section 3.4 and **cc cleanup** is the final mask after the connected component cleanup. **nn** shows the output of the trained neural network.

Outdoor

| Source | IoU ground (%) | IoU obstacle (%) | mIoU (%) | density (%) |
|---|---|---|---|---|
| naive | 95.8 | 81.2 | 88.5 | 95.6 |
| crf refined | 98.0 | 89.3 | 93.7 | 95.6 |
| cc cleanup | **98.1** | **91.6** | **94.9** | 95.4 |
| nn | 96.7 | 88.4 | 92.5 | **100** |

Indoor

| Source | IoU ground (%) | IoU obstacle (%) | mIoU (%) | density (%) |
|---|---|---|---|---|
| naive | 97.4 | 82.1 | 89.8 | 98.8 |
| crf refined | **98.2** | 85.9 | 92.0 | 98.8 |
| cc cleanup | **98.2** | **86.1** | **92.2** | 98.7 |
| nn | 97.9 | 84.7 | 91.3 | **100** |

### IoU comparison

Contrary to initial expectations the outdoor dataset shows better mIoU values. We expected that the varying lighting conditions from sunlight and the greater distances would decrease outdoor performance. The difference in mIoU stems mostly from the obstacle IoU. When comparing indoor and outdoor results it becomes clear indoor there are many additional obstacles of small or thin sizes. These are for example chair or table legs.

In the outdoor environment most obstacles are walls or ledges. This becomes clear when analyzing the difference in obstacle size for the in and outdoor recordings and their respective class distribution.

| recording | ground (%) | obstacle (%) | median obstacle size (px) |
|---|---|---|---|
| indoor | 82.7 | 17.3 | 200841 |
| outdoor | 70.4 | 29.6 | 436636 |

### Density comparison

The density of the outdoor dataset is lower due to the fact that points at large distances beyond the camera's specifications are set to ignore. Such great distances are not present in the indoor dataset.

The pixel density will always decreases during the non-connected component refinement step, as removed components are set to the ignore class and not the opposing class.

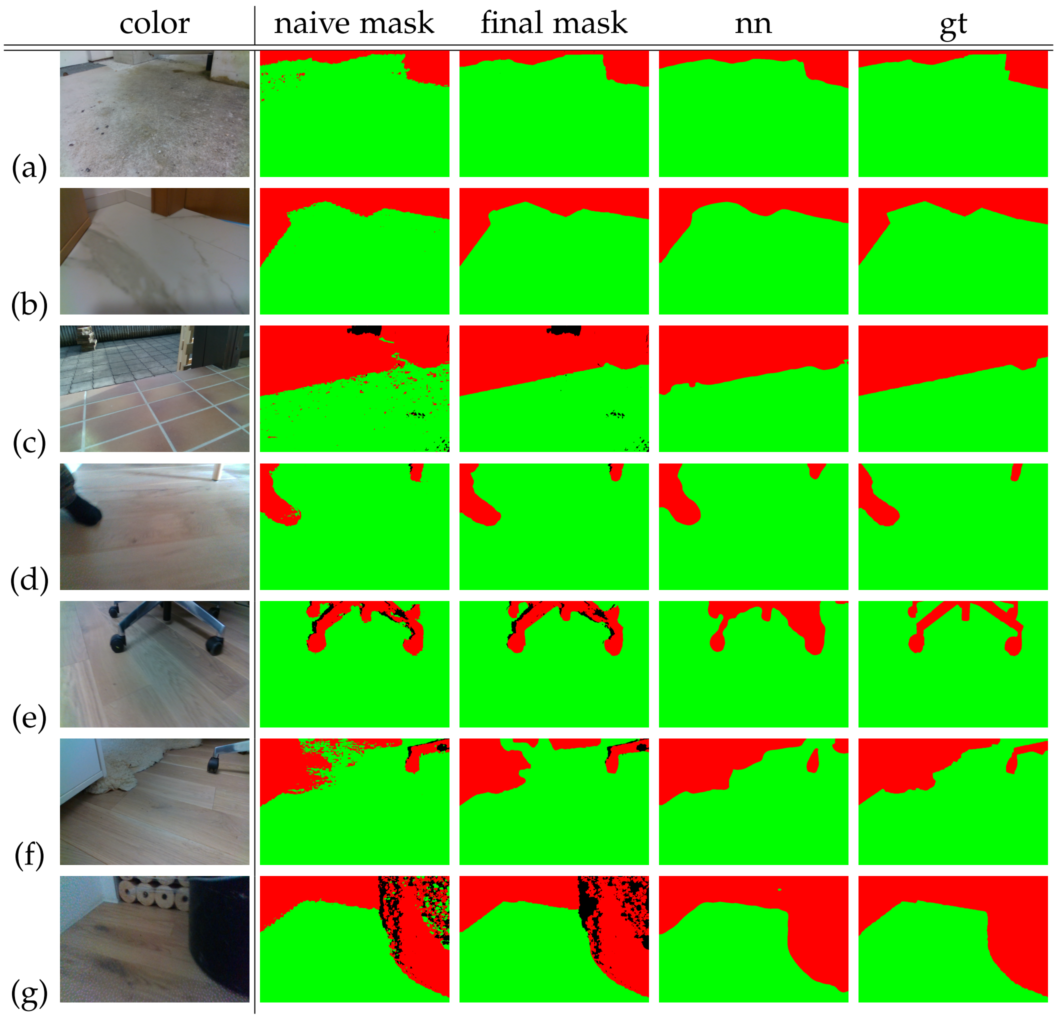


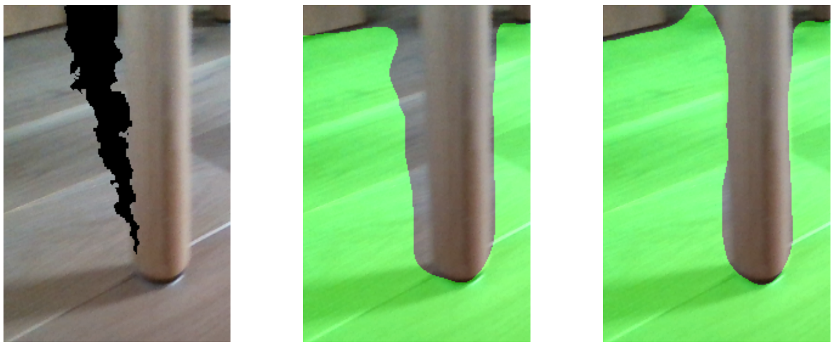
Figure 6.1: Segmentation masks of differents steps

### 6.2.1 Shadows with no stereo data

The color camera in the IntelReal Sense D435 is located on the left side of the assembly. Parts of the image that the camera sees may be occluded from one of the two infrared cameras. These parts then have no depth information in the color image and must be set to the ignore mask. During training the output of the network at those pixels does not add to the loss, thus the network does not learn anything from these pixels.

A problem was found in the indoor dataset, where corners and chair legs would always have a shadow to their left. Figure 6.2 (a) shows such an example from the test set. There is no training data available for how the left most pixels of a chair leg, and any adjacent pixels should be classified, thus the network performed poorly in these areas (see Figure 6.2 (b)). By adding flipped versions of the training data to the training set, this problem was mitigated, as the chair legs in this case look identical in mirrored view but the mask for the then-left side will be better. This improved the network output further (see Figure 6.2 (c)).

Mirroring training data may not be suitable for all datasets or environments. Thus it is better to create two recordings, where in one recording the camera is mounted upside down. In this case all occlusions will be on the opposite side as the color camera is not centered in the camera. The final training data will then have labels available for either case. A special mode was added to the processing pipeline to process upside down images correctly (see section 5.2.2).



(a) color image with the occluded area marked black

(b) network output after training

(c) network output after training with mirrored data

Figure 6.2: Occluded areas in color image and network output before and after training with flipped images.

### 6.2.2 Observations for the CRF refinement

The following examples show instances where the CRF refinement improves the mask quality and where it does not work well.

#### Walls at ground height

A common error appears if a wall somewhere behind an obstacle is on the same height as the ground model. Figure 6.3 (a) shows an example. The wall behind the ledge and the wooden palette are classified as a wall initially (Figure 6.3 (b)). In an ideal point cloud those points would have been rejected by the normal vector constraint described in section 3.3, but the inaccuracy from stereo matching creates a bumpy surface where some points's normal vectors manage to fall within the threshold. The CRF refinement step corrected the error (Figure 6.3 (c)).

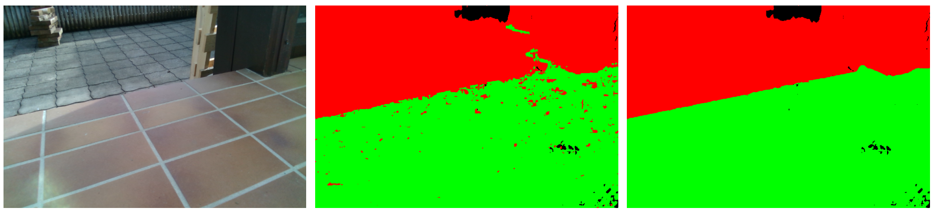

(a) color image (b) initial mask (c) mask after crf

Figure 6.3: Walls at ground level introduce errors

#### Noisy ground planes

It was observed that some ground planes were initially segmented very noisy. Figure 6.4 (a) shows a tiled tiled floor. The small grooves in between the tiles create noisy normal vectors. In addition the smooth surface does not offer a good texture for stereo matching and the infrared projector is not visible in sunlight. Because of that, some of the points of the actual ground plane are classified as an obstacle. Since the majority of the points is classified as ground, the CRF refinement manages to correct for this noise (see Figure 6.4 (b)).

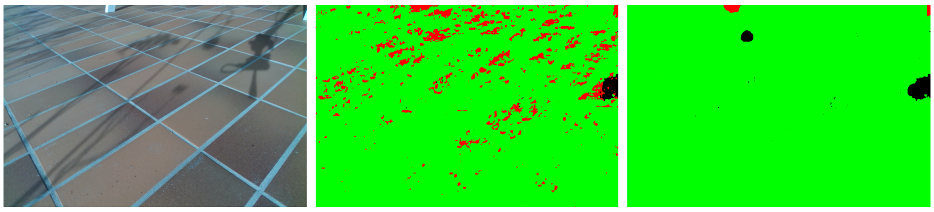

(a) color image (b) initial mask (c) mask after crf

Figure 6.4: Noisy ground points for outdoors tiled floor

### Using CRF to infill missing data

As described in section 3.4, the CRF result is not used to infill areas with no or bad depth information. It was tested if infilling unknown areas improves or decreases overall quality. The option to infill was evaluated on the testset. The generated masks with infill increased the pixel density to 100% but reduced the overall mean IoU to 87.5% and thus this technique was not used.

### Bad CRF refinement in low contrast images

The CRF refinement does not improve the image mask quality in all conditions. It was observed that for low contrast or blurry images the CRF did not significally improve the final mask quality. Yet, no instance was found in the testset where the CRF refinement has decreased the IoU score.

## 6.3 Evaluating the practical differences

The generated point cloud from our method is compared against the already existing on-board sensors of the robot. Practical differences and drawbacks between our generated point clouds and the output from the on-board LIDAR range scanner are shown. For this test a dataset was generated for the robot in an outdoor environment, and a subsequent recording was used for the comparison.

Figure 6.5 shows an example of the robot's vision, and the approximate field of view of the camera. The green and blue dot points are the output of the proposed method and the LIDAR sensor.

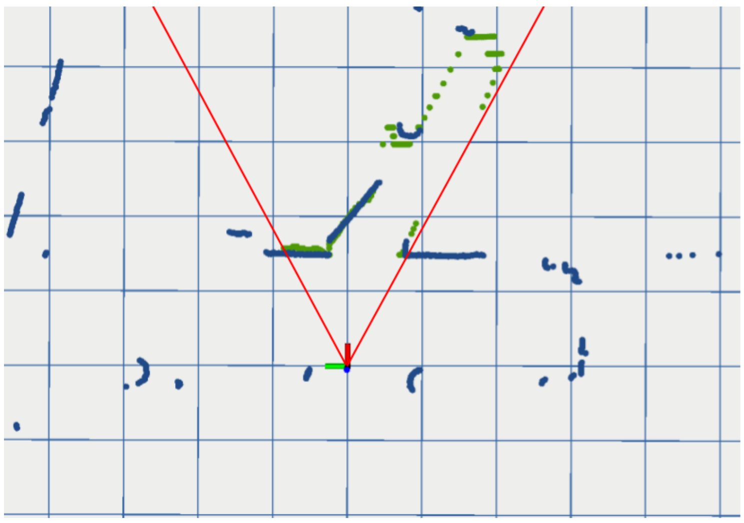


Figure 6.5: Field of view comparison between the LIDAR sensors and our method. green points: our method, blue: LIDAR sensor, red: field of view of the camera.

### 6.3.1 Same height surfaces

The used LIDAR sensor can only scan the surroundings in one axis, thus it cannot see any downward ledges or other objects below the height it is mounted at. It also does not record any color information. This is one advantage of our method, as the neural network can decide an area is an obstacle based on the visual appearance. Figure 6.6 shows an example where the neural network can segment between the paved surface and the grass area, although both surfaces would appear flat and on the same height in a full 3d scan.

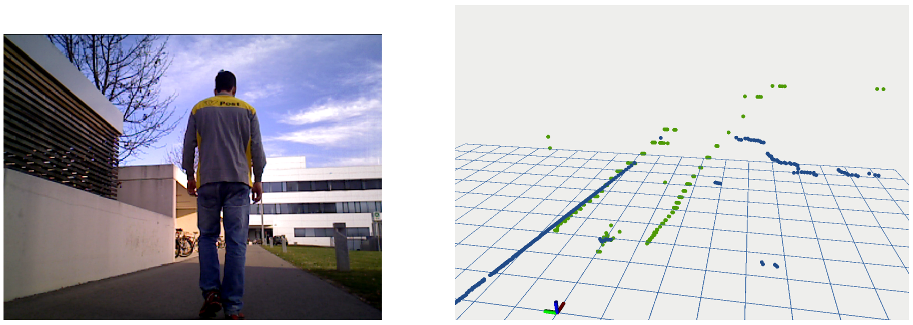

(a) color image (b) point clouds

Figure 6.6: Example where the neural network can segment same-height surfaces (pavement and grass) based on their appearance. green points: our method, blue: LIDAR sensor.

### 6.3.2 Behaviour during tilting

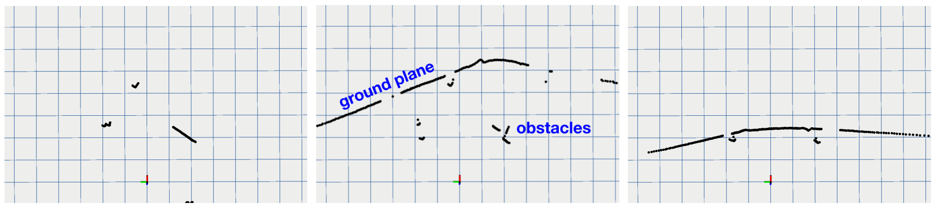
When the robot is braking or accelerating it is tilting slightly due to it's inertia. Both the 2d laser scanner and our method for generating the pointcoud points assume the robot's body is parallel to the ground. When the robot tilts, this assumption no longer holds true.

#### Effects on LIDAR range sensor

For the laser scanner points of vertical objects stay approximately the same distance, and points of the ground plane move closer to the robot. The robot only tilts significantly during hard breaking for either avoiding obstacles or emergency braking. In this case, if an object appears to be closer, it would only encourage more braking. Figure 6.7 shows a sequence of images during braking and in resting position after braking.

#### Effects on our method

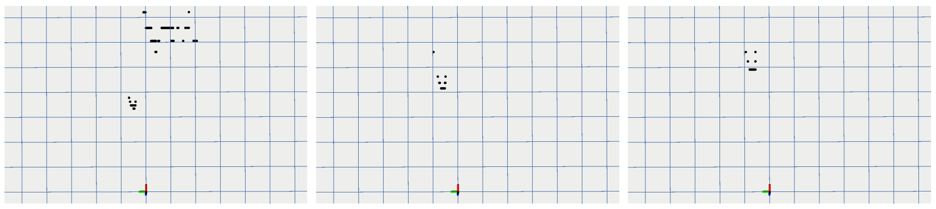
During braking the camera is tilting downwards. This makes obstacles on the ground appear higher in the image. Thus, their points are projected



(a) resting position (b) slight tilt during braking (c) significant tilt

Figure 6.7: LIDAR points during braking. The figure shows how points of the ground are moving closer during a hard brake. In a (b) and (c) the robot's tilt increases. (a) shows the robot in resting position after braking. Points of obstacles are not affected significantly.

further away. This is because from the camera's perspective higher points in the image are further away on the ground plane. The output of our method can be seen in figure 6.8 for the same breaking operation as shown in figure 6.7. The behaviour is inverse the behaviour of the LIDAR sensor. Analog to this, during acceleration objects appear to move closer.



(a) resting position (b) slight tilt during braking (c) significant tilt

Figure 6.8: Projected point cloud points during braking. The figure shows how points of obstacles are moving further away during a hard brake. In a (b) and (c) the robot's tilt increases. (a) shows the robot in resting position after braking.

In worst case scenarios objects during breaking appear further away and would cause the breaking to stop, during subsequent acceleration objects out of range would appear move closer to the robot when it was accelerating and order a immediate brake. This could cause an oscillation where the robot would start and stop repeatedly due to this effect.

# 7 Conclusion and Future work

In this thesis a framework was presented for generating semantic segmentation training data for segmenting flat ground surfaces. It was shown how the output of a trained neural network can be used to provide, or assist a robot with, collision avoidance. We showed that the quality of the automatic data generation is good and well suited for training a neural network. The neural network was able to learn the required task from the generated training data and the resulting 3d point cloud can be used for avoiding collisions.

The generated point cloud will contain errors as soon as the robot is tilting against our assumed position parallel to flat ground. These errors are hard to detect within the system we have constrained ourselves in, but are also present when using 2d LIDAR ranging sensors as they make similar assumptions. Calculating tilt from a inertial measurement unit will only work as long as the robot is always driving on level surfaces and may not be fit for outdoor environments. This could although be further improved by training or using a neural network for estimating the depth from a monocular color image.

The method used for segmenting ground points in 3d point clouds is purposefully kept simple, in order to avoid any human interaction or correction in the pipeline. Also there are no requirements regarding visual or spatial coherence between the recorded frames. Each frame is processed individually. While dense 3d reconstruction over multiple frames may provide better results in terms of accuracy and pixel density, the proposed method avoids any errors or failures which may occur during registration of frames.

The linear model which is assumed for the ground plane works well for indoor or urban environments but fails for curved surfaces or for example in front of ramps. The method could be improved by giving the reviewer a choice as to which surfaces should be included in the final output map, if multiple possible ground surfaces were detected in the image. In this case the user can explicitly annotate the class of each segmented object in the point cloud. This can increase the number of classes in the dataset, while still providing a fast way of labeling. Furthermore different approaches for segmenting the ground plane which do not rely on an assumed model may provide better results for certain environments.

# Bibliography

Beyer, Lucas. *Github: PyDenseCRF*. URL: https://github.com/lucasb-eyer/pydensecrf (cit. on p. 27).

Bokovoy, A., K. Muravyev, and K. Yakovlev (2019). “Real-time Vision-based Depth Reconstruction with NVidia Jetson.” In: *2019 European Conference on Mobile Robots (ECMR)*, pp. 1–6. DOI: 10.1109/ECMR.2019.8870936 (cit. on p. 8).

Bradski, G. (2000). “The OpenCV Library.” In: *Dr. Dobb’s Journal of Software Tools* (cit. on p. 27).

Buchegger, Alexander et al. (2018). “An Autonomous Vehicle for Parcel Delivery in Urban Areas.” In: pp. 2961–2967. DOI: 10.1109/ITSC.2018.8569339 (cit. on p. 1).

Cordts, Marius et al. (2016). “The Cityscapes Dataset for Semantic Urban Scene Understanding.” In: *CoRR* abs/1604.01685. arXiv: 1604.01685. URL: http://arxiv.org/abs/1604.01685 (cit. on p. 9).

Dias, Philipe et al. (2019). “FreeLabel: A Publicly Available Annotation Tool Based on Freehand Traces.” In: pp. 21–30. DOI: 10.1109/WACV.2019.00010 (cit. on p. 2).

Fischler, Martin A. and Robert C. Bolles (1981). “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography.” In: *Commun. ACM* 24, pp. 381–395 (cit. on p. 11).

Hirschmüller, Heiko (2005). “Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information.” In: vol. 2, pp. 807–814. ISBN: 0-76952372-2. DOI: 10.1109/CVPR.2005.56 (cit. on p. 10).

Jäger, Jonas et al. (2019). *LOST: A flexible framework for semi-automatic image annotation*. arXiv: 1910.07486 [cs.CV] (cit. on p. 2).

Jia, Yangqing et al. (2014). “Caffe: Convolutional Architecture for Fast Feature Embedding.” In: *arXiv preprint arXiv:1408.5093* (cit. on p. 5).

Krähenbühl, Philipp and Vladlen Koltun (2012). “Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials.” In: *CoRR* abs/1210.5644. arXiv: 1210.5644. URL: http://arxiv.org/abs/1210.5644 (cit. on p. 27).

Lafferty, John, Andrew Mccallum, and Fernando Pereira (2001). “Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data.” In: pp. 282–289 (cit. on p. 15).

Pech-Pacheco, J. L. et al. (2000). “Diatom autofocusing in brightfield microscopy: a comparative study.” In: *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*. Vol. 3, 314–317 vol.3. DOI: 10.1109/ICPR.2000.903548 (cit. on p. 10).

Pérez, Luis et al. (2016). “Robot Guidance Using Machine Vision Techniques in Industrial Environments: A Comparative Review.” In: *Sensors* 16, p. 335. DOI: 10.3390/s16030335 (cit. on p. 1).

Piemngam, K., I. Nilkhamhang, and P. Bunnun (2019). “Development of Autonomous Mobile Robot Platform with Mecanum Wheels.” In: *2019 First International Symposium on Instrumentation, Control, Artificial Intelligence, and Robotics (ICA-SYMP)*, pp. 90–93. DOI: 10.1109/ICA-SYMP.2019.8646085 (cit. on p. 8).

Reitter. *Github Pages: Groundsegmentation*. URL: https://rettier.github.io/groundsegmentation/ (cit. on p. 26).

Russell, Bryan et al. (2008). “LabelMe: A Database and Web-Based Tool for Image Annotation.” In: *International Journal of Computer Vision* 77. DOI: 10.1007/s11263-007-0090-8 (cit. on p. 2).

Rusu, Radu Bogdan and Steve Cousins (2011). “3D is here: Point Cloud Library (PCL).” In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China (cit. on p. 27).

Stanford Artificial Intelligence Laboratory et al. (2018). *Robotic Operating System*. Version ROS Melodic Morenia. URL: https://www.ros.org (cit. on p. 6).

Tang, Jie, Yong Ren, and Shaoshan Liu (2017). “Real-Time Robot Localization, Vision, and Speech Recognition on Nvidia Jetson TX1.” In: (cit. on p. 8).

Yu, Bengong and Zhaodi Fan (2019). “A comprehensive review of conditional random fields: variants, hybrids and applications.” In: *Artificial Intelligence Review*, pp. 1–45. DOI: 10.1007/s10462-019-09793-6 (cit. on p. 15).

Zhao, Hengshuang, Xiaojuan Qi, et al. (2017). “ICNet for Real-Time Semantic Segmentation on High-Resolution Images.” In: *CoRR* abs/1704.08545.

arXiv: 1704.08545. URL: http://arxiv.org/abs/1704.08545 (cit. on pp. 3, 5).

Zhao, Hengshuang, Jianping Shi, et al. (2017). “Pyramid Scene Parsing Network.” In: *CVPR* (cit. on p. 5).