Masaryk University
Faculty of Informatics

Graphics Processing of LiDAR Data

Master's Thesis

Jaroslav Řehořka

Brno, Fall 2017

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jaroslav Řehořka

Advisor: prof. Ing. Václav Přenosil, CSc.

Acknowledgement

I would like to thank my advisor prof. Václav Přenosil, for his help and willingness during the elaboration of this thesis. I would also like to thank my consultant Jan Popelka from Honeywell for his help, advice, and ideas. My thanks also go to company Honeywell for provided hardware.

Last but not least I would like to express my gratitude to my friends and family for all their support during the elaboration of this thesis and during my studies. This includes my friend, Roman Stoklasa for his ideas and advice regarding algorithms and thesis content, and my girlfriend for her insights, patience, and support.

Abstract

This thesis analyses possibility of use of *LiDAR* sensor for purposes of objects recognition in obtained data. Specifically with the problem of real-time line detection in 2D data, together with a recognition of flat area with specified dimension in 3D data.

For 2D data, we suggest a recursive algorithm, that segments colinear points in groups. Groups are subsequently analyzed for the magnitude change of angle for each point. If the change is more significant than a specified threshold, the group is not considered to a valid line. We have tested this algorithm on more than 230 scans, with various parameter settings. Results were evaluated with respect to a manually specified ground truth. The testing has shown that the algorithm has good sensitivity and speed for use in real-time applications.

Regarding the 3D data and flat area recognition, coplanar points are converted into a binary image, which is further processed with mathematical morphology operator – opening. We have tested this approach on more than 30 3D surfaces. Results were also evaluated manually and testing have shown, that the suggested approach has a good accuracy and might be suitable for real applications.

Keywords

LiDAR, line detection, surface detection, UAV, Raspberry, RPLidar

Contents

1 Introduction

In past years, we are witnesses of the massive spread of unmanned vehicles, whether in the form of ground (UGV) or aerial (UAV) vehicles. Such devices have a wide range of utilization, and they are being used or build by amateurs or professionals for various tasks. Even governmental organizations like a police or firefighters are using this kind of vehicles for rescue, monitoring, and other, similar purposes. To make the device control easier and user-friendly, the device usually does not contain only guidance controls, but many other either software or hardware improvements, that lead to more significant device autonomy (e.g., flight stabilization). Specifically, the hardware improvements are implemented through various sensors.

One of these hardware sensors is a *LiDAR* (an acronym for *Light Detection And Ranging*), which is basically laser distance meter. The *LiDAR* can be used as an altimeter or as a sensor for either *2D* or *3D* space perception. An output of *LiDAR* sensor is a cloud of points, whose processing and analysis is the primary goal of this thesis.

We have defined three goals in this thesis. The first goal is to analyze available *LiDAR* hardware and software solutions for further processing of point cloud data. The second goal is to work with a *LiDAR* device and create real-time analyzer of obtained data from such device. The data analysis contains an algorithm for a detection of a simple object. Finally, the third goal is to process *3D* data, in order to detect a flat area with specified dimensions, that could be used, e.g., for detection of a suitable landing area.

This thesis consists of 8 chapters. In Chapter 2 we introduce important terms and names regarding the *LiDAR* devices, whereas Chapter 3 analyzes existing *LiDAR* hardware and software solutions. In Chapter 4, we analyze *2D* data and propose the way of detecting a line in such data. Chapter 5 is focused on *3D* data processing together with plane analysis and landing spot finding. In Chapter 6, we describe a way of implementation of a *2D* data processing proposed in Chapter 4. The results of implementation testing are presented in Chapter 7, and conclusion is stated in Chapter 8.

2 LiDAR Principle and Data Representation

LiDAR is an optical device for detecting the presence of objects, specifying their position and gauging distance.

It works on a similar principle as radar; however, it's sensor uses discrete pulses of laser light instead of radio waves. In comparison to radar, *LiDAR* is a relatively new technology with initial commercial applications in 1995 [1]. Together with radar, *LiDAR* is a typical representative of active remote sensing technology, which means that it emits a signal towards the target and analyzes the backscattered signal.

The main difference lies in a signal type and beam divergence. While radar emits a wide beam of microwave energy and thus signal usually seize multiple objects, *LiDAR's* uses laser light beam that can target precisely even on tiny objects. This is illustrated in figure 2.1. On the other hand, unlike radar, *LiDAR* cannot penetrate clouds, rain, or dense haze [2].

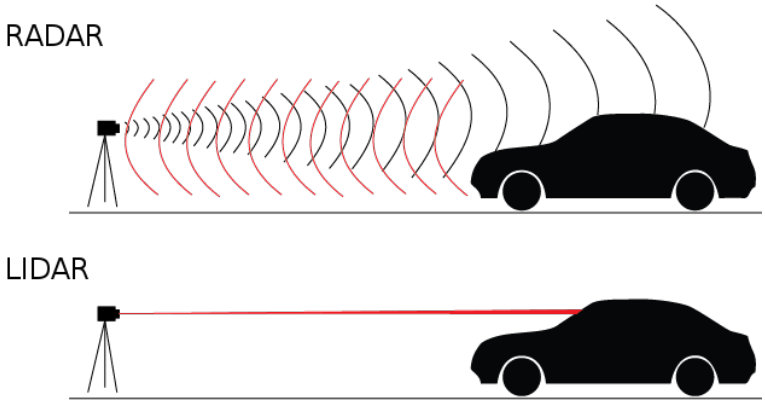


Figure 2.1: The difference of *LiDAR* and radar in size of target's surface hit.

LiDAR has a wide variety of applications such as mapping, autonomous vehicles, cartography, visualization, navigation, meteorology, etc.

2.1 HW Principle LiDAR

2.1.1 Components

Principal components of *LiDAR* device are a transmitter that emits laser signal, and a receiver, that receives returned energy. Both transmitter and receiver can be connected to mirror or multiple mirrors if the construction of

the device requires so. Apart from that, non-static *LiDAR* systems also contain a drive motor which rotates mirrors at a specified frequency. A simple scheme of a *LiDAR* device is displayed in figure 2.2.

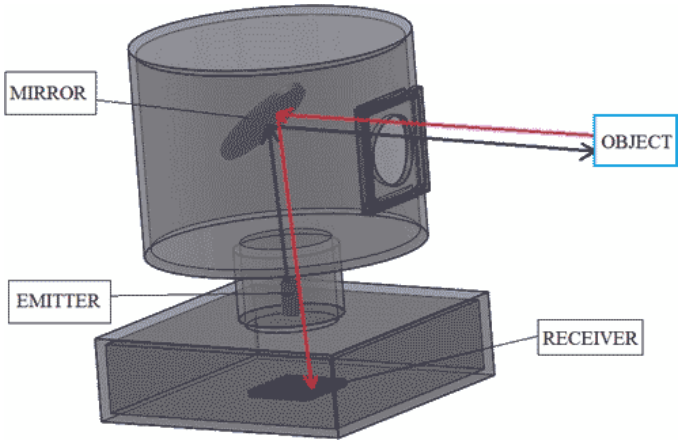


Figure 2.2: Scheme of a *LiDAR* device with a single mirror. Image adapted from [3].

The transmitter consists of a laser source and an electronic component that controls timing and continuance of laser emission. Most imaging lidars use the following wavelength:

- **infrared (1550 nm)** high power, long range systems and military applications [4]
- **near-infrared (1000 - 1500 nm)** for terrestrial mapping, easily absorbed by water [5]
- **blue-green (532 nm)** for water bodies penetration [5]

To help with precise determination of final scans' location, the *LiDAR* can be accompanied by other components. Such components include *GPS* for precise location of the sensor, *Inertial measurement units (IMUs)* used to precisely measure and record the orientation of an aircraft, and high precision clock for accurate timing [6].

2.1.2 Detection Schemes

Early *LiDAR* systems were capable of recording only one discrete return per one pulse (either first or final peak of the reflected wave). By the year 2000 commercial systems were capable of recording 3-5 returns per one pulse. Multiple returns can be analyzed and used for getting information about the scanned environment, which can be divided into separate layers (e.g.,

separation of ground and objects). The illustration of multiple layers scan can be seen in figures 2.3 and 2.4. Such systems principally work by measuring peaks of an amplitude of a returned signal. They are called **incoherent** or **discrete**.

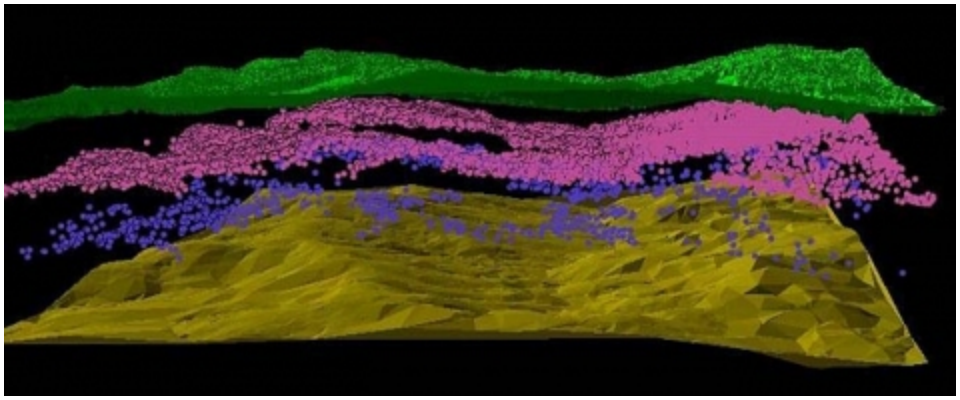


Figure 2.3: Visualization of multiple *LiDAR* returns together with a post-processed bare ground layer. Image adapted from [7].

Apart from discrete return *LiDARs*, **full waveform** (or **coherent**) *LiDAR* systems exist. The difference lays in a way how the returned pulses are recorded. Unlike incoherent systems, coherent *LiDARs* can record the entire waveform of returning pulse and thus provide more accurate data. The main drawback of coherent systems is an increased size of the incoming data, and therefore more computing resources are required. Full waveform *LiDARs* are used in a field of vegetation analysis and related disciplines [7].

2.1.3 Pulse Models

In both coherent and incoherent *LiDAR* systems, there are two types of pulse models: micropulse and high energy systems.

Micropulse *LiDAR* systems have developed as a result of increasing computer power and further development of laser technology. They use less energy than high energy systems in laser and are often eye-safe (they can be used without safety precaution). High power systems are common in atmospheric research, where they are used for measuring atmospheric parameters [9].

All *LiDAR* devices compared in Table A.1 belong to micropulse device family.

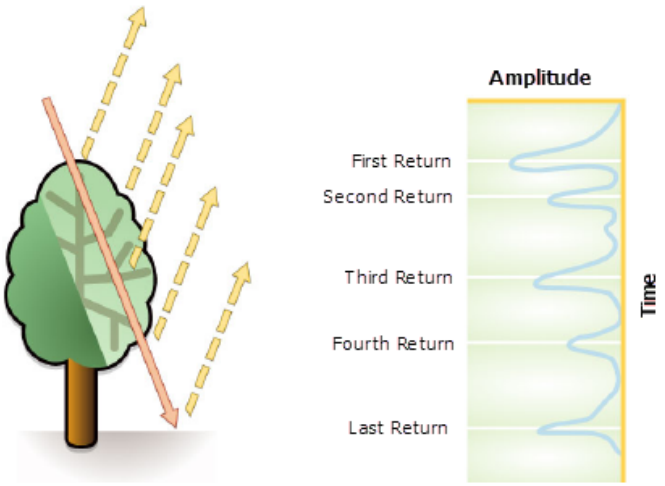


Figure 2.4: Sketch of multiple returns from one *LiDAR* pulse, from [8]. The laser beam can permeate the tree down to the ground while part of the beam is reflected back in different levels to the *LiDAR*.

2.1.4 Sensor Attributes

LiDAR sensors have several attributes, which values may differ from device to device depending on price and field of device use. These attributes are measurement range, accuracy, field-of-view, angular resolution, and frequency.

Measurement Range

Measurement range stands for the minimal and maximal distance of an object that can be detected by a *LiDAR* device. This parameter depends on weather conditions as well as on the type of target's surface. Surface with better reflectance extends maximal range, whereas black, steep, smooth, wet surfaces or their combination can decrease maximum range drastically, weather conditions affect reflection properties (for example a wetness can change reflectance from Lambertian to specular) [10]. Both types or reflectance are illustrated in figure 2.5.

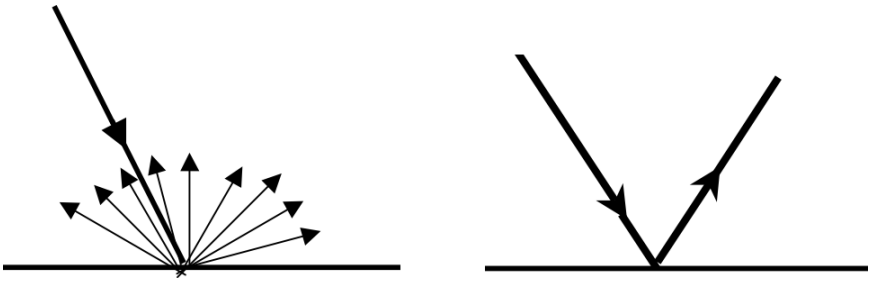


Figure 2.5: Lambertian (also called diffuse) and specular (right side of the image) type of reflection. Specular reflection behaves similarly to mirror, while Lambertian reflection is an ideal diffuse reflection, which reflects the signal to many angles. Image adapted from [10], edited.

Accuracy

The accuracy of a *LiDAR* device means, what is the difference between measured values compared to real values. Accuracy is calculated according to *National Standard for Spatial Data Accuracy* with the following formula [11]:

$$Accuracy_{(z)} = 1.96 * RMSE_{(z)} \tag{2.1}$$

Where 1.96 is 95% confidence interval, and RMSE (Root-Mean-Square Error) is calculated as the square root of the average of the set of squared difference between the target and the *LiDAR* surface [12].

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \hat{x}_i)^2}{n}} \tag{2.2}$$

Field of View

The field-of-view (FOV) is defined as the angle which is covered by *LiDAR* sensor. For some *LiDAR* sensors, the number of signals emitted per scan cycle is always the same, in such cases lowering the FOV will result in a higher density of point and conversely for increasing of FOV [13].

While all 2D *LiDARs* have either horizontal or vertical FOV equal to zero, because in one of these directions there is only one line of laser points, 3D *LiDAR* systems have both horizontal and vertical FOV higher than zero.

Angular Resolution

Angular resolution is a minimal angular distance at which two equal targets can be distinguished from each other when at the same distance. This resolution can be calculated by an application of *Rayleigh Criterion*. Angular resolution (radians) Θ is

$$\Theta = 1.22\frac{\lambda}{D} \tag{2.3}$$

where λ is a wavelength of the light and D diameter of *LiDAR* lens (if the lens is focusing a beam of light with finite extent, the value D is corresponding to the diameter of the light beam not lens) [14, 15]. The factor 1.22 is derived from a calculation of the position of the first dark circular ring surrounding the central Airy disc of the diffraction pattern [16, p. 469].

Some *LiDAR* devices do not have laser beam with finite extent. The main reason is an usage of a diffuser so that *LiDAR* is eye-safe [17]. Because of this fact, the laser beam is getting wider and angular resolution greater with increasing range.

Frequency

Frequency is a number of repeating scans (*LiDAR* device rotations) per time unit, typically expressed in Hertz.

2.1.5 Point Cloud

The point cloud is a typical output from a *LiDAR* device. It is a set of points in a coordinate system. For 2D *LiDARs*, the set of points consists of points with *x* a *y* coordinates, while 3D devices produce points with extra *z* coordinate.

Common file format for storing *LiDAR* data is called *LAS*, and it is being managed by *American Society of Photogrammetry and Remote Sensing (ASPRS)*. In this file, information about every point in the point cloud is stored. Among the *x*, *y* and *z* positions many others information about the point and signal are stored. Such information includes intensity, return number, the number of returns and others [18]. The complete list of attributes is shown in tables 2.1 and 2.2. Sample visualization of both *2D* and *3D* point cloud is presented in figure 2.6.

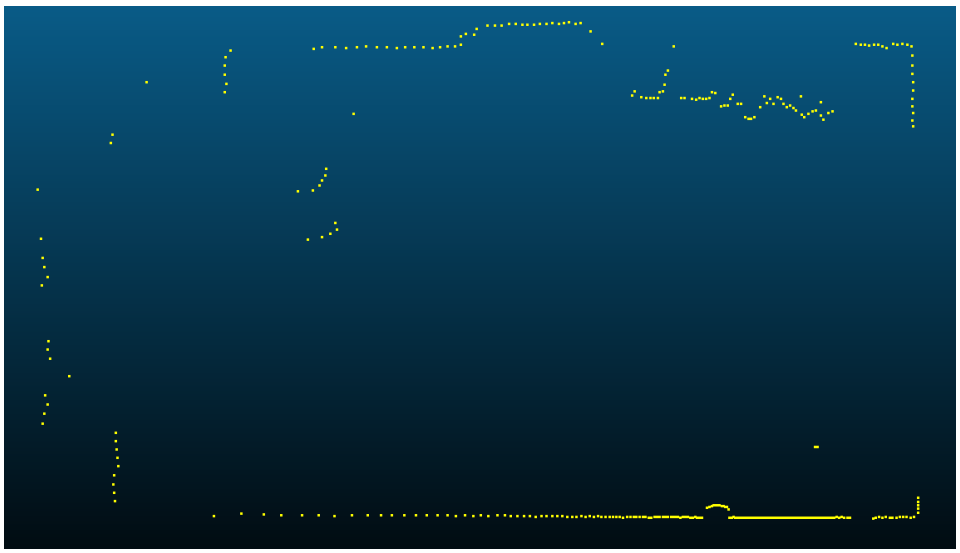
Several others file formats for point cloud storing exists. Such as a *BPF (Binary Point File)* defined by *National Geospatial-Intelligence Agency* [19] or *NITF (National Imagery Transmission Format)* that contains *LAS* file inside and is specified by *US Department of Defense*. It is also possible to store point clouds directly into database systems like *Oracle*, *PostgreSQL* or *SQLite*.

Attribute Name	Description
X	*X* coordinate of the point
Y	*Y* coordinate of the point
Z	*Z* coordinate of the point
Intensity	Pulse magnitude
Return Number	Return number for a given output pulse
Number of Returns	Total number of returns for a given pulse
Scan Direction Flag	The direction at which the scanner the mirror was traveling
Edge of Flight Line	Has a value of 1 only when the point is at the end of a scan
Classification	Point type classification, listing is available in table 2.2
Scan Angle Rank	An angle at which the laser point was the output from the laser system
User Data	User's data
Point Source ID	File from which this point originated
GPS Time	Time tag at which the point was acquired
Red	The Red image channel value
Green	The Green image channel value
Blue	The Blue image channel value

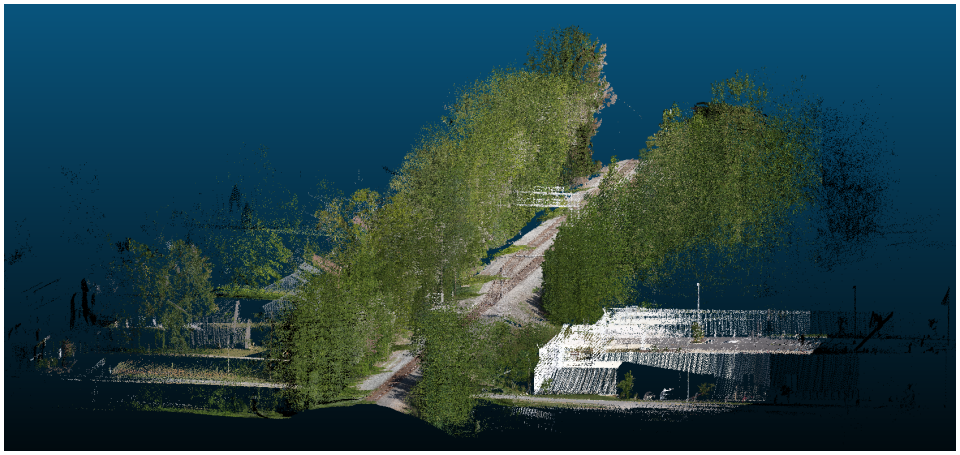
Table 2.1: List of point's attributes in LAS *LiDAR* file [18].

Classification Id	Description
0	Created, never classified
1	Unclassified
2	Ground
3	Low Vegetation
4	Medium Vegetation
5	High Vegetation
6	Building
7	Low Point (noise)
8	Model Key-point (mass point)
9	Water
10	Reserved for future definition
11	Reserved for future definition
12	Overlap Points (culled during the merging of overlapping flight lines)
13-31	Reserved for future definition

Table 2.2: LAS file point classification types [18].



(a) 2D point cloud



(b) 3D point cloud

Figure 2.6: Visualization of *LiDAR* point cloud. 2D in figure 2.6a and 3D point cloud in figure 2.6b.

3 Platforms

This chapter describes available hardware devices and software which might be suitable for a creation of setup for gathering and analyzing a point cloud.

3.1 HW Platforms

3.1.1 Lidar Sensors

In this subsection, we will go through the subset of lidar producers and their solutions. Manufacturers and their devices were selected according to their suitability for use in UAV applications. Suitability was assessed with regard to weight and size.

- Company *Slamtec* focuses on robot localization and mapping solutions. Their only lidar is a low-cost *RPLiDAR A2*. This lidar is a 2D, 360° scanner [M1]. We are using *RPLiDAR A2* in the implementation part of this thesis.
- *Scanse's* lidar called *Sweep* is a crowdfunded low-cost lidar sensor, which offers slightly better hardware than *RPLiDAR A2* [M2].
- *Velodyne LiDAR* offers several solutions for distance sensing. Their focus is mainly on a field of autonomous vehicles. Lidars made by *Velodyne* are being used by companies like *Google, Ford, TomTom or Apple*.

  From heavyweight *HDL-64E* [M3], which is suitable mostly for terrestrial use, to lightweight *HDL-32E* [M4] or *VLP-16 (Puck)* [M5]. All of the available products scan their surroundings both horizontally and vertically with the point cloud as an output.
- *Ocular Robotics* designs and produces lidar solutions with high precision and range, which are suitable for ground, air or sea use.

  Ocular offers two lidar devices, both for professional use with 360° horizontal and 70° vertical scanning. Smaller *RE05* [M6] and bigger and more powerful *RE08* [M7].
- *Quanergy* offers small-sized lidars, solid state or rotational. The most powerful is a 3D lidar *M8* [M8], which can scan its surroundings 360° horizontally and 20° vertically.

- *Hokuyo* develops lidar sensors for both industrial and robotic use. Devices *UTM-30LX* [M9], *URG-04LX* [M10], *URG-04LX-UG01* [M11], *UST-05LA* [M12], *UST-10LX* [M13] and *UST-20LX* [M14] are designed to be used in robotic and UAV applications.

- Company *Riegl* provides a wide variety of lidar solutions like terrestrial, unmanned, mobile, industrial or airborne scanning. Suitable devices for UAV use are *VUX-1UAV* [M15], *miniVUX-1UAV* [M16], *VUX-1LR* [M17] and *VUX-1HA* [M18].

All *LiDAR* devices mentioned above are compared by their parameters in appendix A, table A.1.

3.1.2 Lidar Boards

Real-time processing and analyzing of the data produced by a *LiDAR* sensor requires significant computing power. Because of that, choice of a suitable motherboard is a crucial task for building a complex *LiDAR* system. Essential parameters of such motherboard are computing power, energy requirements, size, communication interfaces, and price.

All of these parameters depend on required data analysis and size of whole system.

In the chapter 6, we will work with *Raspberry Pi3*, which is credit-card sized computer with quad-core ARM CPU, 1GB RAM, and USB powered (5 V). Communication peripherals are 4x USB 2.0, 40x GPIO, 100Mbps LAN and HDMI [M19].

At the moment, many raspberry-like computers from various producers exist. They offer more or less different hardware setup and price. We compare a selection from available boards in appendix A, table A.2.

3.2 SW Platforms

In this subsection, we will analyze existing programming libraries and applications that work with *LiDAR* and *LiDAR* output. From applications, we will consider only such applications, which can detect objects in *LiDAR* output data.

3.2.1 Libraries for LiDAR Data Processing

Several libraries are able to process *LiDAR* output data. These libraries provide programming API for processing of *LAS* files and handling other *LiDAR* output data.

LASzip

LASzip is a compression library for *LAS* format file data. This library is completely lossless and can compress *LAS* files into 7-20% of original size.

LASzip is provided as an *LGPL* licensed library.

libLAS

libLas is a C/C++ library for reading, writing *LAS* files and command line toolset for inspecting, manipulating, transforming and processing such *LAS LiDAR* data.

libLas has the ability to filter points according to specified rules, classify point according to *ASPRS* object classification, filter such classifications and assign colors to them.

libLas is available under *BSD* license.

LASlib

LASlib is a C++ programming API and provides same functionality as *libLAS* library, however, it can work with various coordinate systems and is able to convert them among each other.

This library is available under *LGPL* license.

PCL – Point Cloud Library

PCL a C++ open source library for 2D/3D image and point cloud processing. The library contains algorithms for filtering, feature estimation, surface reconstruction, model fitting, segmentation, and visualization [20]. *PCL* can also work with real-time data (e.g., object tracking).

PCL has been released under *BSD* license.

Laspy

Laspy is a python library for creating, reading and manipulating *LAS* files. *Laspy* also provides a few command line tools for *LAS* files comparison, validation, and conversion.

Laspy is distributed under a custom license that allows both personal a commercial use in a certain condition.

3.2.2 Applications and Tools

Among *LiDAR* data processing libraries and frameworks, some toolsets and end-user applications that operate with a *LiDAR* data exists.

LASTools

LASTools is a collection of command line tools that work with *LAS* and *LAZ* files.

LASTools contains tools for visualization, ground extraction, building and high vegetation classification, *LAS* data rasterization, points colorization, multiple *LAS* files overlap computation, boundary detection, object clipping, object height estimation, tree canopy density and cover computation, *LAS* and *LAZ* files processing (removing duplicates, cloning such files, displaying file info, file splitting).

This toolset is not open source, nor free and price starts at 2000 EUR.

PDAL – Point Data Abstraction Library

PDAL is an open source, command line tool library for translating and processing point cloud data.

This library provides multiple tools for nearest point calculation, contextual difference computation, ground vs. object segmentation, calculation of Hausdorff distance between two point clouds, input file information (format, a number of points, coordinate system, summary data statistics), merging multiple files into one, random point cloud creation, points sorting, large files splitting, and file formats conversion.

It is divided into small, programs that communicate according to *JSON* or *XML* defined pipeline. Thanks to this pipelined system, various input and output file types are supported.

Pipelined input data can be removed, modified or reorganized with filters. *PDAL* provides more than 40 filters such as cropping, chipping large

point clouds into smaller, colorization, range computation, sorting, and others. *PDAL* is open source, licensed under *BSD* license. Example of the pipeline definition can be seen in figure 3.1.

```
{
  "pipeline":[
    "input.las",
    {
      "type":"crop",
      "bounds":"([0,100],[0,100])"
    },
    "output.bpf"
  ]
}
```

Figure 3.1: Example of pipeline specification. Input file *input.las* is processed with *crop* filter, that removes all points outside of specified bounding box. All filtered points are saved into *output.bpf* file in *BPF* file format.

CloudCompare

CloudCompare is a 3D point cloud processing and visualization software. It supports various point cloud operations such as comparison of multiple point clouds, registration, resampling, color/normal/scalar fields handling, statistics computation, sensor management, interactive or automatic segmentation, display enhancement, etc. It supports various point cloud files and extension plugins [21].

It is multiplatform open source software, licensed under GPL license.

ParaView

ParaView is an open-source, multi-platform data analysis and visualization application. This application can extend its functionality by loading external plugins like as *PCL plugin*, which allows using *PCL* library filters and algorithms [22]. These filters and algorithms can be applied repeatedly in a specified order on the incoming data. A sample visualization of applied filters is visible in figure 3.2.

Figure 3.2: Filters application in *Paraview*. Segmentation, threshold filtering, cluster extraction, and object bounding box. Images are taken from [22].

Matlab

Matlab supports visualization and processing of points cloud in either *2D* and *3D* via *Computer Vision System Toolbox*, that is a part of standard *Matlab* installation. The main advantage of using *Matlab* as a point cloud processor is a simple definition of algorithms for processing and comparison of *LiDAR* data. *Matlab* itself does not support loading of *LAS* files, however, there are multiple scripts available that support such operation.

We have used *Matlab* in this thesis for an implementation of *3D* data processing algorithm.

4 Line Detection in 2D Data

This chapter deals with processing of raw data, obtained from a 2D *LiDAR* device. The main goal of this chapter is to provide real-time data analysis. Specifically, to present an algorithm, that is able to detect lines in 2D data.

4.1 Points Filtering

The first step, before any other manipulations with the data, is filtering out invalid points. During some scan iterations, the *LiDAR* device may not detect all returning signals. Such points might have quality or distance equal to zero. We should not use them as valid points because doing so would result in malformed results. For this reason, we have decided to filter out such points from further processing.

4.2 Neighboring Points Linking

In the 2D continuous scan, only points in an angular neighborhood can create a line. We have decided to organize such points into groups of neighboring points. During a perfect scan, where no point has quality or distance equal to zero, all points from the scan create one group of neighboring points. However, scans usually have some defective points which result in multiple neighboring groups.

As a result of this linking process, we have several groups of points that could possibly create a straight line. A sample output of grouped points is visible in figure 4.1.

4.3 Segmentation of Neighboring Groups

In this subsection, we will go through significant part of line detection – a segmentation of groups of neighboring points. In this context, the segmentation means splitting groups that have significant changes in points structure into smaller groups with a small or negligible change of points structure.

To achieve this kind of splitting we have decided to use simple, but effective algorithm presented by David G. Lowe [23].

The algorithm accepts a string of points as an input. If a sub-string of the string is more straight with fewer direction changes the string is split

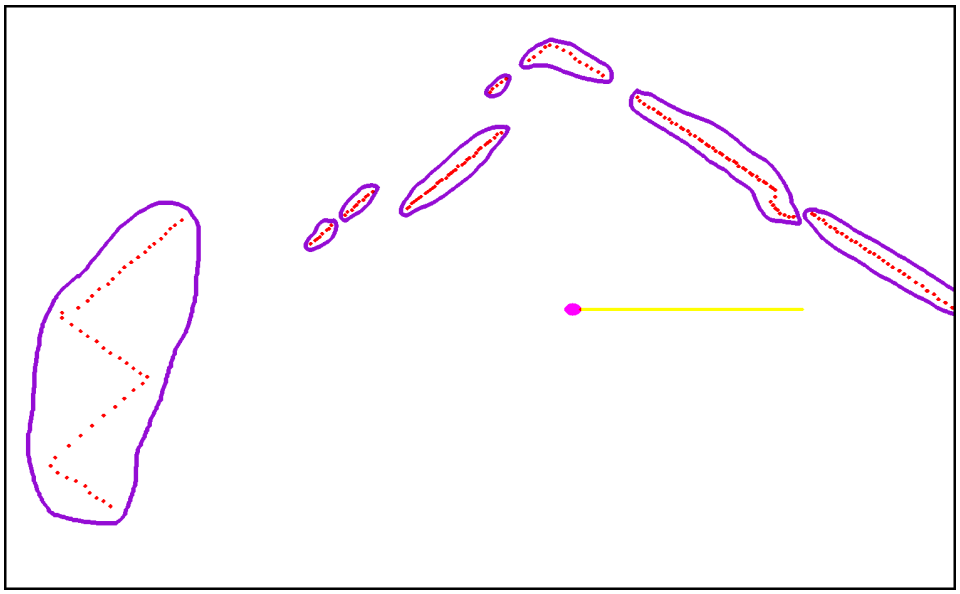


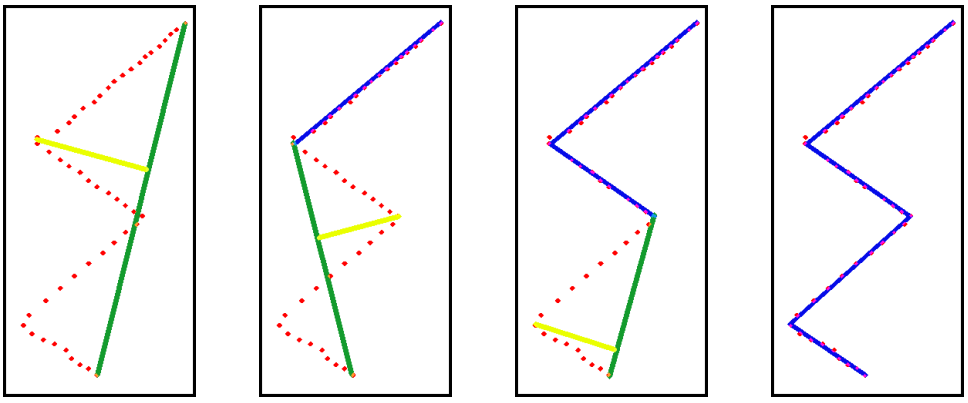
Figure 4.1: Grouping of neighboring points. Each cluster is marked with purple color. Clusters are split in a position, where a point is missing due to an error in measurement or due to the significant difference of adjacent points from *LiDAR* device.

recursively into smaller segments. To decide whether we will split the string into smaller segments or not, we compare a significance of straight line to fit the string and sub-strings. The significance is calculated by a ratio of the length of the segment divided by the maximum deviation of any point from the line [23]. This algorithm is described in the following algorithm 1. The progress of this algorithm is displayed in figure 4.2.

4.4 Lines Detection

After groups segmentation process, the line detection algorithm is applied, together with the calculation of the Root Mean Squared Error (RMSE).

The line detection algorithm measures an angle between three points in one segment; this is applied to all points from beginning to the end of the segment. If any angle is higher than a certain, user defined threshold, the segment is not a valid line. For each detected line, the RSME is calculated.



(a) (b) (c) (d)

Figure 4.2: Example of recursive string splitting on one group from figure 4.1. From left to right (4.2a – 4.2d) we can see an evolution of recursive detection of collinear pixels. The group is split recursively in point of most significant change (indicated by yellow normal). The significance comparison is applied recursively down to minimal, user defined, segment size.

4.5 Data Output and Storage Format

At the moment, we recognize two objects in 2D data – lines and points. We have decided to represent both in a text form. A point is a combination of angle, distance from the beginning and quality:

<quality>,<distance>,<angle>

A line string starts with prefix *l-*, and contains line's RMSE, starting and ending point:

l-<start point distance>,<start point angle>,<end point distance>,<end point quality>,<line rmse>

A common file format for storing *LiDAR* data is the LAS file format, as mentioned in subsection 2.1.5. Data in LAS file are stored point by point, where every combination of *X*, *Y*, and *Z* coordination exists just once. Along with coordinates, every point can have several attributes that are connected to the point. These attributes are listed in table 2.1.

Because of limited data coming from RPLidar, we are using only the *X*, *Y*, and *Z* values, while saving to LAS file. Due to nature of the LAS file format specification, the format is suitable only for saving static data, not for recording changes of scanned data in time.

This fact was one of the reasons why we have decided to create our own format for storing continuous data in time. Lines and points are stored in

Algorithm 1 Finds changes in a string of pixels and divides the string into multiple segments with no or small change

Require: S {*Array of neighboring points*}, minSegmentSize **return** Returns array of segments

```
1:  function FindSegments(S)
2:      biggestDistance ← −1
3:      segmentSignificance ← 0
4:      mostSignificantPoint ← nil
5:      for all p in S do
6:          distance ← distance of p to line between first and last point from S
7:          if distance > biggestDistance then
8:              segmentSignificance ← length of line between first and last
    point from S / distance
9:              mostSignificantPoint ← p
10:         end if
11:     end for
12:     if S.size >= minSegmentSize then
13:         {Split S in mostSignificantPoint into leftChild and rightChild}
14:         childrenSegments ← FindSegments(leftChild)
15:         childrenSegments ← FindSegments(rightChild)
16:         if Any child in childrenSegments is more significant than S then
17:             return childrenSegments
18:         end if
19:     end if
20:     return Array({ significance = segmentSignificance, segment = S })
21: end function
```

text file, in the same format as presented in subsection 4.5, one iteration per line, where every object is separated by a semicolon. Such data format allows reconstruction of the all data that were recorded by a *LiDAR* device and their changes in time. Sample scans in this format can be found on enclosed DVD, stored as .txt files in folder scans/rplidar.

5 3D Data Processing

This chapter deals with the analysis of 3D data, obtained with *LiDAR* scanners. Specifically with the problem of finding a flat area with given dimensions. In other words, we are looking for a flat area in a point cloud, that is spacious enough to fit a specified shape.

The main goal of this chapter is not to provide a real-time 3D data processing solution, but rather a validation of possible way of solving this problem.

We have identified multiple sub-tasks whose solutions will lead to an identification of suitable area. These tasks include:

- find a plane in a 3D space
- define an area on the plane
- fit a specified object into the area

Several approaches to these problems exist, and in following sections we will describe, how we decided to solve them.

5.1 Detecting Plane in Point Cloud

Detecting a plane in random 3D data point set is a task, that can be solved in several ways. The data obtained by a *LiDAR* scanner is a copy of a real-life environment, that usually contains more than one plane, advanced and robust algorithms include *The Hough Transform* [24] and *Random Sample Consensus (RANSAC)* [25]. Both have several parameters that affect final results, and both have multiple variations that alter computation requirements.

We have decided to use *RANSAC*, considering noise-robustness and implementation simplicity. The algorithm for finding a plane in 3D data is following:

1. Randomly select sample of minimum points, required to fit a model.

   In case of a plane, it means to select 3 points from the point cloud that define the plane. This selection step is presented in figure 5.1.

2. Compute fitting model from selected points.

   Having three points **A**, **B** and **C**, we need to compute coefficients **a, b, c** for the general plane equation:

$$ax + by + cz + d = 0 \tag{5.1}$$

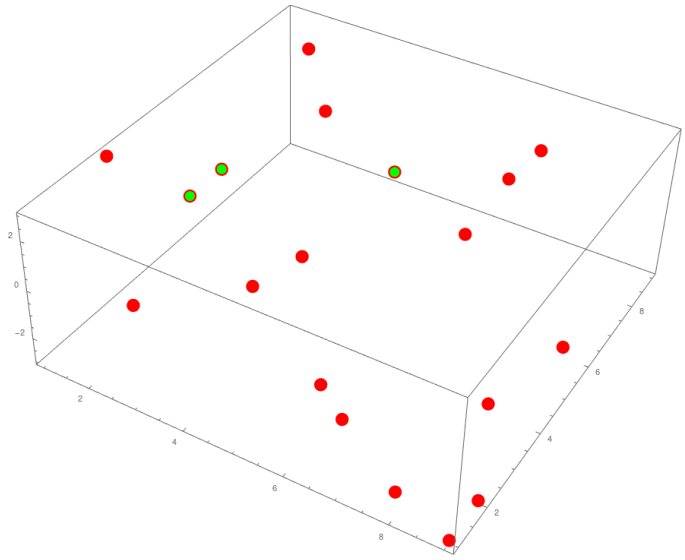


Figure 5.1: Randomly selected points (marked with a green color) in the point cloud.

Knowing that **a, b, c** are coefficients of the plane's normal vector, we can compute them as a cross product of vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$. Then we are able to compute **d** by using calculated coefficients **a, b, c** and any of **A**, **B** or **C** in the general plane equation. The interlacing is visualized in figures 5.2 and 5.3.

For the purposes of finding a landing area we want to find such plane, that is horizontal or near horizontal. To determine if the plane has demanded slope we just need to find angle α between normal vector of horizontal plane (for example $v = (0, 0, 1)$) and normal vector of plane $w = (a, b, c)$, using following formula 5.2.

$$\alpha = cos^{-1}\left(\frac{\overrightarrow{v}\,\overrightarrow{w}}{||\ \overrightarrow{v}\ ||||\ \overrightarrow{w}\ ||}\right) \tag{5.2}$$

3. The last step is to estimate a distance to the plane for each point of the point cloud. If the point is located on the plane or in defined neighborhood δ, the point is considered to be a part of the plane. For point $P = (x_0, y_0, z_0)$ and plane defined by the general plane equa-

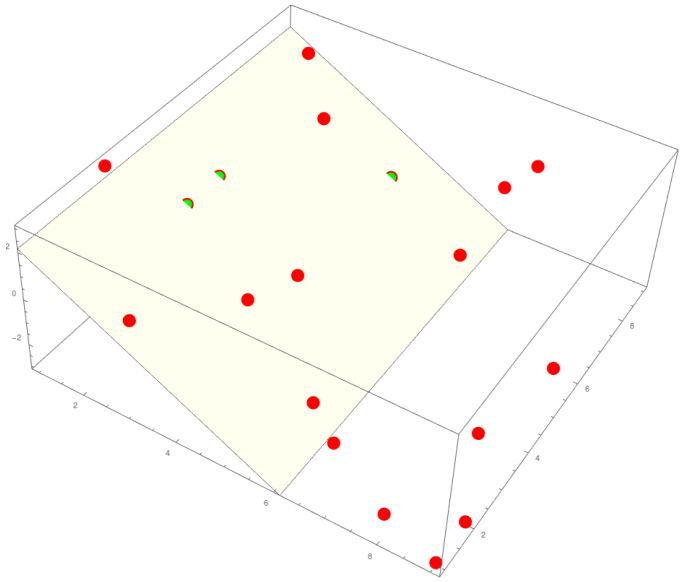


Figure 5.2: Compute fitting model of the plane.

tion, the distance d can be calculated with following equation 5.3. This step is displayed in figure 5.4.

$$d = \frac{\mid ax_0 + by_0 + cz_0 + d \mid}{\sqrt{a^2 + b^2 + c^2}} \tag{5.3}$$

Steps 1-3 are repeated for a fixed number of times until the plane with the most inliers is found. When the best plane is estimated, the set of points, that define the plane, is removed from the point cloud and a new instance of *RANSAC* is started on the reduced point cloud.

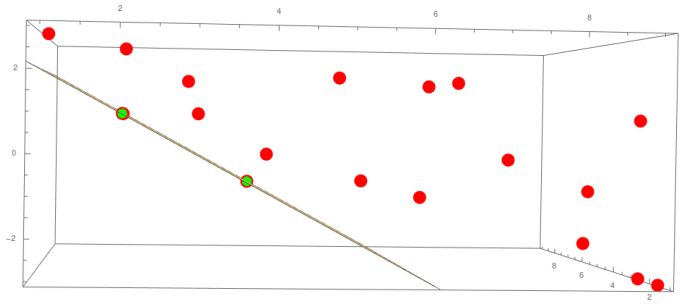


Figure 5.3: Fitting model from the side.

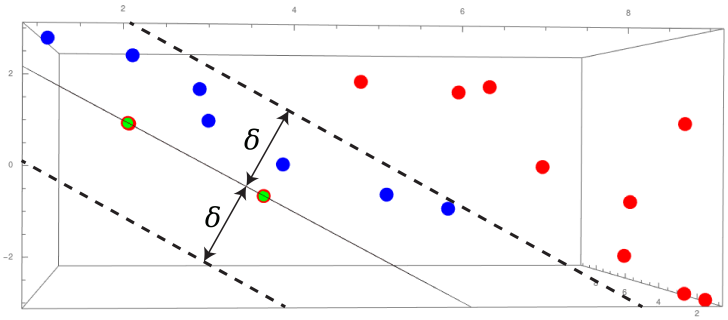


Figure 5.4: Select the set of points that are located below and above the plane in neighborhood δ.

5.2 Plane Processing

We have decided to use mathematical morphology and binary image to solve the problem of fitting an object into a plane. The whole process is described in section 5.3. In order to prepare the data of plane for a conversion to a binary image, we are required to perform some linear transformations that will convert the data to simpler format, more suitable for further processing.

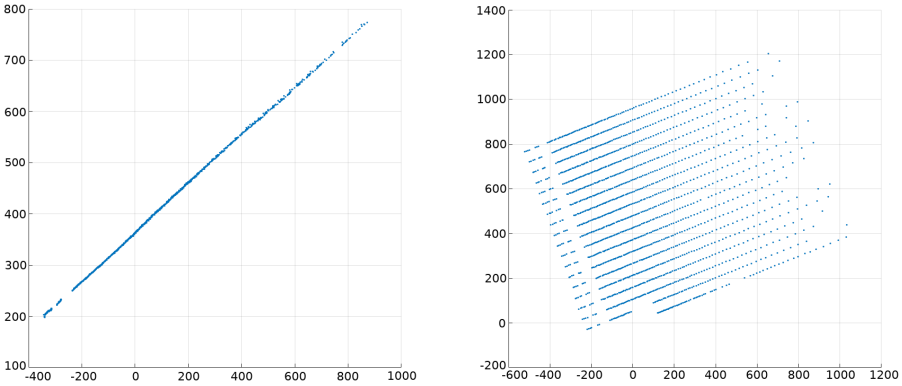
5.2.1 Points Normalization

Having a plane, that is defined by points in 3D space we can use the *Principal Component Analysis* (PCA), to get 3D plane transformed into 2D coordinate system. *PCA* is a statistical procedure that uses linear transformation which is being used for a change of an object dimension [26].

The algorithm has following steps:

1. Collect the input data $b^{(i)}$ (= detected plane)
2. Calculate covariance matrix Cv of $b^{(i)}$
3. Calculate eigenvectors and eigenvalues for matrix Cv
4. Use eigenvectors to form the transform matrix A
5. Use matrix A as a transformation matrix for original plane data
6. Get transformed and decorrelated plane data $w^{(i)} = A * (b^{(i)} - \bar{b})$

After the application of steps 1 to 6, we will obtain data, that is being centered in the origin and are aligned with coordinate axes [27]. Figure 5.5 visualizes the original and final position of points, that generate the plane.



(a) (b)

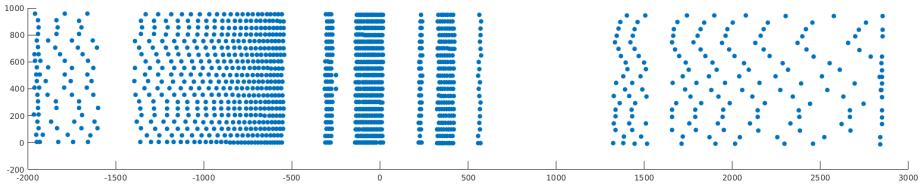
Figure 5.5: Figure a) contains the original position of points that define plane $b^{(i)}$ in a point cloud, b) represents the position of points after application of PCA.

5.2.2 Conversion to Binary Image

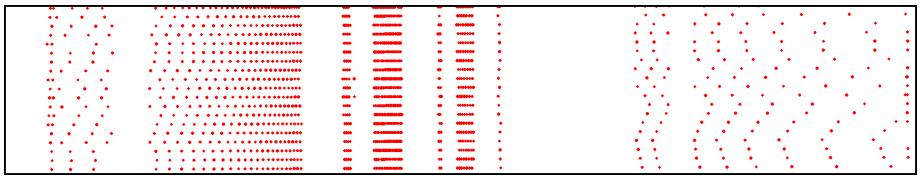
Having the plane centered, we can easily convert the data to a binary image. The image will have dimensions of $|\max x - \min x| \times |\max y - \min y|$, or proportionally scaled. Because the binary image we use, has the origin in the top left corner, all points have to be converted from Cartesian coordinates to image coordinates. Each point is moved to positive quadrant, with an equation from 5.4. The result can be seen in figure 5.6.

$$\begin{aligned} img_x &= \lceil x \rceil - \min x \\ img_y &= \lceil y \rceil - \min y \\ img_z &= \lceil z \rceil - \min z \end{aligned} \tag{5.4}$$

To get the result faster, together with lower computational and memory requirements, but with lower accuracy, we can apply scaling to every point of the plane.



(a) Normalized, aligned plane.



(b) Plane converted to a bitmap image.

Figure 5.6: Conversion of the point-defined plane to the binary image.

5.2.3 Points to Plane Approximation

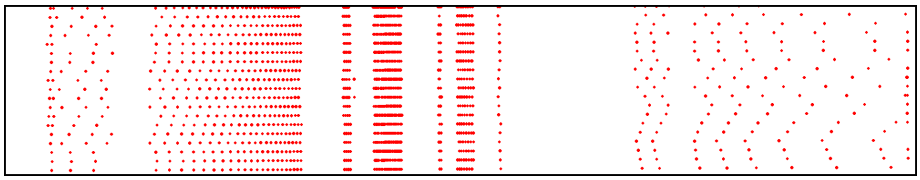
As one of the last part of preprocessing the plane, before fitting an object into the given data, we are required to define exact areas to search. In our case, it means to extend or link all points from the binary image and connect them together, in order to define the exact area.

We have decided to work with *Euclidian distance transform* [28]. The algorithm as an input accepts a binary image and the output is a distance map. The binary image is converted to a grayscale image, where the value in each pixel represents the shortest distance to any TRUE pixel from the binary image.

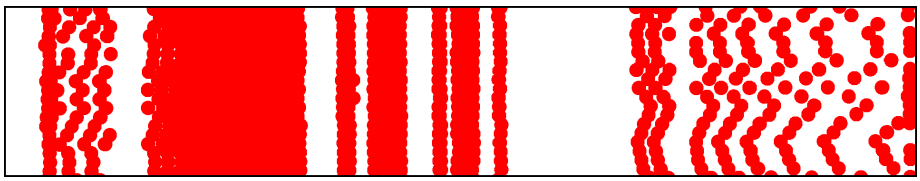
After that, we apply thresholding in size of d to every pixel of the distance map. The result is a binary image, that contains all original points, which are extended by pixels, that are within the maximal distance of d from original points. The parameter d is in our case *LiDAR's* accuracy. In figure 5.7 we can see the result of this transformation. In figure 5.7b, on the right side of the image, we can see distortion of points, which creates empty areas among them. Due to this fact, these points did not get connected, even though they are a part of real, flat surface.

5.2.4 Removing Invalid Points

Due to nature of circular scans, we can sometimes get a plane that is not valid for object fitting. Such plane can have some others objects above or



(a) Plane converted to a bitmap image.



(b) Extended points with *Euclidian distance transform*

Figure 5.7: Process of points extension.

below its surface, which makes the plane inconvenient for an aircraft landing.

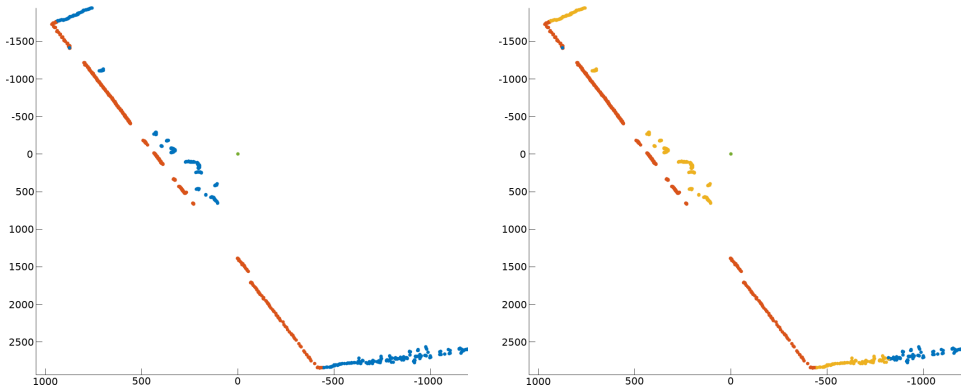
In figure 5.8a, we can see an example of such plane from a side, we assume, that the position of *LiDAR* scanner is $(0, 0, 0)$. Blue points represent whole point cloud but the detected plane and origin. The detected plane is marked with orange color and the origin is the only green point. There are some blue points in between origin and the plane, as can be seen in the picture. These blue points are not dense enough, so the *LiDAR* was able to obtain some information about the plane underneath.

To detect all points that are lying in between the origin and the plane, we can just simply find all points, that are located in between the detected plane P_1 and plane P_2 that is going through the origin and is parallel to P_1. Having the P_1 defined as $ax+by+cz+d = 0$, a parallel plane to P_1 would be $ax + by + cz = 0$. The set of all points, lying in between P_1 and P_2 is defined in equation 5.5 and also visible in figure 5.8b, marked with yellow color.

$$\{ (x, y, z) \mid ax + by + cx >= min(d, 0), ax + by + cx <= max(d, 0) \} \quad (5.5)$$

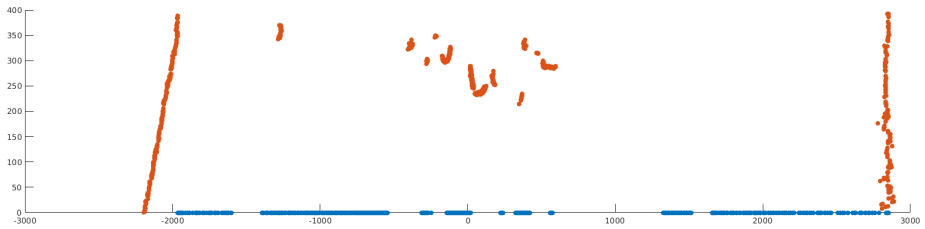
Having the set of points above the plane detected, we can process the set in the same way as we did process the set that defines the plane (sections 5.2.1-5.2.3). The result can be seen in figure 5.9. The same also applies for points bellow the plane.

To get the final plane, that is suitable for an area detection, we just have to subtract a binary image of points above and below the plane and the binary image of the plane.

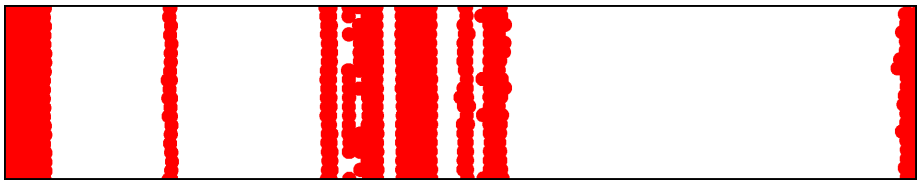


(a) (b)

Figure 5.8: Figure a) visualizes a top view on a scene, where a plane was detected (marked with orange color), all other points in the scene are marked with blue color. Figure b) shows the scene from a), with highlighted points (marked with yellow color) between a device and the detected plane.



(a) Points above the plane from the side. Blue points visualize detected plane, whereas orange points represent points above the plane.



(b) The mask of points, that were identified to be above the plane. Points in this mask are already enlarged using the *Euclidian distance transform*.

Figure 5.9: Visualization of points above the plane. Figure 5.9a visualizes height profile of points above the plane.

In figure 5.10, we can see the binary image 5.7b after application of binary mask from figure 5.9b. We can see that the solid area in the middle of 5.7b was disrupted after this application.

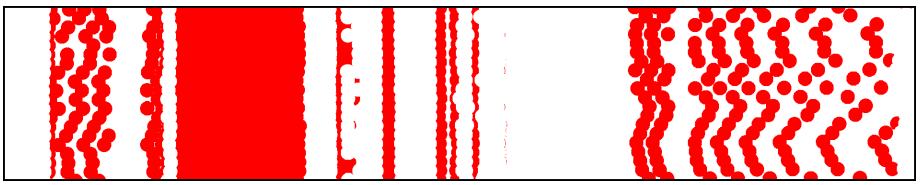


Figure 5.10: Points of the plane after subtraction.

5.3 Fitting Object into Plane

For suitable areas detection, we are using morphological transformation called **opening** [29]. Opening is a morphological filter that is composed of two basic morphological operations erosion and dilatation, where dilatation is applied after the erosion.

The **erosion** is an operation for a binary image, that fits the *structuring element* at every location of the image. The pixel is accepted as valid only if all pixels of the input image under the *structuring element* are set to true. Pixels near boundary are discarded depending on the *structuring element* size. It is useful for removing noise or detaching connected objects. In figure 5.11 we can see figure 5.10 after application of the erosion because the *structuring element* is bigger than the diameter of *Euclidian distance transform*, we can see that standing alone points were erased.

Figure 5.11: Figure 5.10 after the application of erosion filter.

Dilation is just opposite of the erosion. The pixel is accepted if at least one of pixels under the *structuring element* is true [30]. In figure 5.12 we can see figure 5.11 after application of dilation. We can see that original area was slightly extended.

To find out which points of original point cloud are valid plane points we just need to go through all points that define the plane and get the point

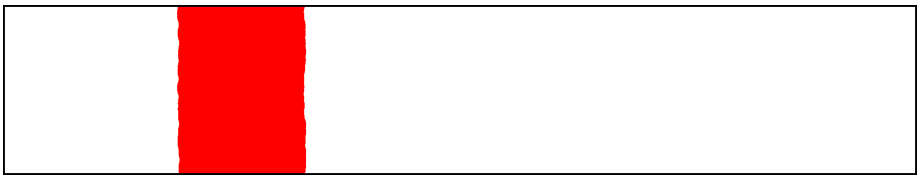


Figure 5.12: Figure 5.11 after the application of dilatation filter.

coordinate using formula 5.4. If the value in the binary picture on given coordinates is higher than zero, the point is a valid point of the plane. To get original position of valid plane points in 3D space, we can use inversion transformation using inversion matrix to matrix A from subsection 5.2.1.

5.4 Implementation

The implementation of an algorithm described in this chapter was created only as a script for *MATLAB* environment. We enclose the implementation to this thesis on DVD. The script requires the use of one non-standard library. Installation of this library and manual how to use this script are described in appendix D.

6 Implementation of Line Detection

Tables A.1 and A.2 provide a list of several *LiDAR* devices and suitable computational boards. Together with hardware possibilities, we have also described software and libraries available for *LiDAR* data processing in subsection 3.2. Following chapter describes the use of software and hardware in the implementation part of this thesis.

6.1 Hardware

In the implementation part of this thesis, we have worked with *Raspberry Pi 3* computation board. As a *LiDAR* device, we have used RPLidar A2 manufactured by Slamtec.

Raspberry Pi is a credit card size ARM computer, that is able to run full ARM operating system. As a primary operating system, Raspberry uses Raspbian, which is modified version of Debian Linux. Newest version of Raspberry, Raspberry Pi 3 is able to run IoT Core version of Windows 10. Used Raspberry is displayed in figure 6.1.

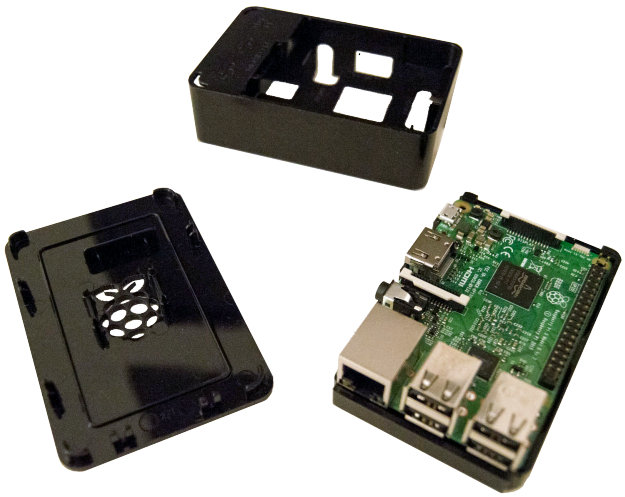


Figure 6.1: Raspberry Pi board with a protection case.

For input and output, there are several IO ports available – 17 General-purpose input/output (GPIO), 4 USB ports, single 10/100 Mbit Ethernet port,

Bluetooth 4.1 and wireless LAN. The board needs external, 2.5A micro USB power supply [M19].

Figure 6.2: Slamtec RPLidar A2 scanner with micro USB reduction board.

RPLidar A2, manufactured by Slamtec, was our sample *LiDAR* device in this thesis. It is a small device (diameter 75,7 mm, height 40,8 mm, and weigh 190 g), that is capable of scanning its surroundings in a full angle with a maximal distance of 6 m. It uses 3 mW infrared laser with a typical wavelength of 785 nm. As for safety, this device meets requirements of *Class I laser safety standard*, which means that this device is not dangerous to human nor animals. The device is shown in figure 6.2.

It is equipped with one IO XH2.54-5P male connector, which scheme is described in figure 6.3. The only connector is being used for charging, data transfer and controlling the device. RPLidar A2 is being delivered with reduction board to micro USB (visible in figure 6.2) [M19].

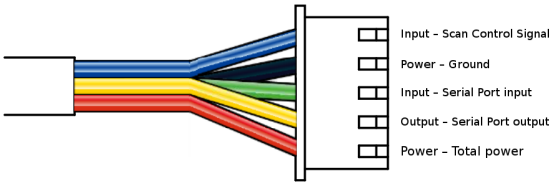


Figure 6.3: XH2.54-5P connector. Image adapted from [M19], edited.

We used one of Raspberry's USB ports for connection with the *LiDAR* device. After the connection is established (signalized by the green light on reduction board), Raspbian is ready to communicate with *LiDAR* without

any additional drivers. Through this connection, we are able to obtain scan data and control speed of scanning.

For a practical reason (higher speed of development), we have not used Raspberry for a development of communication and processing software. Instead of that, we have decided to develop directly on a PC with Linux with RPLidar connected, which provided us with more powerful tools as well as better usability. The setup of RPLidar and Raspberry Pi is shown in figure 6.4.

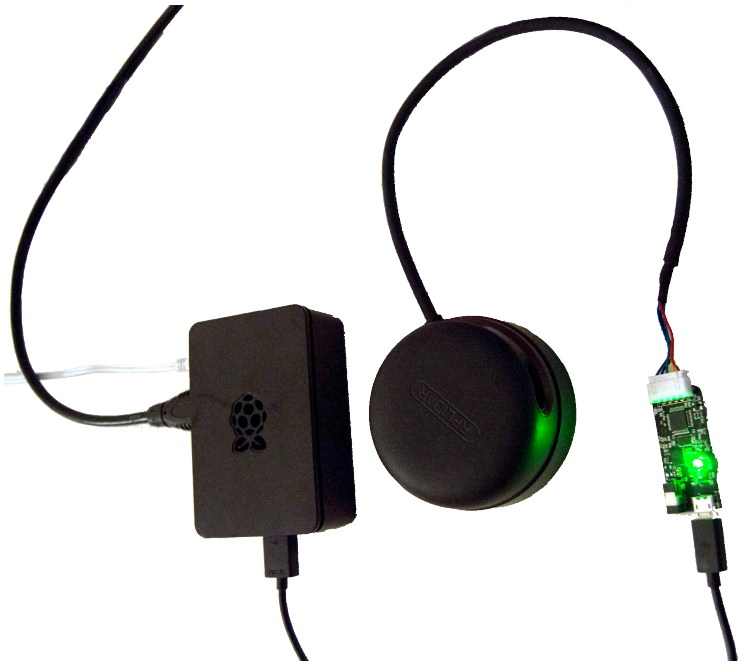


Figure 6.4: RPLidar A2 and Raspberry board setup.

6.2 Software

In a software development part of this thesis, we have decided to split the implementation into 3 smaller programs that can communicate with each other and possibly work alone, rather than making one, monolithic solution. These programs communicate with *LiDAR* device and fetch the data (**connector**), analyze obtained data (**analyzer**), and visualize the data (**visualizer**).

One of the reasons why we have chosen to split the implementation was also the possibility of interoperability among these programs. Not all of

them are required to be in use at all times (for example, when we want to display just raw data without analysis) so we do not have to include such program into data processing chain. This data processing chain is illustrated in figure 6.5.

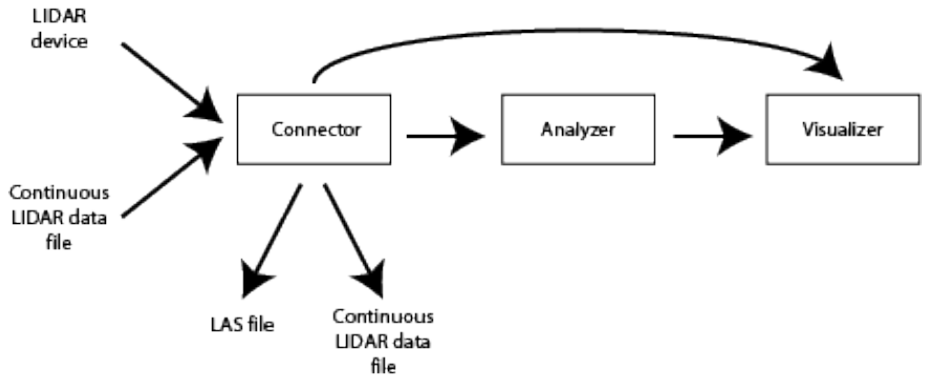


Figure 6.5: Data processing chain. Arrows represent a possible flow of the data. This diagram can also be understood as a data flow between computing units, running each program separately.

Thanks to usage of Raspberry Pi with Raspbian, we could have chosen from all variety of programming languages. Due to performance reasons and the best support provided by Slamtec, we have chosen *C++*. For a communication between programs, we have decided to use sockets, mainly due to widespread compatibility. Another reason for using sockets is a possibility to distribute challenging tasks among multiple machines. In following subsections, we will describe implementation and possibilities of each program.

6.2.1 Connector

This program provides a communication channel between *LiDAR* and computer. The program itself is an executable file *lidar_connector* and can be controlled through parameters, list of all of them and detailed usage is described in the attachment D.

The Connector is capable of reading data from multiple sources – currently from a *LiDAR* device or from a file with previously scanned data in the continuous format specified in section 4.5. The input file can be specified with parameter *-i* or *--input*. Apart from reading the data, the program is also capable of saving incoming data into continuous format described

in section 4.5 or into *LAS* file format (only one scan iteration can be saved in this case). Output to a file can be activated by applying parameter *-l* or *--log*, *LAS* format will be saved if a suffix of a file name is ".las", otherwise continuous, text format will be used.

The data flow through the program without any change, except for filtering out invalid points and application of sliding average. It is applied to reduce extremes in scanning caused by inaccuracy of a device. Parameter *-m* or *--avg* specifies how many iterations will be averaged.

Typical data flow would be from the Connector to the Analyzer, and then to the Visualizer. However, the Analyzer can be skipped and Visualize can accept raw data which are in the form of points. Sample output of data raw data from the Connector, visualized in the Visualizer is shown in figure 6.6.

Several libraries are used in the Connector implementation:

- **RPLidar A2 SDK** – provides basic API for the *RPLidar* device
- **LASLib** and **LASZip** – toolset for creating files in *LAS* format
- **libsocket** – simplifies POSIX sockets API
- **cpplinq** – simplifies manipulation with collections

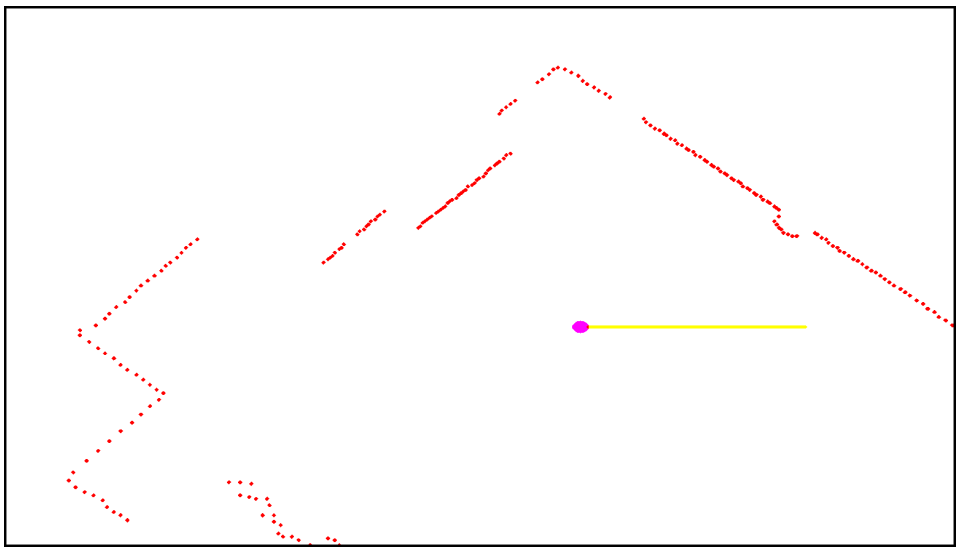


Figure 6.6: Raw non-analyzed data from the Connector visualized in the Visualizer.

6.2.2 Analyzer

The Analyzer part is a significant part of this thesis. Implementation is located in a binary file named *lidar_analyzer* and implements algorithms described in sections 4.2 – 4.4, which provide lines detection. Parameters that are accepted by the program are described in appendix E. Only accepted input through sockets is the text form of continuous scan data, that is described in section 4.5. The same format is also provided by the output.

Firstly, when analysis stars, the Analyzer detects neighboring points and links them into groups as described in section 4.2. The corresponding implementation is located in class *PointsLinker*, method *FindStrings*.

After all neighboring points are grouped, the segmentation algorithm from section 4.3 is applied to them, in order to split them into almost colinear subgroups. This algorithm is implemented in class *LinesSegmentor*, method *FindSubStrings*.

Last but not least, the final lines detection algorithm is applied and after each line detection, the RMSE is calculated. The whole process is described in section 4.4 and source code is located in class *LinesDetector*, method *DetectLines*.

Together with **libsocket** (API for sockets) and **cpplinq**, **Eigen3** math library was used for a development of the Analyzer.

6.2.3 Visualizer

The last part of the data processing chain is the Visualizer. The program itself is located in the binary file *lidar_visualizer*. This program provides a visual representation of scanned data and analyzed lines, together with user interaction during data exploration. Apart from that, the Visualizer allows using filters in order to disable or enable visibility of points or lines, filtering lines according to their RMSE or length, and to distinguish lines with colors depending on their quality.

The Visualizer communicates through sockets and accepts incoming data in the text form continuous scan. This means that this program is able to communicate with the Connector or the Analyzer. Several different parameters are accepted by the program and are described in appendix F.

Multiple external libraries were used for a development of this program including **libsocket** for socket communication, **cpplinq** and **Irrlicht** for graphical representation and rendering the visualized scene. Screenshots of the *Visualizer* are shown in figures 6.7 and 6.8.

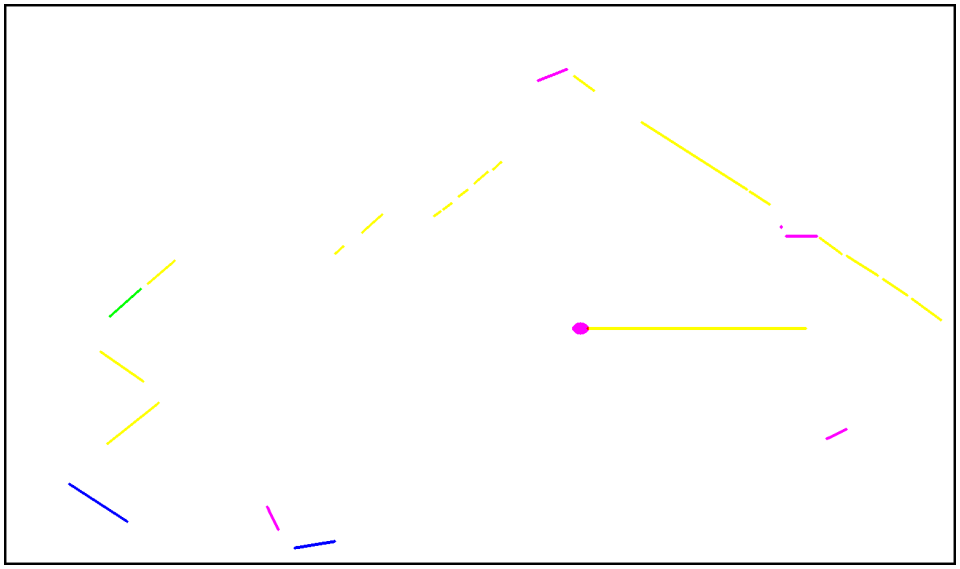


Figure 6.7: Visualizer with just lines displayed.

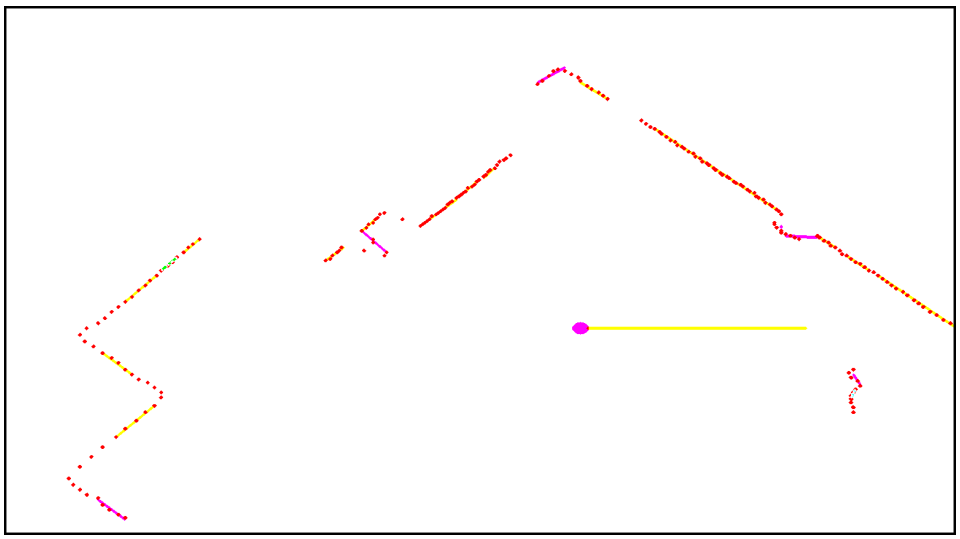


Figure 6.8: Visualizer with filtered lines, together with original detected points.

7 Evaluation

This chapter contains an evaluation of line detection algorithm proposed in Chapter 4, and object to plane-fitting algorithm proposed in Chapter 5. The data acquisition and used methodology are also described in this chapter.

7.1 Data Acquisition

To test proposed algorithms, we had to obtain a sample either 2D or 3D *LiDAR* data. We have primarily used our own scans, taken with Slamtec RPLidar A2 scanner, supplemented with an open source data. Technical information about RPLidar is mentioned in subsection 6.1 and table A.2. All of the mentioned data files are available on data DVD enclosed to this thesis.

As for *2D* data, we have created multiple scans in continuous scan format from the section 4.5. These scans are stored in files *room01<location>.txt - room04<location>.txt*, where <*location*> stands for the name of location where was the scan taken. All of them contain maximally 360 points per iteration (due to hardware limitations) and 59 iterations in average. Records also include moving elements in captured data, so the detected lines may change in time. For a *2D* data, we have not used any open source, third-party scans.

7.1.1 3D Data Acquisition

Acquisition of *3D* point cloud data was more complicated than *2D* acquisition because we did not have any *3D LiDAR* device available. However, we managed to simulate *3D* scanner behavior with RPLidar A2 using slightly modified *lidar_connector* (see 6.2.1). The modification is following – after each cycle, the *lidar_connector* pauses scanning and wait for user input, that gives the user an opportunity to manipulate the device (move the device upwards). On user input, the *lidar_connector* continues with scanning, with increased z coordinate, which results in layering *2D* on each other and creating similar output as would *3D* scanner make.

All *3D* scans are saved in *LAS* format, which means that they are just static, single iteration scans. The reason why they are all static is that we are not able to achieve records moving data correctly with layering *2D* data. Scans are stored in files *3Droom01<location>.las - 3Droom04<location>.las*.

7.2 Evaluation Methodology

To evaluate the accuracy of proposed algorithms, we are using following statistics:

Positive Samples (P) – a number of all points that were recognized as a part of a line or a plane.

Negative Samples (N) – a number of all points that were recognized as surroundings of a line or a plane.

True Positive (TP) – a number of all points that create a line or a plane and were correctly recognized as a part of such object.

True Negative (TN) – a number of all points that do not create a line or a plane and were correctly recognized as a neighborhood of a line or a plane.

False Positive (FP) – a number of all points in a point cloud, that are valid points of a line or plane, but were recognized incorrectly as a neighborhood.

False Negative (FN) – a number of all points in a point cloud, are not a valid part of a line or a plane but were recognized as a part of such object.

Thanks to the knowledge of most of the scans' locations and locations dimensions, we can manually determine the *ground truth* and compare scanned data to the real situation. In case of *3D* scans, the analysis is not done for all points of the point cloud, but only for points that were detected by *RANSAC* algorithm as a plane.

From these statistics, we can calculate proportional statistics, which give us more specific information about the accuracy of proposed algorithms and can be compared with statistics from another data sets. The statistics are:

True Positive Rate / Sensitivity (TPR) – a percentage of positives that are correctly identified as positives.

$$TPR = \frac{TP}{(TP + FN)} \tag{7.1}$$

True Negative Rate / Specificity (TRN) – a percentage of negatives that are correctly identified as negatives.

$$TRN = \frac{TN}{(TN + FP)} \tag{7.2}$$

False Positive Rate / Fall-Out (FPR) – a percentage of positives that are incorrectly identified as negatives.

$$FPR = \frac{FP}{(FP + TN)} \tag{7.3}$$

False Negative Rate / Miss-Rate (FNR) – a percentage of negatives that are incorrectly identified as positives.

$$FNR = \frac{FN}{(FN + TP)} \tag{7.4}$$

Accuracy (ACC) – is a combination of precision and trueness.

$$ACC = \frac{(TP + TN)}{(TP + FP + FN + TN)} \tag{7.5}$$

7.3 Data Sets

Thanks to the continuity of *2D* data, we have a significant amount of *2D* samples available for the analysis. We have the following number of scans in obtained data files: *room01-office.txt* contains 51 iterations, *room02-meetingroom.txt* contains 46 iterations, *room03-kitchen.txt* contains 102 iterations, and *room04-livingroom.txt* contains 37 iterations. That is 236 full scans in total for *2D* data.

As for *3D* scans, we have static scans of 4 rooms available – *3Droom01-corridor.las* (10), *3Droom02-bathroom.las* (7), *3Droom03-kitchen.las* (10), and *3Droom04-livingroom.las* (6). The number in brackets is the number of walls which we have manually identified as a suitable location for a plane, which means that we have approximately 33 possible areas for object fitting.

7.4 Results

As mentioned in section 7.3, for a *2D* testing, we have 236 iterations available. We evaluate these iterations to get statistics discussed in section 7.2

with changing a number of previous iterations n for calculation of moving average.

Regarding *3D* scans, we have 33 possible planes to analyze. We observe how the change of *Euclidian distance transform* diameter d, influences statistics.

In tables 7.1 and 7.2 we can see obtained statistics. All values represent average values of all iterations for a given scan.

	N	P	TPR	TNR	FPR	FNR	ACC
$n = 0$							
room01	295	51	99,09 %	43,03 %	56,97 %	0,91 %	**90,51** %
room02	280	49	99,05 %	35,06 %	64,94 %	0,95 %	89,43 %
room03	212	46	98,92 %	**47,18** %	**52,82** %	1,08 %	89,58 %
room04	265	55	**99,49** %	37,18 %	62,82 %	**0,51** %	88.88 %
$n = 5$							
room01	292	55	99,31 %	36,58 %	63,42 %	0,69 %	**89,29** %
room02	260	66	99,09 %	36,45 %	63,55 %	0,91 %	86,38 %
room03	220	62	99,21 %	**53,97** %	**46,21** %	0,79 %	89,25 %
room04	254	62	**99,37** %	44,73 %	55,27 %	**0,63** %	88,75 %
$n = 10$							
room01	290	59	99,51 %	37,71 %	62,29 %	0,49 %	88,88 %
room02	257	70	99,49 %	43,53 %	56,47 %	0,51 %	87,47 %
room03	216	63	99,34 %	**53,11** %	**46,89** %	0,66 %	**88,96** %
room04	245	88	**99,59** %	48,64 %	51,36 %	**0,41** %	88,94 %

Table 7.1: Statistics of lines detected real time in 2D data. Best values for a given column are visualized with bold font. We can see that for TPR, the algorithm did well. However in case of FPR the results were not satisfactory enough and need more investigation of parameter choice.

	P	N	TPR	TNR	FPR	FNR	ACC
$d = deviceResolution$							
3Droom01	343	288	100 %	71,40 %	28,6 %	0 %	90,67 %
3Droom02	661	261	100 %	75,59 %	24,41 %	0 %	**97,44** %
3Droom03	326	321	100 %	**88,39** %	**11,81** %	0 %	96,04 %
3Droom04	388	538	100 %	85,35 %	14,65 %	0 %	93,49 %
$d = 2deviceResolution$							
3Droom01	369	337	**100** %	71,08 %	28,92 %	**0** %	93,82 %
3Droom02	619	303	99,15 %	**80,96** %	**19,14** %	0,85 %	**94,95** %
3Droom03	353	294	**100** %	78,79 %	21,21 %	**0** %	94,78 %
3Droom04	362	564	99,91 %	80,41 %	19,59 %	0,09 %	90,63 %

Table 7.2: Statistics of finding suitable areas in 3D data. The *structuring elements* is a circle with a diameter of 30 cm. Resolution of the *LiDAR* device is 5 cm. Best values for a given column are visualized with bold font. We can see that algorithm did very well in case of TPR, with generally low FPR.

7.5 Results Interpretation

7.5.1 2D Accuracy

In table 7.1 we can see, that the accuracy is in average **88,91 %**. As visible in figure 7.1, we have achieved very high TPR for the line determination – **99,27 %**; however, the average TNR is **42,33 %**. We assume that this low percentage result is caused by the combination of multiple factors. Firstly, the application parameters were the maximal deviation angle allowed was 5°, which might be too strict, and secondly, the *RPLidar* is not too accurate. This means that even the same point is usually returned in a slightly different distance per each iteration.

As for the moving average, we can see increasing **True Positive Rate** with higher n.

The results were stable during all scans, with no significant extremes compared to other scans.

7.5.2 3D Accuracy

As for *3D* results, in table 7.2 and in figure 7.2 we can see, that we have achieved a very high percentage of plane determination, **TPR** – almost **99,91 %**. Regarding the plane surroundings estimation – **SPC**, the worst case is **71,08 %** for *3Droom01*, and the average is **78,88 %**. The **False Negative Rate** was low, with some extremes visible in figure 7.2, and total average of **21,12 %**. The average accuracy is **93,58 %**.

The change of parameter d from $deviceResolution$ to $2deviceResolution$ results in a lower average of **TNR** and introduces errors to plane determination, which leads to a lower average of **TPR**. The decrease of **TNR** is caused by an extension of marginal points of surroundings.

We assume that the gained percentage of **TPR** could be increased by more wise choice of parameter d. Instead of choosing only one static value, it might be better to select the d dynamically, with concerning scanned angle and distance.

7.5.3 Speed

The line detection algorithm is supposed to be real-time. Therefore we have measured a time for every iteration with following results – average time: **4,5 ms**, maximal time: **12 ms**, and minimal time: **1 ms**. Tests were taken on a computer with CPU *Intel Core i5-3210M @ 2.50GHz*. Measured values should be satisfactory enough for application in real-time situations, in combination with *RPLidar*, whose maximum frequency is 15 Hz, which is 66,7 ms for one rotation.

As for *3D* algorithm, we have not planned to provide real-time implementation, and our *Matlab* implementation analyses a plane in tens of seconds. Usually, the analysis does not take more than 15 seconds.

7.6 Landscape Data

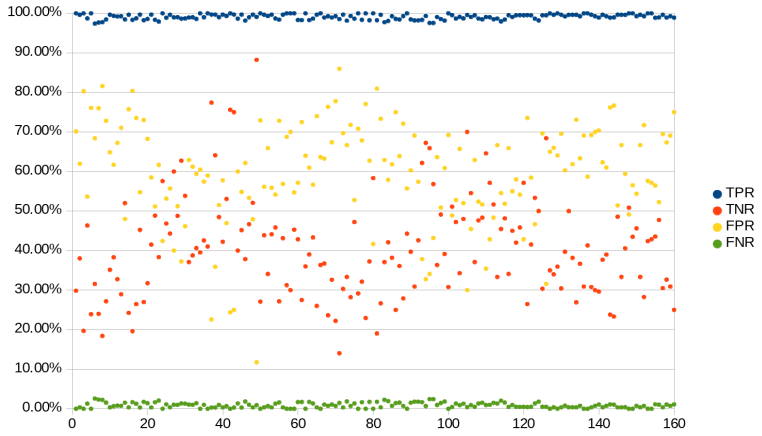
One of the tasks of this thesis, was also to test the *3D*, area search algorithm with an open source, real-world data. In this section, we are showing the algorithm results on aerial, *3D* scan from a UAV. The *LiDAR* scan was downloaded from `lidarusa.com`, where multiple, free scans can be found. We have chosen this one because it contains some flat areas, that might be suitable for testing or the *3D* algorithm.

The original scan is extremely dense (it contains more than 100 millions points), so we were required to subsample the scan, in order to make

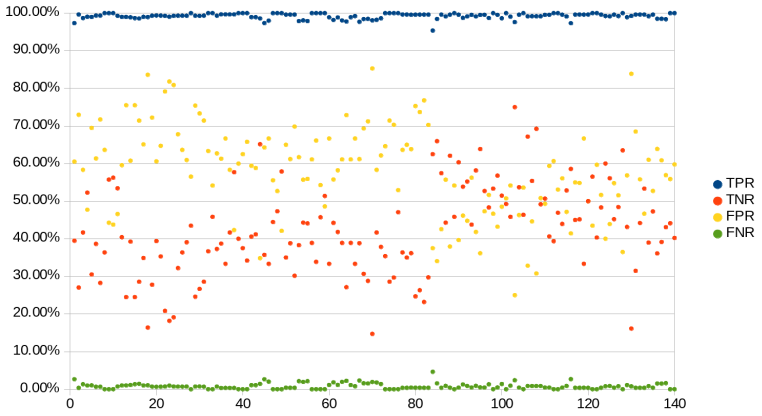
it suitable for *Principal Component Analysis*, which has significant memory requirements. We can see the original scan in figure 7.3, whereas the subsampled version in figure 7.4. The detected area is visible in figure 7.5. The original scan was captured with *Velodyne HD-32*, however, we do not know any other circumstances of the scanning process, so we cannot create analysis similar to the one, presented in subsection 7.5.2.

We have only checked visually if there are any obviously invalid points marked as valid and we have found out, that there are some invalid points marked as the suitable area, this is visible in figure 7.6. We assume, that this is caused by subsampling, where some information is lost. All in all the algorithm did well and marked the only area, that we would mark as well. The other areas contain some objects on their surface or are too small.

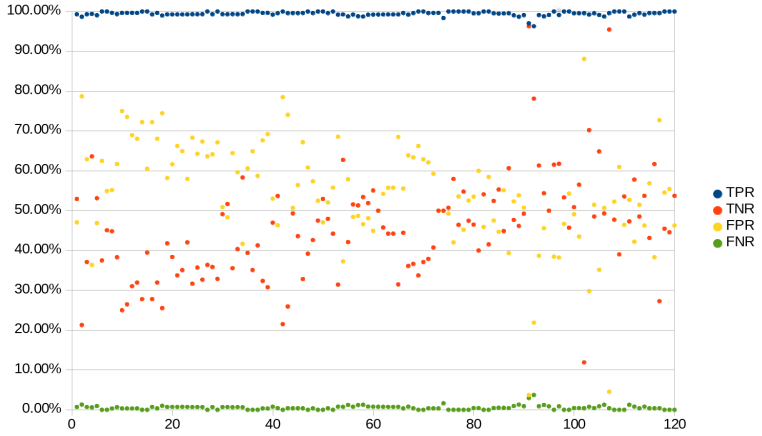
We are enclosing this scan to this thesis on DVD, it is located in folder *scans/thirdpart*, the original file is called *ny-uav.laz*, and the subsampled a denoised is called *ny-uav-subsampled.laz*. Both are compressed to *LAZ* format and have to be decompressed to *LAS* before use with enclosed *Matlab* script.



(a) $n = 0$

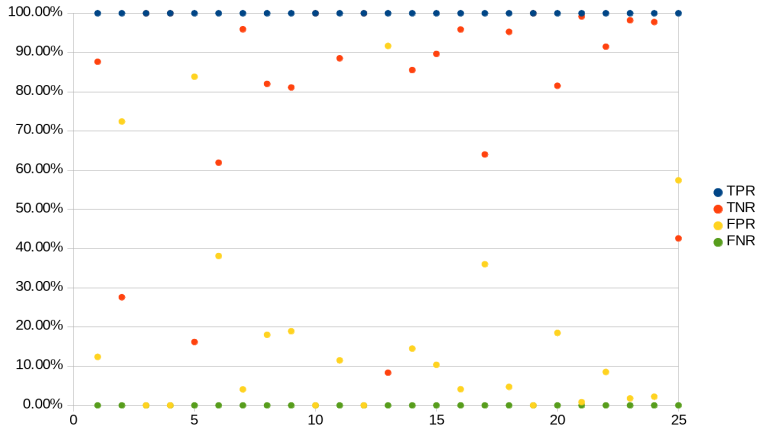


(b) $n = 5$

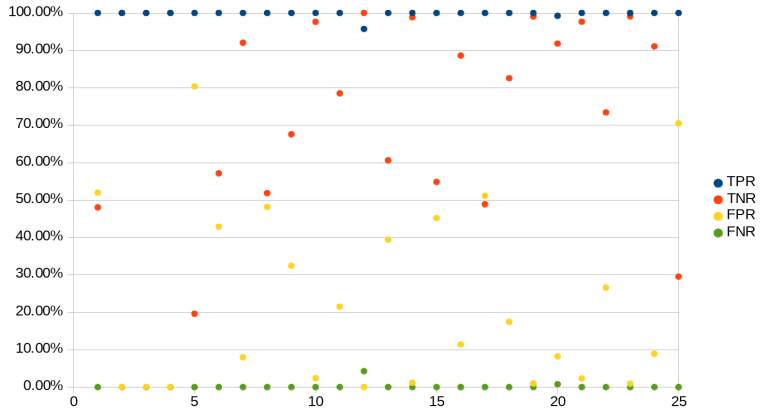


(c) $n = 10$

Figure 7.1: Visual representation of all results from table 7.1.



(a) $d = deviceResolution$



(b) $d = 2 * deviceResolution$

Figure 7.2: Visual representation of all results from table 7.2.

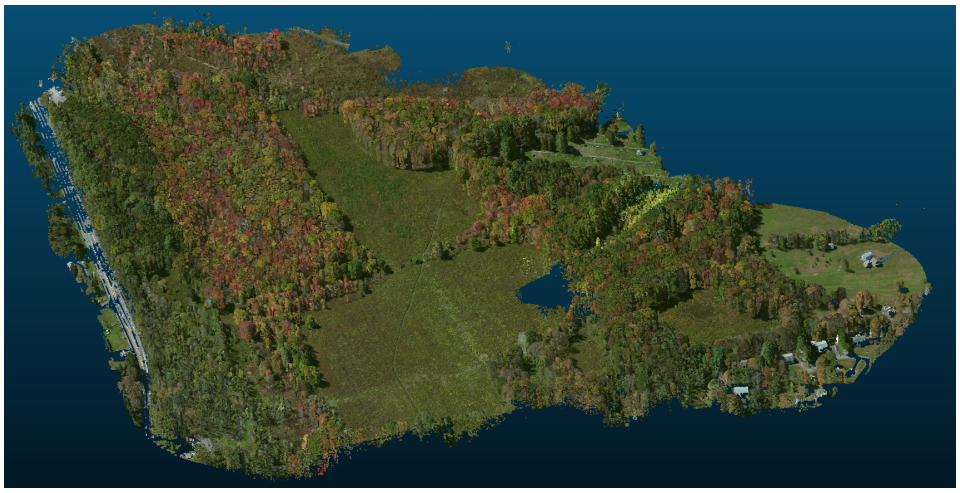


Figure 7.3: The original scan from `lidarusa.com`.

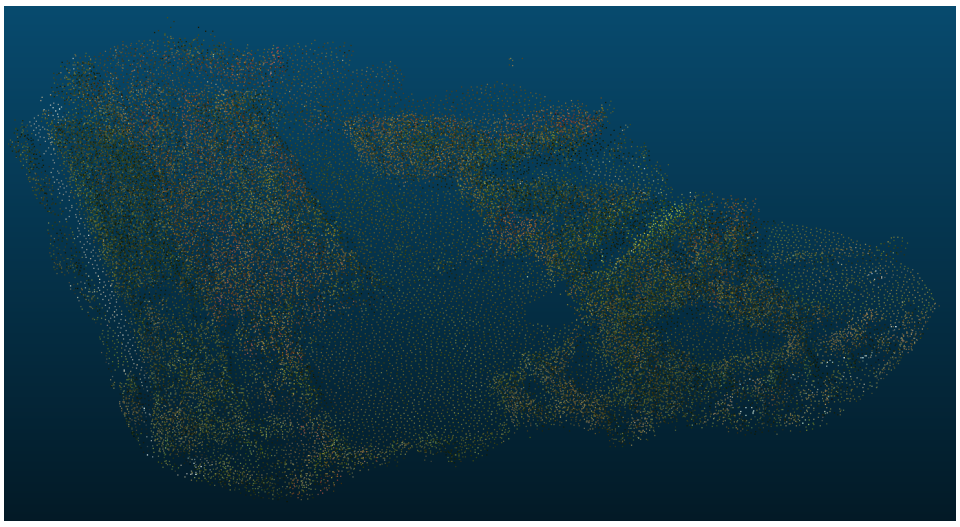


Figure 7.4: The original scan, subsampled to have less memory requirements.

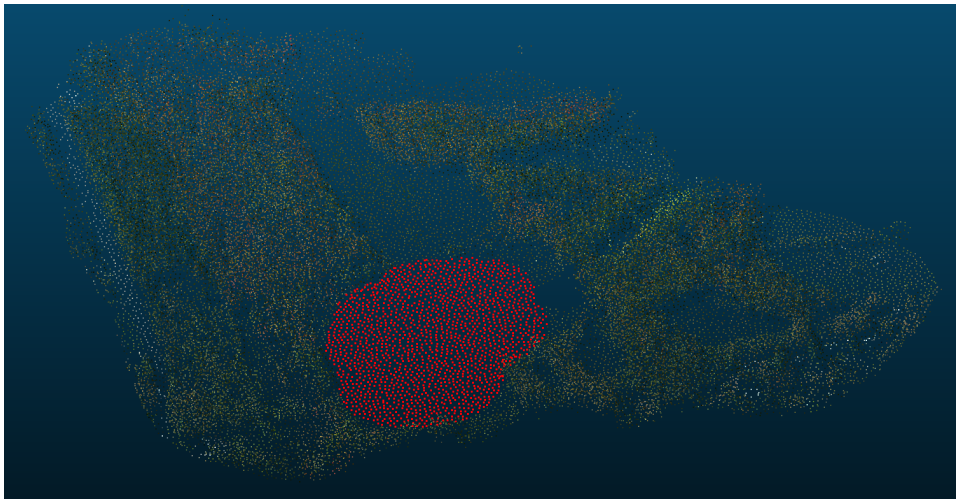


Figure 7.5: Subsampled scan with highlighted area, that fulfills requirements.

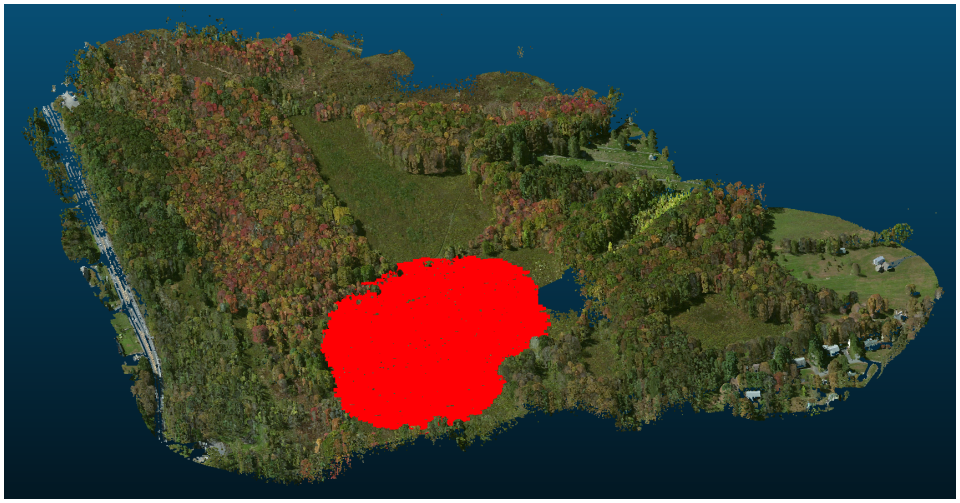


Figure 7.6: Suitable area, visualized with original scan.

8 Conclusion

We have introduced the basic information about *LiDAR* devices to the reader, together with a comparison of multiple different devices as well as a comparison of suitable accessories. Besides the hardware, we have also listed multiple software libraries and tools which are suitable for processing of point cloud data. We have tested some of these libraries and tools during the implementation part of this thesis.

In the main part of the thesis, we have proposed algorithm for a line detection in *2D* data and area determination in *3D* data. As for *2D* line algorithm, we have provided real-time implementation, and for area determination in *3D* data, we have provided a *MATLAB* script that demonstrates suggested algorithm.

From obtained results we can see, that the accuracy of *2D* line detection algorithm is quite high, however for real use it would need an investigation to find the best parameters combination to reduce *False Positive Rate*. The implementation is sufficiently fast for real-time usage.

The algorithm for *3D* area determination also has a quite high accuracy, together with overall low error rate. We have also tested the algorithm with the data, that might be suitable to test the finding a landing spot scenario, with fairly good result. For real use it would be necessary to create an effective implementation, that would remove some bottlenecks of our solution (e.g., *Principal Component Analysis*). Also, more depth inspection of the influence of dynamically selected circle diameter might increase the accuracy as well.

Bibliography

1. PARRISH, Christopher. *Lidar and Height Mod Workshop* [online]. 2011 [cit. Nov. 19, 2016]. Available from: https://www.ngs.noaa.gov/corbin/class_description/Parrish_Lidar_and_Height_Mod_Presentation.pdf.
2. OCEANIC, National; CENTER, Atmospheric Administration (NOAA) Coastal Services. *Lidar 101: An Introduction to Lidar Technology, Data, and Applications* [online]. 2012 [cit. Nov. 20, 2016]. Available from: https://coast.noaa.gov/digitalcoast/training/lidar-101.html.
3. VALLABHANENI, Thagoor. *Autonomous Vehicles* [online]. 2011 [cit. Oct. 10, 2017]. Available from: http://www.cvel.clemson.edu/auto/AuE835_Projects_2011/Vallabhaneni_project.html.
4. *How does LiDAR work?* [online]. 2017 [cit. Dec. 10, 2017]. Available from: http://www.lidar-uk.com/how-lidar-works/.
5. NAYEGANDHI, Amar. *Green, waveform lidar in topo-bathy mapping – Principles and Applications* [online] [cit. Dec. 10, 2017]. Available from: https://www.ngs.noaa.gov/corbin/class_description/Nayegandhi_green_lidar.pdf.
6. CAMPBELL, James. *Introduction to remote sensing*. 5th ed. New York: Guilford Press, 2011. ISBN 9781609181765.
7. *Characteristics of Lidar Data* [online]. 2014 [cit. Jan. 24, 2017]. Available from: https://www.e-education.psu.edu/geog481/l1_p6.html.
8. ARCGIS. *What is lidar data?* [online]. 2016 [cit. Nov. 17, 2017]. Available from: http://desktop.arcgis.com/en/arcmap/10.3/manage-data/las-dataset/what-is-lidar-data-.htm.
9. GANEEV, Rashid A. *Laser - Surface Interactions*. Springer Science & Business Media, 2013. ISBN 9789400773417.
10. LIADSKY, Joe. *Introduction to LIDAR* [online]. 2007 [cit. Feb. 4, 2017]. Available from: http://nps.edu/Academics/Centers/RSC/documents/IntroductiontoLIDAR.pdf.
11. FLOOD, Martin (ed.). *ASPRS Guidelines Vertical Accuracy Reporting for Lidar Data* [online]. 2004 [cit. Feb. 4, 2017]. Available from: http://www.asprs.org/a/society/committees/lidar/Downloads/Vertical_Accuracy_Reporting_for_Lidar_Data.pdf.

12. LOESCH, Tim. *Understanding LiDAR Accuracy* [online]. 2013 [cit. Feb. 4, 2017]. Available from: http://www.mngeo.state.mn.us/cgi-bin/LiDAR/topic_show.pl?tid=29.
13. ROHRBACH, Felix. *FOUR ESSENTIAL LIDAR PARAMETERS* [online]. 2015 [cit. Feb. 5, 2017]. Available from: http://felix.rohrba.ch/en/2015/four-essential-lidar-parameters/.
14. *Limits of Resolution: The Rayleigh Criterion* [online] [cit. Feb. 5, 2017]. Available from: https://cnx.org/exports/f403618a-c8e1-4b20-8737-6164df697aea@5.pdf/limits-of-resolution-the-rayleigh-criterion-5.pdf.
15. GRIOT, Melles. Melles Griot Optics Guide [online]. 2002 [cit. Dec. 10, 2017]. Available from: https://web.archive.org/web/20110708214325/http://www.cvimellesgriot.com/products/Documents/TechnicalGuide/fundamental-Optics.pdf.
16. HECHT, Eugene. *Optics*. 4th ed. Guilford Press, 2001. ISBN 0805385665.
17. *Angular resolution and nominal range of Lidar-Lite* [online]. 2015 [cit. Feb. 15, 2017]. Available from: http://www.robotshop.com/forum/angular-resolution-and-nominal-range-of-lidar-lite-t12456.
18. *LAS SPECIFICATION VERSION 1.4 – R13* [online]. 2013 [cit. Feb. 15, 2017]. Available from: http://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf.
19. *Binary Point File 3 (BPF3), BPF Public License File Format Definition, Implementation Guide v1.1* [online]. 2015 [cit. Feb. 19, 2017]. Available from: https://nsgreg.nga.mil/doc/view?i=4220.
20. RUSU, R. B.; COUSINS, S. 3D is here: Point Cloud Library (PCL). 2011, pp. 1–4. ISSN 1050-4729. Available from DOI: 10.1109/ICRA.2011.5980567.
21. CLOUDCOMPARE. *CloudCompare – Presentation* [online] [cit. Dec. 4, 2017]. Available from: http://www.cloudcompare.org/presentation.html.
22. P. MARION; R. KWITT, B. Davis; GSCHWANDTNER, M. PCL and ParaView - Connecting the Dots. In: *CVPR Workshop on Point Cloud Processing (PCP)*. 2012.
23. LOWE, David G. Three-dimensional object recognition from single two-dimensional images. *Artificial Intelligence*. 1987, vol. 31, no. 3, pp. 355–395. ISSN 0004-3702. Available from DOI: http://dx.doi.org/10.1016/0004-3702(87)90070-1.

24. C, Hough Paul V. *Method and means for recognizing complex patterns*. [Cit. Oct. 31, 2017]. Available from: https://www.google.com/patents/US3069654.
25. FISCHLER, Martin A.; BOLLES, Robert C. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*. 1981, vol. 24, no. 6, pp. 381–395. ISSN 0001-0782. Available from DOI: 10.1145/358669.358692.
26. SMITH, Lindsay I. *A tutorial on Principal Components Analysis* [online]. 2002 [cit. Nov. 6, 2017]. Available from: http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf.
27. SVOBODA, David. *Filters in Image Processing Image Transforms* [online]. 2016 [cit. Nov. 7, 2017]. Available from: https://is.muni.cz/auth/el/1433/jaro2017/PA171/um/67947973/04-lecture.pdf.
28. DANIELSSON, Per-Erik. Euclidean distance mapping. *Computer Graphics and Image Processing*. 1980, vol. 14, no. 3, pp. 227–248. ISSN 0146-664X. Available from DOI: https://doi.org/10.1016/0146-664X(80)90054-4.
29. SOILLE, Pierre. *Morphological Image Analysis: Principles and Applications*. Springer, 2010. ISBN 9783642076961.
30. OPENCV. *Morphological Transformations* [online] [cit. Nov. 14, 2017]. Available from: https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html.

Datasheets and Manuals

M1. *RPLIDAR A2 Low Cost 360 Degree Laser Range Scanner – Introduction and Datasheet* [online]. 2016 [cit. Nov. 27, 2016]. Available from: http://www.slamtec.com/download/lidar/documents/en-us/LD204_SLAMTEC_rplidar_datasheet_A2M4_v1.0_en.pdf.

M2. *Sweep data sheet* [online]. 2016 [cit. Nov. 27, 2016]. Available from: https://s3.amazonaws.com/scanse/SWEEP_DATA_SHEET.pdf.

M3. *USER'S MANUAL AND PROGRAMMING GUIDE – HDL-64E S2 and S2.1* [online]. 2012 [cit. Nov. 27, 2016]. Available from: http://velodynelidar.com/docs/manuals/63-HDL64ES2h%20HDL-64E%20S2%20CD%20Users%20Manual.pdf.

M4. *USER'S MANUAL AND PROGRAMMING GUIDE – HDL-32E* [online]. 2016 [cit. Nov. 27, 2016]. Available from: http://velodynelidar.com/docs/manuals/63-9113%20REV%20K%20MANUAL,USER'S,HDL-32E_Outlined.pdf.

M5. *USER'S MANUAL AND PROGRAMMING GUIDE – VLP-16* [online]. 2016 [cit. Nov. 27, 2016]. Available from: http://velodynelidar.com/docs/manuals/63-9243%20Rev%20B%20User%20Manual%20and%20Programming%20Guide,VLP-16.pdf.

M6. *RobotEye RE05 3D LIDAR 3D Laser Scanning System Product Datasheet* [online]. 2014 [cit. Dec. 8, 2016]. Available from: http://www.ocularrobotics.com/wp/wp-content/uploads/2015/12/RobotEye-RE05-3D-LIDAR-Datasheet.pdf.

M7. *RobotEye RE08 3D LIDAR 3D Laser Scanning System Product Datasheet* [online]. 2015 [cit. Dec. 8, 2016]. Available from: http://www.ocularrobotics.com/wp/wp-content/uploads/2016/08/RobotEye-RE08-3D-LIDAR-Datasheet.pdf.

M8. *360° 3D LIDAR M8-1 SENSOR* [online]. 2015 [cit. Dec. 11, 2016]. Available from: http://www.lidarusa.com/uploads/5/4/1/5/54154851/quanergy_m8-1_lidar_datasheet_v4.0.pdf.

M9. *Scanning Laser Range Finder UTM-30LX/LN Specification* [online]. 2012 [cit. Dec. 15, 2016]. Available from: http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/UTM-30LX_spec_en.pdf.

M10. *Scanning Scanning canning Laser Range Finder Laser Range Finder URG-04LX Specifications* [online]. 2005 [cit. Dec. 15, 2016]. Available from: http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_spec_en.pdf.

M11. *Scanning Laser Range Finder URG-04LX-UG01 (Simple-URG) Specifications* [online]. 2009 [cit. Dec. 16, 2016]. Available from: http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_UG01_spec_en.pdf.

M12. *Scanning Laser Range Finder Smart-URG eco UST-05LA Specification* [online]. 2014 [cit. Dec. 16, 2016]. Available from: https://www.hokuyo-aut.jp/02sensor/07scanner/download/UST-05LA_spec_en.pdf.

M13. *Scanning Laser Range Finder Smart-URG mini UST-10LX (UUST003) Specification* [online]. 2014 [cit. Dec. 16, 2016]. Available from: https://www.hokuyo-aut.jp/02sensor/07scanner/download/UST-10LX_spec_en.pdf.

M14. *Scanning Laser Range Finder Smart-URG mini UST-20LX (UUST004) Specification* [online]. 2014 [cit. Dec. 16, 2016]. Available from: https://www.hokuyo-aut.jp/02sensor/07scanner/download/UST-20LX_spec_en.pdf.

M15. *RIEGL VUX-1UAV* [online]. 2016 [cit. Dec. 16, 2016]. Available from: http://www.riegl.com/uploads/tx_pxpriegldownloads/DataSheet_VUX-1UAV_2016-09-16_01.pdf.

M16. *RIEGL miniVUX-1UAV* [online]. 2016 [cit. Dec. 16, 2016]. Available from: http://www.riegl.com/uploads/tx_pxpriegldownloads/DataSheet_MiniVUX-1UAV_2016-10-05_01.pdf.

M17. *RIEGL VUX-1LR* [online]. 2016 [cit. Dec. 16, 2016]. Available from: http://www.riegl.com/uploads/tx_pxpriegldownloads/DataSheet_VUX-1LR_2016-09-16.pdf.

M18. *RIEGL VUX-1HA* [online]. 2015 [cit. Dec. 16, 2016]. Available from: http://www.riegl.com/uploads/tx_pxpriegldownloads/DataSheet_VUX-1HA__2015-10-06.pdf.

M19. *Raspberry Pi 3 Model B* [online] [cit. Mar. 15, 2017]. Available from: http://docs-europe.electrocomponents.com/webdocs/14ba/0900766b814ba5fd.pdf.

M20. *Banana PI BPI-M3 User Manual* [online] [cit. Mar. 15, 2017]. Available from: https://goo.gl/JiKnmp.

M21. *UP Specification* [online] [cit. Mar. 15, 2017]. Available from: http://up-shop.org/index.php?controller=attachment&id_attachment=1.

M22. *Parallella-1.x Reference Manual* [online]. 2014 [cit. Mar. 15, 2017]. Available from: http://www.parallella.org/docs/parallella_manual.pdf.

M23. *C.H.I.P. Pro Overview* [online] [cit. Mar. 15, 2017]. Available from: https://docs.getchip.com/chip_pro.html.

M24. *Introducing the Raspberry Pi Zero* [online]. 2017 [cit. Mar. 15, 2017]. Available from: https://cdn-learn.adafruit.com/downloads/pdf/introducing-the-raspberry-pi-zero.pdf.

M25. *NanoPi 2 Fire* [online] [cit. Mar. 15, 2017]. Available from: http://nanopi.io/nanopi2-fire.html.

M26. *Jetson TK1 Development Kit Specification* [online]. 2014 [cit. Mar. 15, 2017]. Available from: http://developer.download.nvidia.com/embedded/jetson/TK1/docs/3_HWDesignDev/JTK1_DevKit_Specification.pdf.

M27. *GIZMO 2 SPECIFICATIONS* [online] [cit. Mar. 15, 2017]. Available from: http://www.gizmosphere.org/wp-content/uploads/2014/11/4531_Gizmo2_ProBRIEF_FNL.Element14.pdf.

M28. *Intel Galileo Gen 2 Development Board* [online] [cit. Mar. 15, 2017]. Available from: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/galileo-g2-datasheet.pdf.

Appendices

A Hardware Comparison Tables

The following tables A.1 and A.2 compares suitable *LiDAR* scanners and micro computer boards, currently available on the market, that can be used for embedded system prototyping.

Model	Max. Distance	Min. Distance	Accuracy	Field of View	Angular Resolution	Frequency	Weight	Price
RPLiDAR A2	6 m	0.15 m	-	360° H	0.45°-1.35° H	5-15 Hz	0.19 kg	$400[1]
Sweep	40 m	0.1 m	1 cm	360° H	1.4°-7.2° H	2-10 Hz	0.12 kg	$349[2]
VLP-16	100 m	1 m	3 cm	360° H, 30° V	0.1°-0.4° H, 2° V	5-20 Hz	0.83 kg	$7999[3]
HDL-32E	70 m	1 m	<2 cm	360° H, 40° V	0.16° H, ~1.33° V	10 Hz	1.3 kg	$29900[4]
RE05 3D	160 m	-	0.5 cm	360° H, 70° V	0.01° H / V	15 Hz	3 kg	-
M8-1	150 m	-	<5 cm	360° H, 20° V	0.03-0.2° H / V	5-30 Hz	0.8 kg	$7950[5]
UTM-30LX	60 m	0.1 m	3-5 cm	270° H	0.25° H	40 Hz	0.21 kg	$4825[6]
URG-04LX	5.6 m	0.02 m	1 cm	240° H	0.36° H	10 Hz	0.16 kg	$1935[6]
URG-04LX-UG01	4 m	0.02 m	3 cm	240° H	0.36° H	10 Hz	0.16 kg	$1115[6]
UST-05LA	5 m	0.06 m	4 cm	270° H	0.5° H	50 Hz	0.13 kg	$1794[6]
UST-10LX	10 m	0.06 m	4 cm	270° H	0.25° H	40 Hz	0.13 kg	$1720[6]
UST-20LX	20 m	0.06 m	4 cm	270° H	0.25° H	40 Hz	0.13 kg	$2861[6]
VUX-1UAV	920 m	3 m	1 cm	330° H	0.001° H	10-200 Hz	3.5 kg	-
miniVUX-1UAV	250 m	3 m	1.5 cm	360° H	0.001° H	10-100 Hz	1.55 kg	-
VUX-1LR	1350 m	5 m	1.5 cm	330° H	0.001° H	10-200 Hz	3.5 kg	-
VUX-1HA	420 m	1.2 m	0.5 cm	360° H	0.001° H	10-250 Hz	3.5 kg	-

Table A.1: Comparison of selected laser scanners

1. https://www.dfrobot.com
2. http://scanse.io
3. http://velodynelidar.com/vlp-16.html
4. http://www.hizook.com/blog/2010/08/24/velodyne-hdl-32e-new-high-end-laser-rangefinder
5. http://quanergy.com/m8/
6. http://www.robotshop.com/

Model	CPU Freq.	CPU Cores	CPU Arch.	RAM	Peak Power	Size	Price	I/O
Raspberry Pi 3	1,2GHz	4	ARM	1Gb	720mA[1]	85,6x56,5mm	$26[2]	4xUSB, 40xGPIO, 100MB LAN
Banana Pi M3	1,6GHz	8	ARM	2GB	2300mA	92x60mm	$79[2]	2xUSB, 28xGPIO, 1Gb LAN [M20]
UP board	1,92GHz	4	x86-64	4GB	3000mA	85,6x56,5mm	$149[3]	4xUSB, 40xGPIO, 1Gb LAN [M21]
The Parallella Board	1GHz	2, 16 RISC	ARM	1Gb	390mA[4]	87x54mm	$149[2]	1xUSB, 24xGPIO, 1Gb LAN [M22]
C.H.I.P.	1GHz	1	ARM	512MB	300mA[5]	60x40mm	$9[5]	1xUSB, 80xGPIO [M23]
Raspberry Pi Zero	1GHz	1	ARM	512MB	310mA	65x30mm	$5[5]	1xMicroUSB, 26xGPIO [M24]
NanoPi 2 Fire	1,4GHz	4	ARM	1Gb	-	75x40mm	$29[6]	1xUSB, 40xGPIO, 1Gb LAN [M25]
NVIDIA Jetson TK1	2,32GHz	4 192 GPU	ARM	2GB	5000mA	127x127mm	$188[2]	2xUSB, 9xGPIO, RS232, 1Gb LAN [M26]
AMD Gizmo Board 2	1GHz	2	x86-64	2GB	2000mA	101.6x101.6mm	$169[7]	2xUSB, GPIO, PCIE, 1Gb LAN [M27]
Intel Galileo 2 Gen.	400Mhz	1	x86	256MB	379mA[8]	123,8x72mm	$42[2]	1xUSB, 20xGPIO, 100Mb LAN [M28]

Table A.2: Comparison of motherboards that are suitable for an embedded application.

1. http://raspi.tv/2016/how-much-power-does-raspberry-pi3b-use-how-fast-is-it-compared-to-pi2b
2. https://www.amazon.com
3. http://up-shop.org
4. http://www.adapteva.com/white-papers/measuring-power-consumption-of-epiphany-chip-on-parallella-prototype-board/
5. http://makezine.com/2015/11/28/chip-vs-pi-zero/
6. http://www.friendlyarm.com
7. http://www.newark.com
8. http://www.intel.com/content/www/us/en/support/boards-and-kits/intel-galileo-boards/000020085.html

B Content of Enclosed DVD

```
DVD
├── implementation ..............Implementation part of this thesis
│   ├── lidar_analyzer ... Analyser implementation, see appendix E
│   ├── lidar_connector  Connector implementation, see appendix D
│   ├── lidar_shared ........ Shared code for the 2D implementation
│   └── lidar_visualizer  Visualizer implementation, see appendix F
└── scans ................Scans downloaded or obtained with RPlidar
    ├── rplidar ..........................Scans obtained with RPlidar
    │   ├── 3Droom01-corridor.las
    │   ├── 3Droom02-bathroom.las
    │   ├── 3Droom03-kitchen.las
    │   ├── 3Droom04-livingroom.las
    │   ├── room01-office.las
    │   ├── room01-office.txt
    │   ├── room02-meetingroom.las
    │   ├── room02-meetingroom.txt
    │   ├── room03-kitchen.las
    │   ├── room03-kitchen.txt
    │   ├── room04-livingroom.las
    │   └── room04-livingroom.txt
    └── thirdpart ............................... Downloaded scans
        ├── ny-uav.laz
        └── ny-uav-subsampled.laz
```

C 3D Implementation Usage

This appendix describes the usage of *Matlab* script that provides an implementation of the algorithm described in Chapter 5.

C.1 Requirements

To run the script, we need to have the standard installation of *Matlab* together with image processing and analysis toolbox called *DIPimage*. To get the *DIPimage* toolbox, we need to download the toolbox from `http://www.diplib.org/download`. Windows users can download the automatic installation package. Linux users have to install the toolbox manually. The steps are following:

1. Unzip *.tbz* archive into some folder; we have chosen *~/.matlab/plugins*. So the command in such case would be

   ```
   mkdir ~/.matlab/plugins && tar -xf dipimage.tbz -C ~/.matlab/plugins
   ```

2. Add *LD_LIBRARY_PATH=<dip_image_location>/Linuxa64/lib* variable (for 64-bit Linux, for 32-bit Linux replace *Linuxa64* by *Linux*) to *PATH* or edit the *Matlab* launcher to use this variable.

Finally, for both Windows and Linux, we have to initialize the library in *Matlab*. It is possible by running following commands in *Matlab* command line:

```
addpath('<dip_image_location>/common/dipimage')
dip_initialise
```

To automate this initialization of *DIPimage* toolbox, we can add the initialization section mentioned above to *Matlab* startup file *startup.m*. This file can be located in any *Matlab* search path and will be loaded automatically.

C.2 Running the Script

Matlab implementation is located in the enclosed DVD, in folder *implementation/matlab*. The main file is called *plane_estimation.m* which is the main script, which runs the analysis on file *3Droom01-corridor.las*.

The analyzed file can be changed in constant $FILE\_TO\_ANALYZE$, other configuration variables are $SCALING\_FACTOR$ (for scaling factor), $ED\_TRANSFORM\_SIZE$ (size of the *Euler Distance* transformation), and $KERNEL\_RADIUS$ (for circular structuring element radius).

The script opens three figure windows – plane position in the point cloud, detected points above and under the plane together with suitable detected areas separately and in the whole point cloud. The script cycles through all detected planes, after each plane the user input (pressing any key) is required.

D Lidar Connector Usage

D.1 Compilation

To compile the *Connector*, we need to have *cmake* and *c++* compiler that supports C++ 14 library installed. In lidar_connector folder (this folder can be found in the enclosed DVD, in folder implementation), we have to run following command:

```
mkdir bin && cd bin && cmake ../
```

The command above creates folder bin and the *Connector* itself is located in binary file *lidar_connector*.

D.2 Usage

Running the program with parameter *-h* results in the following output

```
$ ./lidar_connector -h
Usage: ./lidar_connector [OPTIONS]
-p <port>, --port=<port>      Port number that will be used as port for socket host
-a <addr>, --address=<addr>   Address that will be used as an address for socket host,
                              if the port is not provided sockets will not be used,
                              if the address is not provided "localhost" will be used
-d <port>, --device=<port>    A lidar device port. Default is "/dev/ttyUSB0"
-l <file>, --log=<file>       Filename of log file. When has a ".las" suffix,
                              the LAS file format is used, otherwise uses plain text.
                              When saving to ".las" file, the --single-run is forced.
-v, --verbose                 Whether more verbose output should be used.
-q, --quiet                   Whether quiet output mode should be used, has higher
                              priority than verbose.
-s, --single-run              Flag that makes the LiDAR device to run only one scan cycle.
-i <file>, --input=<file>     Input file with a LiDAR data. The data is expected to be in format
                              - one line per each iteration
                              - point by point in format "<quality>,<distance>,<angle>;"
                              separated by a comma, ending with semicolon
-m, --avg                     Number of iterations to be used for moving average
                              (accepted values are > 0)
-h, --help                    Print this help and exit
```

To connect to device on */dev/ttyUSB0* and open port 123456 for listening we need to run following command

```
./lidar_connector -p 123456
```

To keep the same configuration as mentioned above and apply moving average for last 5 scans we need to run following command

```
./lidar_connector -p 123456 -m 5
```

To connect to device on */dev/ttyUSB0* and log scanned data to *output.las* file we need to run following command

```
./lidar_connector -l output.las
```

To read data from file *data.txt* with continuous *Lidar* data format and open port 123456 for sending such data we need to run following command

```
./lidar_connector -p 123456 -i data.txt
```

E Lidar Analyzer Usage

E.1 Compilation

To compile the *Analyzer*, we need to have *cmake*, *c++* compiler that supports C++ 14 and *Eigen3* library installed. In lidar_analyzer folder (this folder can be found in the enclosed DVD, in folder implementation), we have to run following command:

```
mkdir bin && cd bin && cmake ../
```

The command above creates folder bin and the *Analyzer* itself is located in binary file *lidar_analyzer*.

E.2 Usage

Running the program with parameter *-h* results in the following output

```
$ ./lidar_analyzer -h
Usage: ./lidar_analyzer [OPTIONS]
-p, --port=<port>               Port number that will be used as port for socket host
-a, --address=<address>         Address that will be used as an address for socket host,
                                if the port is not provided sockets will not be used,
                                if the address is not provided "localhost" will be used
-e, --server-port=<port>        Same as a -p but used for connecting to remote server
-s, --server-address=<port>     Same as a -a but used for connecting to remote server
-v, --verbose                   Whether more verbose output should be used.
-q, --quiet                     Whether quiet output mode should be used, has higher
                                priority than verbose.
-h, --help                      Print this help and exit
```

To read data from port 123456 and open port 12345 for listening we need to run following command

```
./lidar_analyzer -e 123456 -p 12345
```

F Lidar Visualizer Usage

F.1 Compilation

To compile the *Visualizer*, we need to have *cmake*, *c++* compiler that supports C++ 14 and *Irrlicht* library installed. In lidar_analyzer folder (this folder can be found in the enclosed DVD, in folder implementation), we have to run following command:

```
mkdir bin && cd bin && cmake ../
```

The command above creates folder bin and the *Visualizer* itself is located in binary file *lidar_visualizer*.

F.2 Usage

Running the program with parameter *-h* results in the following output

```
$ ./lidar_visualizer -h
Usage: ./lidar_visualizer [OPTIONS]
-p, --port=<port>               Port number that will be used as port for connecting
                                to socket host
-a, --address=<address>         Address that will be used as an address for connecting
                                to socket host, if the port is not provided
                                sockets will not be used, if the address is not provided
                                "localhost" will be used
-v, --verbose                   Whether more verbose output should be used.
-q, --quiet                     Whether quiet output mode should be used,
                                has higher priority than verbose.
-h, --help                      Print this help and exit
```

To read data from port 123456 we need to run following command

```
./lidar_visualizer -p 123456
```

The *Visualizer* also contains an user interface that allows user to filter visualized data by type, quality, *RMSE*, color or size.