TAMPERE UNIVERSITY OF TECHNOLOGY

FATEMEH ALSADAT ESTIRI

3D Object Detection and Tracking Based On Point Cloud Library Special Application In Pallet Picking For Autonomous Mobile Machines

Thesis work

Examiners: Prof. Kalevi Huhtala
Dr. Reza Ghabcheloo
Examiner and topic approved by
The Faculty Council of the
Faculty of Engineering Sciences
9th April 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
International Master's Degree Programme in Machine Automation Technology
ESTIRI, FATEMEH ALSADAT : 3D Object Detection and Tracking Based On Point Cloud Library, Special Application In Pallet Picking For Autonomous Mobile Machines
Master of Science Thesis, 96 pages, 0 Appendix pages
February 2014
Major: Mechatronics Engineering
Examiners: Professor Kalevi Huhtala, Dr. Reza Ghabcheloo
Keywords: 3D object recognition, point cloud library, PCL, object tracking, pallet picking, 3D visual servoing

This work covers the problem of object recognition and pose estimation in a point cloud data structure, using PCL (Point Cloud Library). The result of the computation will be used for mobile machine pallet picking purposes, but it can also be applied to any context that requires finding and aligning a specific pattern.
The goal is to align an object model to the visible instances of it in an input cloud. The algorithm that will be presented is based on local geometry descriptors that are computed on a set of uniform key points of the point clouds. Correspondences (best matches) between such features will be filtered and from this data comes a rough alignment that will be refined by ICP algorithm. Robust dedicated validation functions will guide the entire process with a greedy approach. Time and effectiveness will be discussed, since the target industrial application imposes strict constraints of performance and robustness.
The result of the proposed solution is really appreciable, since the algorithm is able to recognize present objects, with a minimal percentage of false negatives and an almost zero false positives rate. Experiments have been conducted on datasets acquired from a state-of-the-art simulator and some sample scene from the real environment.

PREFACE

This thesis was performed as part of the GIM project in IHA department at Tampere University Of Technology. Hereby I would like to thank my great supervisor Dr.Reza Ghabcheloo for always being supportive and advising me so patiently even in hard times.
My second thanks should go to all department members who warmly welcomed me and aided me through the process, specially Mika Hyvönen who was always so kind and helpful.
Finally the ones without whom I could not have been in such a position, my truly patient and always supportive husband Behnam, and my parents Ali and Masoumeh who raised me strong and blessed me with their prayers.

Fatemeh Alsadat Estiri

February 11, 2014

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

PCL	Point Cloud Library
PCD	Point Cloud Data
TOF	Time Of Flight
LIDAR	LIght Detection And Ranging
DOF	Degree Of Freedom
GUI	Graphical User Interface
FLANN	Fast Library for Approximate Nearest Neighbours
VTK	Visualization Tool Kit
UDP	User Datagram Protocol
IP	Internet Protocol
PFH	Point Feature Histograms
FPFH	Fast Point Feature Histograms
SHOT	Signature of Histograms of OrienTations
RANSAC	RANdom SAmple Consensus
ICP	Iterative Closest Point

1. INTRODUCTION AND SYSTEM SPECIFICATIONS

1.1 Overview

The science of robotics is a creative activity involving state of the art manipulation of different branches of science. Although robotics has achieved great success to date in the world of industrial manufacturing, it still has a long way ahead and fundamental problems to solve. Among all different types of robotic applications, autonomous mobile ones are the ones with the most complicity, as they will need great level of intelligence to overcome various difficulties and handle all kind of situations in harsh and non-ideal environments of the real world.
This mission will not be completed without the robot being able to "see" or "sense" its surrounding. New achievements in sensor technologies has played significant role with this regard. With the advent of different 3D laser scanners it is now possible to capture fast and safe 3D data with great detail and accuracy. On the other hand, processing data received from these sensors has also opened a new field in science. Specifically for laser scanner data processing there is great demand, as this information will not have application only in robotics, but also in urban and city projects [8], medical [9], computer games [16] and industrial marketing will also benefit from the results.
Thus we will tackle one of the most useful applications in mobile robots. Section 1.3 introduces the GIM fork lifter located in Tampere University of Technology as the mobile robot on which we aim to manipulate and test the results, currently using a 2D camera for EUR-pallet detection and tracking. In this project the main focus will be on recognition and pose estimation of a standard EUR-pallet which is introduced in section 1.2, but the results can be easily adapted to any kind of object tracking problem, with understanding the algorithm and replacing datasets with desired ones.
In section 1.4 of this chapter some benefits of 3D analysis as opposed to 2D is discussed by expressing examples revealing some shortcomings of 2D processing. Object detection and tracking will use a servo controlled laser scanner to gather data from the environment. In section 1.5 this main data acquisition equipment is introduced and its properties are described briefly. This introduction is necessary to

understand the platform on which we conducted our final tests. The main output of such introduced sensors is a collection of triplets containing coordinates of a point in 3D environment, which is called a point cloud. We will have a closer look at the definition of this term in section 1.6. Following this introduction, the most important tool of implementing this project will be introduced in section 1.7. The Point Cloud Library (PCL) is introduced as a standalone, large scale, open source project for point cloud processing containing numerous state-of-the art algorithms related to 3D data processing. We will end this chapter with a brief introduction on file format of this type of data in section 1.8.

1.2 EUR-Pallet

Euro pallet is the standard European pallet as specified by the European Pallet Association (EPAL)[1]. The European pallet is a 1200*800*144 mm four-way pallet made of wood that is nailed with special nails in a prescribed pattern. Following the standardization, most of the European industries switched over to use Euro-pallets with trucks, forklifts and high-rack warehouses optimized for their size.
Using a standard size and shape pallet optimized for using in transformation tasks, reduces supply chain costs and increases the productivity. Necessary industry adjustments required for the implementation of standard pallet size(s), and all relevant industries follow the standard nowadays. Due to this fact it is necessary to have pallet recognition algorithms which would benefit a wide range of users. Though all pallets have the same outer size but they may vary in inner dimensions. They may also undergo serious damages due to usage, different climate conditions, and decaying. Figure 1.1 shows a standard Euro pallet.

Figure 1.1: Euro standard pallet

[1] http://www.epal-pallets.org/uk/home/main.php

1.3 GIM Mobile Machine

GIM is a research centre focused on intelligent mobile machines and robotics. Its background is in the strong research tradition of the participating institutes in this area and the urgent need from industry. According to the project official web page[2], GIM is formed by a team of researchers from the Department of Automation and Systems Technology at Aalto University and from the Institute of Hydraulics and Automation (IHA) at Tampere University of Technology and its goal is to research and develop methods and technologies for future intelligent mobile machines and field and service robots. GIM started operations on January 2008.
Currently there are several machines under development in this project. The so called "GIM-Machine" was built completely within GIM and will answer the research needs even further to future and among other things provides a platform for research and development of methods and technologies for future intelligent vehicles.

Figure 1.2: GIM mobile machine

The GIM machine is currently located at Tampere University of Technology mobile hall and it is accessible for research purposes. It is also equipped with a SICK[3] laser scanner as the main device for capturing 3D data from the environment. The required specifications and details are given as we proceed to implement our project.

[2]http://gim.aalto.fi/
[3]http://www.sick.com/

1.4 2D Versus 3D

The motivation of this field of science relies on the answer to this question "why do we need 3D at all?", which might be simple and complicated at the same time. In short we can say because the world is in 3D. Framing the 3D surrounding environment to 2D data has its own limitations and might not completely reflect the semantics needed to implement sufficient enough intelligence on mobile machines.
A brief comparison between 2D and 3D methods is given in [1] which is worth reviewing here, as it illustrates the issue by giving some examples in which unlike 3D, 2D is not successful producing meaningful results. One of the first short comings in 2D processing is related to limitations in the data stream itself. In 2D image processing we usually rely so much on the conditions of the received data and the environment. Many cameras fail to represent precise enough information when there is not enough ambient lighting, or when the contrast of the scene is not so high. Although this issue will be addressed in time, as technology progresses and better camera sensors are developed, but still there might be problems due to objects being in the shadow of other objects. An example of such a deficiency is shown in Figure 1.3, where due to the low dynamic range of the camera sensor, the right part of the image is completely underexposed. This makes it very hard for 2D image processing applications to recover the necessary information for recognizing objects in such scenes.

Figure 1.3: 2D recognition failure due to underexposed parts, taken from [1]

Now assume an object present in a scene and an image of that object printed on another object. If the third dimension of the real world is not taken into consideration, there would be no difference between these two objects if recognized in 2D. Figure 1.4 reveals this issue by an example. The right part shows a 2D algorithm applied with a successful recognition. This basically means that the algorithm result suggests a presence of the object and if this result is to be fed to a robot grabbing arm, it will be obviously a fail, since zooming out of the figure shows clearly that

the semantics has been taken in the wrong way. This is a clear example which shows that the semantics of a particular solution obtained only using 2D image processing can be lost if the geometry of the object is not incorporated in the reasoning process.

(a) Recognition result (b) Zoomed out

Figure 1.4: 2D recognition failure due to semantics loss, taken from [1]

Whereas the advantages of using 3D based data acquisition system might be enormous. Object recognition in 3D easily avoids lots of segmentation problems in 2D, where occlusions only happen in certain angles. In addition, 3D enables measurement of real size of an object and to define scale and ratio for the environment, which has variety of applications such as measuring the volumes of packages, vehicles or pallets, and volume flow measurement for bulk materials to name a few[2]. 3D and 2D can as well be combined easily, making 3D object recognition always superior than 2D recognition and benefit from both advantages.

1.5 Laser Scanner

Though there are many ways of measuring distances and converting them to 3D points, in the context of mobile robotic applications, one of the most used approaches is Time-of-Flight (TOF) systems[3], which measures the delay until an emitted signal hits a surface and returns to the receiver, thus estimating the true distance from the sensor to the surface. This category of systems involves sensing devices such as a Laser Measurement System (LMS) or LIDAR, radars, Time-of-Flight (TOF) cameras, or sonar sensors, which send "rays" of light (for example laser) or sound (for example sonar) in the world, which will then reflect and return to the sensor. Knowing the speed with which a ray propagates and using precise circuitry to measure the exact time when the ray was emitted and when the signal was returned, the distance can be estimated easily [1].
The laser scanner available on our target machine is a SICK LMS111 electro-optical laser measurement system shown in Figure 1.5. The LMS scans the perimeter of its

surroundings electro-sensitively in a plane with the aid of laser beams. According to the product's manual[4] the LMS measures its surroundings in two-dimensional polar coordinates. If a laser beam is incident on an object, the position is determined in the form of distance and direction. LMS111 has a range of 20 m (with 13 percent object remission), a resolution of 0.25 degree, and 25 Hz frequency and is fixed on the machine body, rotating by the means of a servo controlled motor.

Figure 1.5: SICK LMS111 laser scanner

1.6 Point Clouds

A point cloud, as its name suggests, is a set (cloud) of points in some coordinate system. In a three dimensional coordinate system, these points are usually defined by x, y, and z coordinates, and are often intended to represent the external surface of an object. With this definition, each point can be considered as a triplet in the form $p_i = \{x_i, y_i, z_i\}$ and a cloud consisting of n points can be written as $P = \{p_1, p_2, ..., p_n\}$ to represent 3D information about the world.
Point clouds may be created by actual devices such as laser scanners, stereo cameras, and time of flight cameras or produced synthetically via different simulations, software and processes. These devices measure a large number of points on the surface of an object in an automatic way, and often output a point cloud as a data file. The point cloud represents the set of occupied points in the space detected by the measuring device. As the result of a 3D scanning process point clouds are used for many purposes, including creating 3D CAD models for manufactured parts,

[4]https://mysick.com/saqqara/pdf.aspx?id=im0031331& lang=en& page=1

quality inspection, and a multitude of visualization, animation, rendering and mass customization applications.
It makes no sense to refer to a point cloud with actual coordinate values, without mentioning about the point of origin. The $\{x_i, y_i, z_i\}$ coordinates of any point p_i in P are given with respect to a predefined coordinate system, which will depend on the implementation and definition of the system and will be unique for each application. Therefore it is important to know what process has been performed in order to transfer a collection of measured distances into coordinates of points with respect to a coordinate frame. This will be discussed in more details in section 3.7 of this report.

1.7 The Point Cloud Library (PCL)

The Point Cloud Library (PCL)[5] is a standalone, large scale, open project for point cloud processing containing numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting, segmentation, tracking, recognition, and many more. These algorithms can be used, for example, to filter outliers from noisy data, stitch 3D point clouds together, segment relevant parts of a scene, extract key points and compute descriptors to recognize objects in the world, and create surfaces from point clouds and visualize them, to name a few[6].
Point Cloud Library is a fully template modern C++ library written with efficiency and performance on modern CPUs in mind. In order to support applications that require real time point cloud processing, PCL has been designed to take advantage of SSE instructions when available, and a GPU interface in association with NVIDIA. PCL is released under the terms of the BSD license and is open source software. It is free for commercial and research use. Development of PCL is a large collaborative effort driven by researchers and engineers from many different institutions and companies around the world.
The project is financially supported by Open Perception, Willow Garage, NVidia, Google, Toyota, Trimble, Urban Robotics, Honda Research Institute, Sandia Intelligent Systems and Robotics, Dinast, Optronic, Velodyne, CogniMem Technologies, Fotonic, and Ocular Robotics. This library aims to unite the field of point cloud processing, by providing an extensible framework for all the geometric algorithms necessary for 3D perception, PCL enables developers to create applications limited by their imaginations, rather than their 3D geometric knowledge. Version 1.0.0 has been released on May 2011 and until today the project has reached a good level of maturity, since the library is progressively more and more used in academic and

[5] http://pointclouds.org/
[6] most of the text in this section has been inherited from the official website

Figure 1.6: PCL logo

industrial application.
To simplify development, PCL is split into a series of smaller code libraries that can be compiled separately. This modularity is important for distributing PCL on platforms with reduced computational or size constraints. PCL presents an advanced and extensive approach to the subject of 3D perception, and it is meant to provide support for all the common 3D building blocks that applications need.

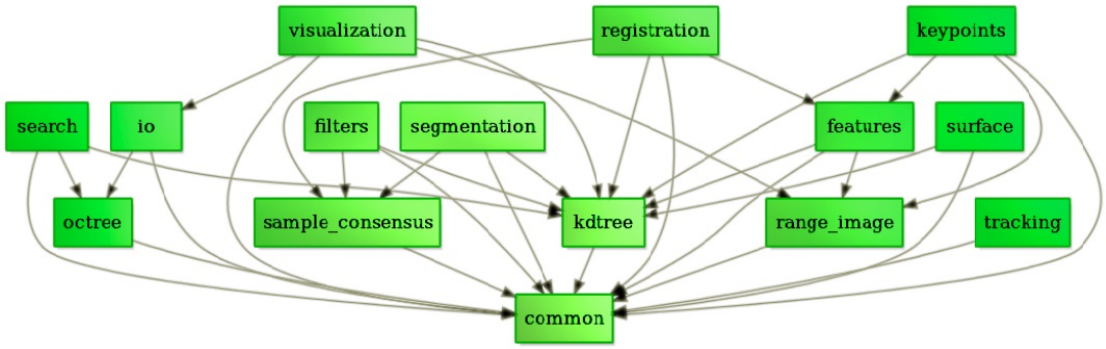


Figure 1.7: Point cloud library modules

In version 1.6 of the framework there are several modules[18]:
✻ Common: contains the common data structures and methods used by the majority of PCL libraries. The core data structures include the Point Cloud class and a multitude of point types that are used to represent points, surface normals, and RGB color values and feature descriptors. It also contains numerous functions for computing distances/norms, means and covariance, angular conversions and geometric transformations.
✻ Features: contains data structures and mechanisms for 3D feature estimation from point cloud data. 3D features are representations at a certain 3D point or position in space, which describe geometrical patterns based on the information available around the point. The data space selected around the query point is usually referred as the k-neighbourhood.
✻ Filters: contains outlier and noise removal mechanisms for 3D point cloud data

filtering applications. It also contains generic filters used to extract subsets of point cloud, or to exclude parts of it. It provides a voxel-grid class to down-sample a point cloud by intersecting it with a lattice of points.
* Geometry: reserved for future work, it will contain computational geometry data structures and algorithms.
* I/O: contains classes and functions for reading and writing point cloud data (PCD and PLY) files, as well as capturing point clouds from a variety of (OpenNI compatible) sensing devices.
* Kd-tree: provides the kd-tree data-structure, using FLANN implementation that allows for fast nearest neighbour searches. A Kd-tree (k-dimensional tree, in most cases in this work it is a 3d-tree) is a space partitioning data structure that stores a set of k-dimensional points in a tree structure that enables efficient range searches and nearest neighbour searches. Nearest neighbour searches are a core operation when working with point cloud data and can be used to find correspondences between groups of points or feature descriptors or to define the local neighbourhood around a point or points.
* Key points: contains implementations of several point cloud key point detection algorithms. Key points (also referred to as interest points) are points in an image or point cloud that are stable, distinctive, and can be identified using well-defined detection criteria. Key points and descriptors can be used to form a compact but distinctive representation of the original data. Harris, Narf, Sift and Uniform key points are implemented in the current version.
* Octree: provides efficient methods for creating a hierarchical tree data structure from point cloud data. This enables spatial partitioning, down sampling and search operations on the point data set. Each octree node has either eight children or no children. The root node describes a cubic bounding box which encapsulates all points. At every tree level, this space becomes subdivided by a factor of 2 which results in an increased voxel resolution.
* Registration: contains many point cloud registration algorithms for both organized an un-organized (general purpose) datasets, including ICP, correspondence finding and rejection, transformation estimators.
* Sample-consensus: holds SAmple Consensus (SAC) methods like RANSAC and models like planes and cylinders. These can combine freely in order to detect specific models and their parameters in point clouds. Some of the models implemented in this library include: lines, planes, cylinders, and spheres. Plane fitting is often applied to the task of detecting common indoor surfaces, such as walls, floors, and table tops. Other models can be used to detect and segment objects with common geometric structures.
* Search: provides methods for searching for nearest neighbours using different data

structures, including kd-trees, octrees, brute force and specialized search for organized datasets.
✻ Segmentation: contains algorithms for segmenting a point cloud into distinct clusters. These algorithms are best suited to processing a point cloud that is composed of a number of spatially isolated regions. In such cases, clustering is often used to break the cloud down into its constituent parts, which can then be processed independently. It also contains algorithms to find differences between two point cloud, that can be used for example for quality inspection purposes.
✻ Surface: deals with reconstructing the original surfaces from 3D scans. Depending on the task at hand, this can be for example the convex/concave hull, a mesh representation or a smoothed/re-sampled surface with normals. Creating a convex or concave hull is useful for example when there is a need for a simplified surface representation or when boundaries need to be extracted. Meshing is a general way to create a surface out of points which algorithms are based on marching cubes. Smoothing and re-sampling can be important if the cloud is noisy, or if it is composed of multiple scans that are not aligned perfectly. The complexity of the surface estimation can be adjusted, and normals can be estimated in the same step if needed.
✻ Visualization: this library was built for the purpose of being able to quickly prototype and visualizes the results of algorithms operating on 3D point cloud data. Similar to OpenCV's highgui routines for displaying 2D images and for drawing basic 2D shapes on screen. The package makes use of the VTK library for 3D rendering for range image and 2D operations. Point clouds, normals, range images, correspondences can be added to the viewer window.

1.8 PCD File Format

PCL has its own file format for point cloud data saving with the extension ".PCD". PCD is a simple file format for storing multi-dimensional point data which consists of a text header with the fields described below, followed by the data in ASCII or Binary.
✻ VERSION: The PCD file version (usually .7)
✻ FIELDS: Name of each dimension/field that a point can have (x, y, z, colour, intensity, ...)
✻ SIZE: The size of each dimension in bytes (for example a float is 4 bytes)
✻ TYPE: The type of each dimension as a char (I = signed, U = unsigned, F = float)
✻ COUNT: Number of elements in each dimension (x, y, or z would only have 1, but a histogram would have N)
✻ WIDTH: The width of the point cloud

✻ HEIGHT: The height of the point cloud
✻ VIEWPOINT: An acquisition viewpoint for the points. This could potentially be later used for building transforms between different coordinate systems, or for aiding with features such as surface normals, that need a consistent orientation. The viewpoint information is specified as a translation (t_x t_y t_z) + quaternion (q_w q_x q_y q_z). The default value is 0 0 0 1 0 0 0
✻ POINTS: Total number of points in the cloud
✻ DATA: The data type that the point cloud data is stored in (ascii or binary)
Each line consists of elements indicated by "fields" in the header which may only have x, y, z or contain RGB, RGBA, intensity, normal, curvature, and even user defined other data as well to suit the needs. In our case we will not be dealing with colours, so our fields will only contain the coordinate information of a point. Figure 1.8 shows how a point cloud PCD file header may look like.

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
0.81915 0.32 0 4.2108e+06
0.97192 0.278 0 4.2108e+06
0.944 0.29474 0 4.2108e+06
0.98111 0.24247 0 4.2108e+06
0.93655 0.26143 0 4.2108e+06
0.91631 0.27442 0 4.2108e+06
0.81921 0.29315 0 4.2108e+06
0.90701 0.24109 0 4.2108e+06
0.83239 0.23398 0 4.2108e+06
0.99185 0.2116 0 4.2108e+06
```

Figure 1.8: PCD file format header

According to [18] PCD is not the first file type to support 3D point cloud data, but it is meant to complement existing file formats that for one reason or another did not/do not support some of the extensions that PCL brings to n-D point cloud processing. The computer graphics and computational geometry communities in particular, have created numerous formats to describe arbitrary polygons and point clouds acquired using laser scanners. Some of these formats include:
PLY - a polygon file format, developed at Stanford University by Turk et al
STL - a file format native to the stereo lithography CAD software created by 3D

Systems
OBJ - a geometry definition file format first developed by Wave front Technologies
X3D - the ISO standard XML-based file format for representing 3D computer graphics data and many others
All the above file formats suffer from several shortcomings, which is natural as they were created for a different purpose and at different times, before today's sensing technologies and algorithms had been invented. According to PCL official website, having PCD as (yet another) file format can be seen as PCL suffering from the "not invented here" syndrome. In reality, this is not the case, as none of the above mentioned file formats offers the flexibility and speed of PCD files. Some of the clearly stated advantages include:
* The ability to store and process organized point cloud datasets. This is of extreme importance for real time applications, and research areas such as augmented reality and robotics.
* Binary mmap/munmap data types are the fastest possible way of loading and saving data to disk.
* Storing different data types (all primitives supported: char, short, int, float, double) allows the point cloud data to be flexible and efficient with respect to storage and processing. Invalid point dimensions are usually stored as NAN types.
* n-D histograms for feature descriptors. This is very important for 3D perception/computer vision applications.
An additional advantage is that by controlling the file format, we can best adapt it to PCL, and thus obtain the highest performance with respect to PCL applications, rather than adapting a different file format to PCL as the native type and inducing additional delays through conversion functions.
Among different properties forming a PCD file, two are of great importance for time sensitive applications. First is storing "organized" data instead of "un-organized" whenever possible. An organized point cloud dataset is the name given to point clouds that resemble an organized image or matrix like structure, where the data is split into rows and columns. Examples of such point clouds include data coming from stereo cameras or Time Of Flight cameras. The advantages of an organized dataset is that by knowing the relationship between adjacent points, nearest neighbour operations are much more efficient, thus speeding up the computation and lowering the costs of certain algorithms in PCL.
The second is using Binary instead of ASCII data types. Although ASCII PCD format is human readable, can be edited manually, and can be exchanged through all machine architectures, but Binary has more advantages when it comes to time consumption; it is very fast since one mmap call is used, usually has smaller size, is easily exchangeable (for example between OpenNIGrabber, PCDGrabber, ON-

IGrabber), is extensible allowing all kind of data to be published, and supports streaming devices (OpenNIGrabber) as well as triggered devices (PCDGrabber). Thus we will be using organized Binary type point clouds in our implementation whenever applicable.

2. THEORETICAL BACKGROUND

2.1 Overview

The problem of aligning various 3D point cloud data views is known as registration and its goal is to find the relative positions and orientations of the separately acquired views such that the intersecting areas between them overlap perfectly. The work presented here is motivated by finding correct point-to-point correspondences in real-world noisy data scans, and estimating rigid transformations that can rotate and translate each individual one. Section 2.2 of this chapter defines this subject more clearly.

To understand the geometry around a query point, most geometric processing steps need to discover a collection of neighbouring points that represent the underlying scanned surface through sampling approximations. The registration system therefore needs to employ mechanisms for enabling the search of point neighbours in fast ways, without re-computing distances between each other every time. A solution already implemented in PCL to tackle this issue is presented in section 2.3 and described in more details.

Environment surfaces are usually scanned giving a very high number of points that have similar characteristics, when described with a typical 3D descriptor. Using all these points would mean a lot of computation and memory usage, because every point needed to be described and inserted into the model. In addition tests have shown that using all these similar points leads to false recognitions because every planar surface casts many votes [17]. Thus a pre-process is necessary. This is done in different ways addressed in section 2.4.

The word "feature" can have many different meanings, but in PCL it is defined as a vector based on each points local neighbourhood, or sometimes a single vector for the whole point cloud. Feature vectors can be anything from simple surface normals to the complex feature descriptors needed for registration or object detection. By including the surrounding neighbours, the underlying sampled surface geometry can be inferred and captured in the feature formulation, which contributes to solving the ambiguity of comparison. Section 2.5 attempts to address some of these related initiatives and to explain the differences between them and their role in our proposed approach.

The word "correspondence" which will be widely used in this report identifies a set

of paired points across the given training shape instances which have been proposed to represent the same point according to comparison results between their features. But not all found correspondences are correct due to different reasons, and thus need to be filtered to reject those not fulfilling specific requirements. Finding these pairs and different filters to reject the bad ones are topics for section 2.6.
Finally based on the remaining correspondences the algorithm calculates a rigid transformation and the output will be a translation matrix along with a single float number indicating the "goodness" of the matching result. Section 2.7 will attempt to introduce different approaches to finding a validity score and represent the existing methods as functions in PCL to perform the task.
When the initial rough alignment has been performed, the algorithm switches to the final procedure that is done by the ICP algorithm, already implemented in PCL library. ICP is a great sharp point to point registration algorithm usually used for refining other algorithms results. In section 2.8 we will exactly explain how and why this implementation is used.
On that note we will end the second chapter, having equipped ourselves with a very brief, yet useful theoretical background and useful tools to step into the next level. Before moving to the implementation phase, a flowchart is also provided in section 2.9 to give a more clear view of the scenario, from the first step and towards the final goal. In addition, to avoid repetition, in each section after describing the principals the exact piece of final code related to that part is pasted. This will provide the clear understanding of how these tools are actually implemented and should be used in PCL, as well as explaining the code piece by piece at the same time.

2.2 Registration Definition

Registration can be simply defined as the problem of finding corresponding points on two different point clouds for the purposes of object recognition, tracking and finding the transformation matrix which will align one point cloud to another. Registration is a very common and basic technique for combining several data sets into one global consistent model. Using these techniques and algorithms provided by literature, it is also possible to align a single object to a bigger scene. In this case we call it object recognition problem, but the basic steps are very similar to registration.
The input to a general registration algorithm is two point cloud data sets either taken from a single environment from two different points of view, or an object model and a scene containing that object (object recognition problem). The output of the algorithm is a set of M roto-translation matrices M_i in the form $M = \{M_1, M_2, ..., M_n\}$ used to align the object point cloud to the most part of the visible objects in the

world point cloud (n objects). This kind of rigid transformation is a 4*4 matrix in the form of

$$M = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_{11} \\ r_{21} & r_{22} & r_{23} & t_{21} \\ r_{31} & r_{32} & r_{33} & t_{31} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

which is composed of a 3*3 rotation matrix R and a translation 3D vector T

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \qquad T = \begin{bmatrix} t_{11} \\ t_{21} \\ t_{31} \end{bmatrix}$$

With each matrix there is also a validation score v_i which is the result of different validation functions, and measures the perfectness of the alignment, as will be described in its relevant section. Hence, each alignment provided by this algorithm is described by the couple $\{M_i, v_i\}$; the matrix and its validation score. This matrix is the main output, but of course other outputs could be personalized by the programmer to visualize the result of recognition, compute other consequent outputs or show some text on the screen.

2.3 Point Search Methods

Most of the processing methods and specially pre-processing methods need to discover and go through a collection of neighbouring points that represent the underlying surface, in order to understand the geometry around a query point. Therefore we need to employ mechanisms for enabling the search of point neighbours in fast ways, without re-computing distances between two points every time.
A solution available in PCL is to use spatial decomposition techniques such as "kd-trees" or "octrees", and partition the point cloud data into chunks, such that queries with respect to the location of the points can be answered faster. Though different from an implementation point of view, both decomposition techniques can construct and give hints of a volumetric representation for a cloud by enclosing all its points in boxes also called "voxels" with different widths. An example of such representations is given in Figures 2.1 and 2.2 for octree data structures and box-decomposition(bd) trees respectively.

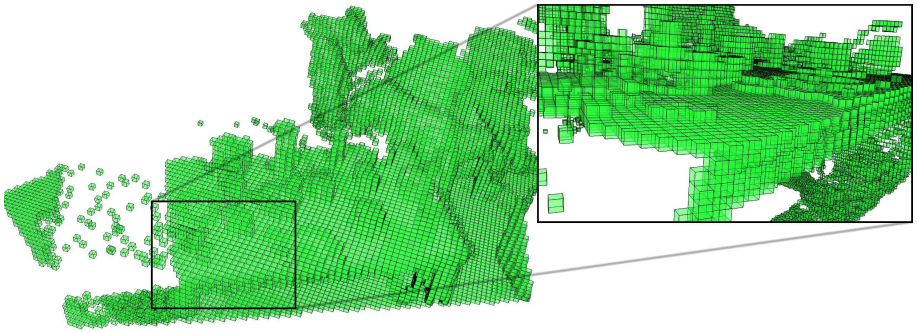


Figure 2.1: Octree volumetric representation (leaf size of 1.5cm) taken from [1]

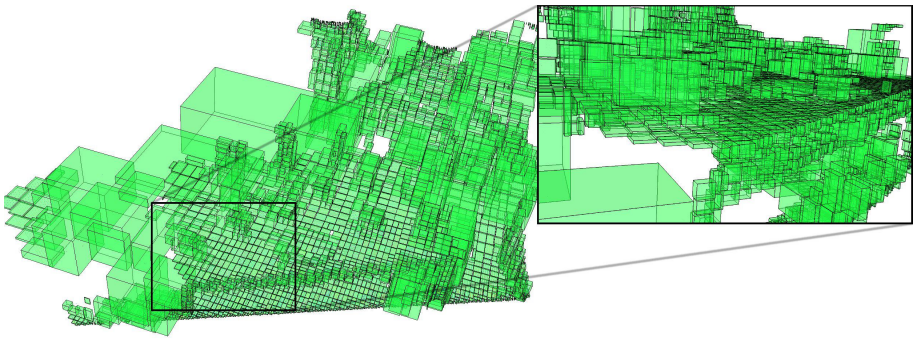


Figure 2.2: Bd-tree volumetric representation (bucket size of 30 points) taken from [1]

Octree is a tree-based data structure for managing sparse 3-D data. Each internal node has exactly eight children. The PCL octree implementation is a powerful tools for spatial partitioning and search operation. Several octree types are provided by the PCL octree component which basically differ by their individual leaf node characteristics. On the other hand, kd-Tree is a wrapper class which inherits the pcl::KdTree class. It is a generic type of 3D spatial locator using kd-tree structures for performing search functions. The class is making use of the FLANN (Fast Library for Approximate Nearest Neighbour) project. The other spatial decomposition technique mentioned above, the bd-tree, represents one variant of a kd-Tree structure optimized to provide a greater robustness for highly cluttered point cloud datasets. Using a bucket instead of a constant distance will be more reliable if the resolution of the cloud (the minimum distance between two points) is not exactly known, since it may happen that no point lies in that certain distance.
Each level of a kd-tree splits all children on a specific dimension. At the root of the tree all children will be split based on the first dimension (if the first dimension coordinate is less than the root it will be in the left sub-tree and if it is greater than the root it will obviously be in the right sub-tree). Each level down in the tree divides on the next dimension, returning to the first dimension once all other have been exhausted. The most efficient way to build a kd-tree is to use a partition method like Quick Sort to place the median point at the root and everything with

a smaller one dimensional value to the left and larger to the right.
It is however important to note, in contrast to octree structures, bd-tree or kd-tree are more difficult to update, and thus their usage is mostly limited to static scenes for applications working with individual point cloud datasets. It is recommended by PCL to use octree structures for real time applications. Thus this would be our choice of search method when implementing our project.

2.4 Data Pre-Processing

Once a point dataset has been acquired, it needs to go through a series of processing steps in order to extract meaningful information that can help a robot in performing its tasks. This is the role of a specific algorithm therefore to process and convert the raw input data into different representations and formats based on the requirements imposed by each individual processing step. Before the data set is fed to the actual recognition algorithm it will undergo different processes, called filters, such as down sampling, extracting key points, and cropping via a pass through filter.
Applications needing a real time or near real time working programs need to use different strategies to decrease execution time. In the context of point cloud processing these filters will be critically beneficial. Most of the functions performed on a point cloud to extract geometrical features, must go through searching all points one by one, so obviously more points will cause longer execution time. Additionally more similar points will cause more redundant correspondences when searching for similar points and might cause wrong registration results or need of additional processes. We will discuss this issue in more details later when we discuss the effect of different parameters in the results section. In short, these filtering stages are of great importance and need to be performed accurately.

2.4.1 Key-Points

Among different filters, the ones extracting key points are of great use, since the computational cost of descriptors is generally high, so it does not make sense to extract descriptors in all points of a cloud. Thus, key point detectors are used to select interesting points in the cloud on which descriptors are then found. Key points are points in an image or point cloud that are stable, distinctive, and can be identified using a well-defined detection criterion. Typically, the number of interest points in a point cloud will be much smaller than the total number of points in the cloud, and when used in combination with local feature descriptors at each key point, the key points and descriptors can be used to form a compact, yet descriptive representation

of the original data. They are also known as points of interest and should be able to describe electively the entire cloud using much less data.
There are different types of key points and various methods to extract them, and each technique has its own specific output. A useful comparative description and evaluation of the key point detectors most often cited in the literature and available in PCL is given in [11] to verify the invariance of the 3D key point detectors according to different rotations, scales changes and translations. Among all key point extraction methods available in PCL, uniform key points are the only one suitable for our application.

2.4.2 Down-Sampling

Down sampling filter may be simply defined as reducing the number of points in a point cloud data set. Down sampling is the most important filter for increasing the performance of any algorithm applied to point clouds. Decreasing the number of points has a significant role in reducing process time and speeding up the whole computation. This is logical since every algorithm has to go through each and every point during execution and meeting fewer points means finishing in less time.
There are two ways of implementing this filter in PCL. As previously described in search methods, an input point cloud will be represented by voxels to the filter, each voxel being a small box with a specific size in the space. Down sampling function will replace all points inside each grid with the centroid of that grid (which is different from the center of the grid), or the nearest existing point to the centroid. In the first case the function is called "voxel-grid" and in the second case it is called "uniform sampling". Figure 2.3 shows how voxels incorporate points for down sampling. The difference between these two methods is that in voxel grid, a new point is produced, but in the second case no new point is inserted to the cloud. In other words, the uniform key points belong to the original cloud, whereas the voxel key points do not belong to the original cloud. The uniform approach is a bit slower than voxel-grid, but it represents the underlying surface more accurately. In addition the uniform sampling has been chosen because with voxels, more noise will affect the results due to the fact that it introduces new points[1].
To see how this algorithm is implemented in PCL we have pasted the exact piece of our final code, which performs the down sampling. It starts with creating an object (here called `uniform_sampling` followed by a point cloud to hold the indices for sampled output points. The `setInputCloud()` function inputs the original cloud, `setRadiusSearch()` gives the size for voxels, and finally the `compute()` function performs the actual sampling with the result points held in `sampled_indices`, then copied to a pointer to be used further with other algorithms.

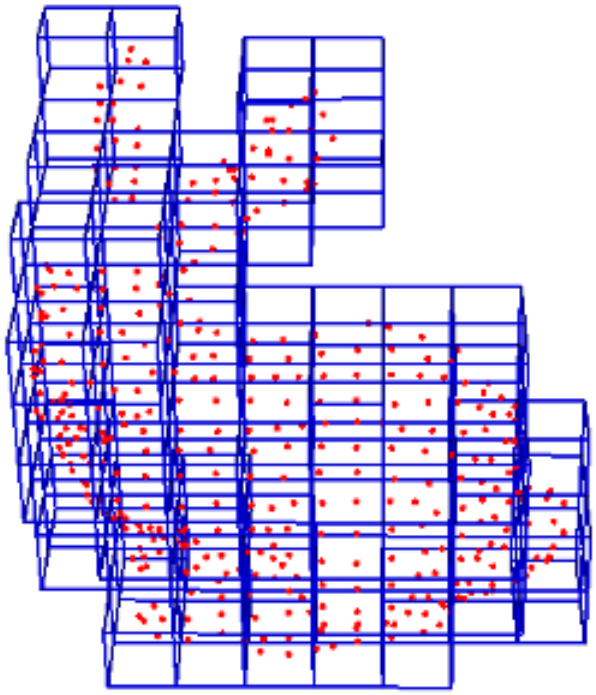


Figure 2.3: A voxelized point cloud for down-sampling, taken from[18]

Algorithm 1 Pseudo Code For Down-Sampling A Point Cloud

1. Create 3D voxel grids over the input point cloud data
2. For each grid compute the centroid of the present points
3. Set the result of 2 as the voxel for that cube
4. (if uniform sampling) The nearest point of the original cloud to the computed voxel is set as the key point for that cube

```
pcl::UniformSampling<pcl::PointXYZ> uniform_sampling;
pcl::PointCloud<int> sampled_indices;
uniform_sampling.setInputCloud (scene);
uniform_sampling.setRadiusSearch (scene_ss);
uniform_sampling.compute (sampled_indices);
pcl::copyPointCloud (*scene, sampled_indices.points, *scene_keypoints);
std::cout<<"Scene total points:"<<scene->size()<<
          ";Selected Keypoints:"<<scene_keypoints->size()<<std::endl;
```

2.4.3 Pass-Through Filter

The pass through filter implemented in PCL is a simple filtering along a specified dimension, that is to cut off values that are either inside or outside a given user range. PassThrough passes points in a cloud based on constraints for one particular field

of the point type. It iterates through the entire input once, automatically filtering non-finite points and the points outside the specified interval which applies only to the specified field. This is a rather simple class and needs no more description. Here is a pseudo code for this filter.

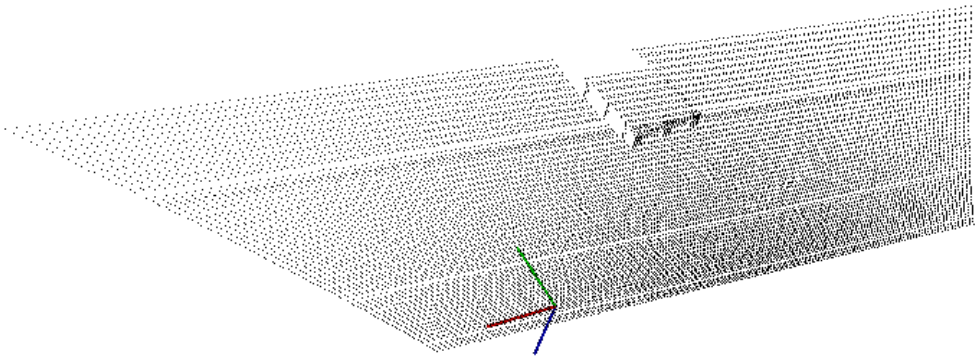
Algorithm 2 Pseudo Code For Pass-Trough Filter

1. For each point i in the model cloud, find the nearest point j in the scene cloud
2. Calculate the squared distance between these two points
3. Calculate the mean value of all squared distances

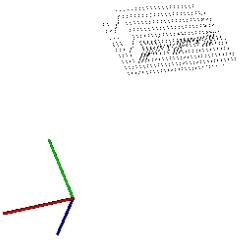
To see how this filter is implemented in PCL, here is the piece of our final code which performs filtering both on "x" and "y" axes. In our specific case in the simulation we know that the pallet is always located at $(0, 0, -3)$, since we have placed it there manually. With this in mind and considering the size of the pallet plus some margin, we cut the x values outside the range $(-1.5m, 1.0m)$ and y values outside the range $(-0.6m, 1.5m)$. In the real data as we will be seeing in the results chapter, we change this value for x to $(0.0m, 5.0m)$ and for y to $(-2.0m, 2.0m)$ as the upper level controller will place the machine in such a relative position to the pallet where the recognition will then start.
The process starts by creating an object of this class and a pointer to a cloud to hold the output. The function `setInputCloud()` inputs the original cloud, `setFilterFieldName()` specifies the axis on which we want the filter to be applied, and `setFilterLimits()` specifies the boundary limits for values on that axis. If setting the `setFilterLimitsNegative()` as true, the filter will behave the opposite, meaning that it will discard values in between, and holds values outside the specified limit. Finally calling the `filter()` function will perform the actual filtering, leaving the result points in the defined point cloud. We have performed this routine twice, once on the x axis and once on the y axis to gain our desired cloud. Figure 2.4 shows a sample scene before and after filtering with pass through.

```
pcl::PassThrough<pcl::PointXYZ> pass;
pcl::PointCloud<pcl::PointXYZ>::Ptr scene_filtered1
            (new pcl::PointCloud<pcl::PointXYZ> ());
pass.setInputCloud (scene_unfiltered);
pass.setFilterFieldName ("x");
pass.setFilterLimits (-1.5f,1.0f);
pass.filter (*scene_filtered1);
pcl::PointCloud<pcl::PointXYZ>::Ptr scene_filtered2
            (new pcl::PointCloud<pcl::PointXYZ> ());
pass.setInputCloud (scene_filtered1);
pass.setFilterFieldName ("y");
```

```
pass.setFilterLimits (-0.6f , 1.5f);
pass.filter (*scene_filtered2);
pcl::PointCloud<pcl::PointXYZ>::Ptr scene
            (new pcl::PointCloud<pcl::PointXYZ> ());
*scene = *scene_filtered2 ;
```



(a) Before



(b) After

Figure 2.4: Applying pass-through (crop) filter to a sample scene

2.5 Features

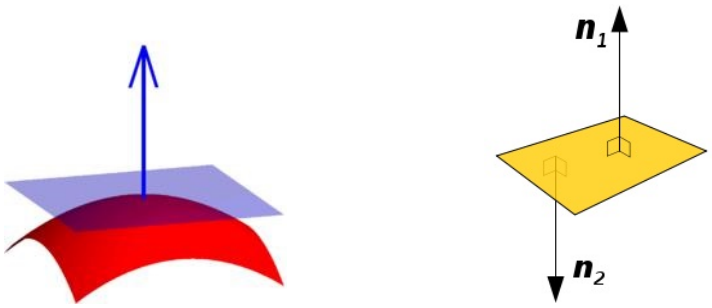
In their native representation, points are simply represented using their Cartesian coordinates with respect to a given origin. Assuming that the origin of the coordinate system does not change over time, there could be two points acquired at two different times having the same coordinates. Comparing these points however is a problem, because even though they are equal with respect to some distance measure, they could be sampled on completely different surfaces, and thus represent totally different information when taken together with the other surrounding points in their vicinity. That is because there are no guarantees that the world has not changed between two time instances. Some acquisition devices might provide extra information for a sampled point, such as an intensity or surface remission value, or even a color, however that does not solve the problem completely and the comparison remains ambiguous.

Applications which need to compare points for various reasons require better characteristics and metrics to be able to distinguish between geometric surfaces. The concept of a 3D point as a singular entity with Cartesian coordinates therefore disappears, and a new concept, that of local descriptor takes its place. The literature is abundant of different naming schemes describing the same conceptualization, such as shape descriptors or geometric features. For the remaining of this document they will be referred to as both features and descriptors.
In vision and perception the word "feature" can have many different meanings. In PCL, feature estimation is defined as computing a feature vector based on each points local neighbourhood, or sometimes computing a single feature vector for the whole point cloud. Feature vectors can be anything from simple surface normals to the complex feature descriptors needed for registration or object detection. By including the surrounding neighbours, the underlying sampled surface geometry can be inferred and captured in the feature formulation, which contributes to solving the ambiguity of comparison. Ideally, the resultant features would be very similar (with respect to some metric) for points residing on the same or similar surfaces, and different for points found on different surfaces. A good point feature representation distinguishes itself from a bad one, by being able to capture the same local surface characteristics in the presence of rigid transformations, that is, 3D rotations and 3D translations in the data should not influence the resultant feature vector estimation; varying sampling density, in principle, a local surface patch sampled more or less densely should have the same feature vector signature; and noise, the point feature representation must retain the same or very similar values in its feature vector in the presence of mild noise in the data.
An example for a local 3D descriptor "Signature of Histograms of OrienTations" (SHOT) descriptor introduced in [21]. Another one is the "Fast Point Feature Histogram" (FPFH) descriptor, shown in [14]. This descriptor generates a histogram of the angular variations of the normals found in the neighbourhood of the point. Global descriptors use a single vector to describe the whole point cloud. In contrast, local descriptors describe the local region around each point, hence many local descriptor vectors are needed to describe the whole point cloud. In [7] the "Global Fast Point Feature Histogram" (GFPFH) descriptor is introduced, which generates a global object description on the basis of the local FPFH descriptors. "Viewpoint Feature Histogram" is another global feature introduced in [20] which is estimated on a point cloud using points, normals and FPFH features.
When using global descriptors the whole model can be described in one vector. Having just one feature for the model could be useful, but it is extremely difficult to segment a single object in the scene cloud, so the global features will contain too much noise and it is not exploitable in this case. Due to this fact we will not be

using or describing global descriptors here. But the ones we are interested in are the fastest and more reliable ones. In most applications nowadays the FPFH and SHOT are widely used, since we will focus and compare these two types which will be described later in this section.

2.5.1 Surface Normals

In the 3D space a surface normal or simply a normal to a surface at a point P is a vector that is perpendicular to the tangent plane to that surface at p as shown in Figure 2.5(a). The normal is often used in computer graphics to determine an orientation of a surface towards a source. Figure 2.5(b) shows a plane and its two possible normals.

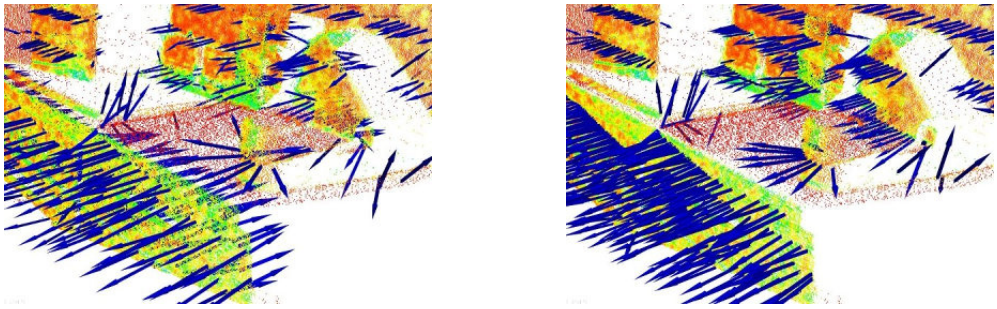


(a) A surface normal

(b) Two possible directions of normals

Figure 2.5: Surface normal definition
pictures taken from http://en.wikipedia.org

Though many different normal estimation methods exist, the one that we will concentrate on at this point, is one of the simplest. The problem of determining the normal to a point on the surface is approximated by the problem of estimating the normal of a plane tangent to the surface as shown in Figure 2.5(a), which in turn becomes a least-square plane fitting estimation problem. The solution for estimating the surface normal is therefore reduced to an analysis of the eigenvectors and eigenvalues of a covariance matrix created from the nearest neighbours of the query point.
In general, because there is no mathematical way to solve for the sign of the normal, its orientation computed via Principal Component Analysis (PCA) as shown in Figure 2.5(b) is ambiguous and not consistently oriented over an entire point cloud dataset. Figure 2.6(a) presents this effect on a section of a larger dataset. Due to the orientation inconsistency, all the normals of a single plane are not correctly oriented, but spread across the entire plane.



(a) None oriented surface normals in a point cloud

(b) Oriented surface normals in a point cloud

Figure 2.6: Oriented versus non-oriented surface normals, taken from [1]

The solution to this problem would be obvious if the viewpoint v_p is in fact known. To orient all normals $\overrightarrow{n_i}$ consistently towards the viewpoint, they need to satisfy the equation:

$$\overrightarrow{n_i}.(v_p - p_i) > 0$$

The results after all normals in the datasets have been consistently oriented towards the viewpoint are represented in Figure 2.6(b). Given a geometric surface, it is usually trivial to infer the direction of the normal at a certain point on the surface as the vector perpendicular to the surface in that point. However, since the point cloud datasets that we acquire represent a set of point samples on the real surface, there are two possibilities:

1 . Obtain the underlying surface from the acquired point cloud dataset, using surface meshing techniques, and then compute the surface normals from the mesh;

2 . Use approximations to infer the surface normals from the point cloud dataset directly.

The implementation in PCL will address the latter that is, given a point cloud dataset, directly compute the surface normals at each point in the cloud. The algorithm for normals estimation may be briefly described in the following pseudo code.

Algorithm 3 Pseudo Code For Point Cloud Normal Estimation

1. For each point p in the model cloud, get the nearest neighbours within the specified radius
2. Compute the surface normal n of p
3. Check if n is consistently oriented towards the viewpoint and flip otherwise

As explained, a surface normal at a point needs to be estimated from the surrounding point neighbourhood support of the point (also called k-neighbourhood).

The specifics of the nearest-neighbour estimation problem raises the question of the right scale factor: given a sampled point cloud dataset, what are the correct k values that should be used in determining the set of nearest neighbours of a point. This issue is of extreme importance and constitutes a limiting factor in the automatic estimation (without user given thresholds) of a point feature representation. To better illustrate this issue, Figure 2.7 presents the effect of selecting a smaller scale (small r or k) versus a larger scale (large r or k).

The left part of Figure 2.7 depicts a reasonable well-chosen scale factor, with es-

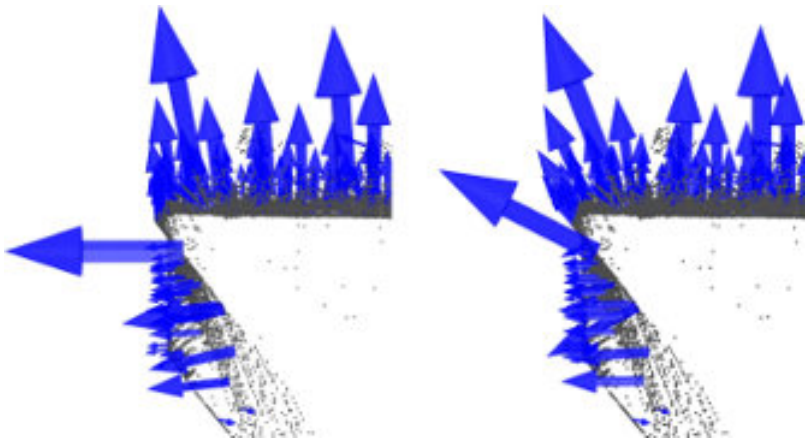


Figure 2.7: Scale factor effect on surface normals, taken from [1]

timated surface normals approximately perpendicular for the two planar surfaces. If the scale factor however is too big (right part), and thus the set of neighbours is larger covering points from adjacent surfaces, the estimated point feature representations get distorted, with rotated surface normals at the edges of the two planar surfaces, and smeared edges and suppressed fine details.

Without going into too many details, it suffices to assume that for now; the scale for the determination of a point's neighbourhood has to be selected based on the level of detail required by the application. Simply if the curvature at the edge between the handle of a mug and the cylindrical part is important, the scale factor needs to be small enough to capture those details, and large otherwise. Surface normals are widely used for different purposes such as surface reconstruction, segmentation, different quality enrichments, smoothing and etcetera.

```
pcl::NormalEstimation <pcl::PointXYZ, pcl::Normal> norm_est;
pcl::search::KdTree <pcl::PointXYZ>::Ptr tree
    (new pcl::search::KdTree<pcl::PointXYZ>);
norm_est.setSearchMethod (tree);
norm_est.setRadiusSearch (0.1f);
//    Model Normals
```

```
norm_est.setInputCloud (model_keypoints);
norm_est.compute (*model_normals);
//    Scene Normals
norm_est.setInputCloud (scene_keypoints);
norm_est.compute (*scene_normals);
```

2.5.2 PFH - Point Feature Histogram

PFH descriptors are a local features based on surface normals, 3D data and curvature, which are capable of storing information about the geometry around a specific point. The goal of the PFH formulation is to encode a point's k-neighbourhood geometrical properties by generalizing the mean curvature around the point using a multi-dimensional histogram of values. Figure 2.8 presents an influence region diagram of the PFH computation for a query point p_q, marked with red and placed in the middle of a circle (sphere in 3D) with radius r, and all its k-neighbours (points with distances smaller than the radius r) are fully interconnected in a mesh. The final PFH descriptor is computed as a histogram of relationships between all pairs of points in the neighbourhood, and thus has a computational complexity of $O(k^2)$.

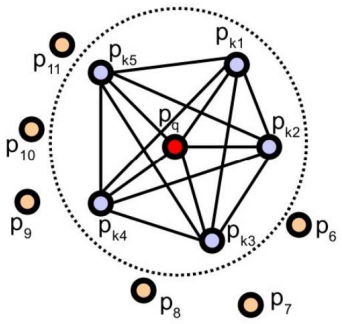


Figure 2.8: Point feature histogram descriptor influence region, taken from [1]

The `pcl::PFHEstimation` class is the PCL implementation that computes this kind of feature. This class accepts 3D coordinates (x, y, z) of n points, normals of the input points (computed with a normal radius r_n, and feature estimation radius r_f (which has to be greater than r_n) or number of neighbours to consider (k neighbours) as inputs and the result is an array of 125 float values (125 bytes) that represent the histogram of the so called "bins" which encodes the neighbourhood's geometrical properties and provide an overall point density and pose invariant multi-value feature. This feature is stored in the `pcl::PFHSignature125` type.

2.5.3 FPFH - Fast Point Feature Histogram

For real-time or near real-time applications, the computation of PFH features in dense point neighbourhoods can represent one of the major bottlenecks. FPFH features class computes another type of feature reducing the computation time and complexity in comparison to PFH, yet maintaining most of its information. To simplify the PFH computation, FPFH proceeds as follows:
1. First, for each query point p_q a set of tuples α, ϕ, θ between itself and its neighbours are computed as in PFH descriptors. This will be called the Simplified Point Feature Histogram (SPFH).
2. Then for each point, its k neighbours are re-determined, and the neighbouring SPFH values are used to weight the final histogram of p_q (called FPFH) as with the following formula:

$$FPFH(p_q) = SPFH(p_q) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{\omega_k}.SPFH(p_k)$$

where the weight ω_k represents a distance between the query point p_q and a neighbour point p_k in some given metric space, thus scoring the (p_q, p_k) pair, but could just as well be selected as a different measure if necessary. To understand the importance of this weighting scheme, Figure 2.9 presents the influence region diagram for a k-neighbourhood set centred at p_q.

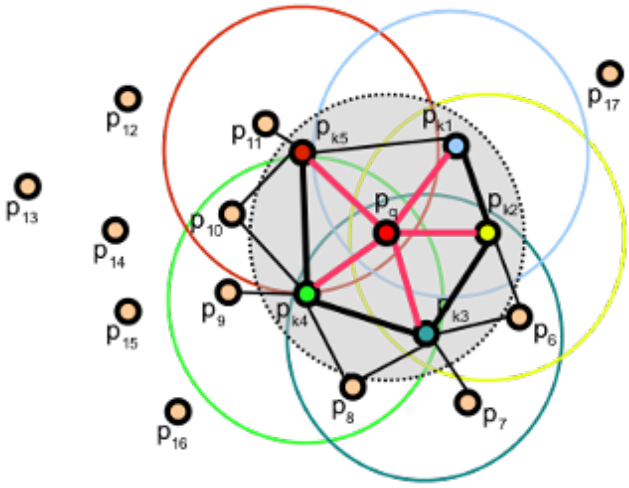


Figure 2.9: Fast point feature histogram descriptor influence region, taken from [1]

The key differences between PFH and FPFH computation procedure are:
✻ The PFH of the point is computed with all the mesh of its neighbours, the FPFH takes into account only the direct connections between itself and its neighbours.

* There are two steps for computation: in the first step the SPFH of all points are computed, in the second step for each point the values of neighbour's SPFH are used to weight the final FPFH histogram.
At high level we can say:
* The PFH of the point contains all and only the information provided by its k neighbours (or distance r).
* The FPFH of a point does not contain all the relationships between its neighbours, but contains some relationship between its neighbour's (at distance $2r$).
The input of this class is the same as the input of PFH features, and the output is an array of 33 float values (33 bytes) that represent the FPFH histogram, that is stored in the `pcl::FPHSignature33` type. Note that the effectiveness of this feature depends on the chosen radius, hence different radii expressed in millimetres should be tested to get the best result.
Now as for our explanation routine, the piece of code is pasted here, since this is the feature type we used in our algorithm. Like previous PCL algorithms, feature extraction also starts with creating an object, here called `fpfh_est`. With the function `setSearchMethod()`, tree is given to be the method for searching. The input cloud is given by calling the `setInputCloud()` function, which is here the key points, since we want the features to be computed only on extracted key points. Also previously calculated normals are provided by `setInputNormals()`. `setRadiusSearch()` specifies the radius to consider for computing the features, and finally the `compute()` function performs the actual calculations with the result values held in a previously defined point cloud.

```
pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33>fpfh_est;
fpfh_est.setSearchMethod (tree);
fpfh_est.setInputCloud (scene_keypoints);
fpfh_est.setInputNormals (scene_normals);
fpfh_est.setRadiusSearch (scene_descr_rad);
fpfh_est.compute (*scene_descriptors);
```

2.5.4 SHOT - Signature of Histograms of OrienTations

The SHOT descriptor presented in [21] is based on obtaining a repeatable local reference frame using the eigenvalue decomposition around an input point. Given this reference frame, a spherical grid centred on the point divides the neighbourhood so that in each grid bin a weighted histogram of normals is obtained. The descriptor concatenates all such histograms into the final signature. The spherical grid is shown in Figure 2.10. It uses 9 values to encode the reference frame and the authors propose the use of 11 shape bins and 32 divisions of the spherical grid, which gives an additional 352 values. The descriptor is normalized to sum 1.

Descriptors except SHOT are either "histograms" or "signatures" based. The former type maintains a histogram or properties of neighbouring points, while the latter type calculates some single value of the neighbouring points[21]. The histogram index i of a neighbouring point q inside a sector is calculated by $\frac{1}{2}(1+r.n_q)b$ where b is the number of bins in the histogram. To reduce quantization errors, a neighbouring point also contributes to histograms in neighbouring sectors of the subdivision using a quadrilinear interpolation. Figure 2.10 shows subdivisions used by the SHOT descriptor.

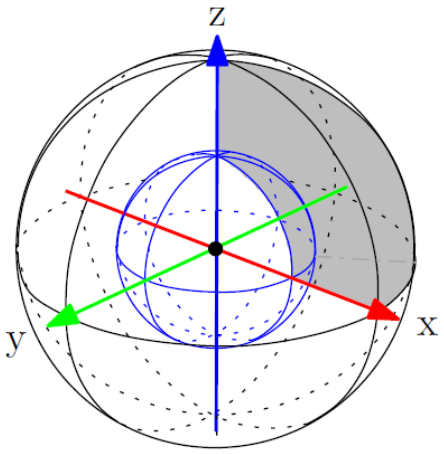


Figure 2.10: Subdivision used by signature of histograms of orientations descriptor, the inner sector or shell is depicted in blue and one sector of the descriptor is highlighted in light gray, taken from [19]

2.6 Correspondences

A correspondence is simply a couple of any class that are similar to some extent. In point cloud processing context, correspondences represent a match between two entities (points, descriptors) represented via the indices of a source point and a target point, and the distance between them. In other words each correspondence is a pair of points which the correspondence finding algorithm suggests them to represent the same point in both clouds.

2.6.1 Finding Correspondences

It is both possible to find all possible correspondences first and then reject them by provided classes, or to search features one by one and add them to correspondences manually if they fulfil some conditions. To search manually, one may find correspondences iteratively on parts of the scene cloud enclosed in spheres, or find

correspondences iteratively on every cluster of the scene cloud, or go through all scene key points and find the nearest feature in the model key points.
`pcl::registration::CorrespondenceEstimation` class is inherited by all following classes as of PCL 1.7:
* `pcl::registration::CorrespondenceEstimationBase` represents the base class for determining correspondences between target and query point sets/features.
* `pcl::registration::CorrespondenceEstimationBackProjection` computes correspondences as points in the target cloud which have minimum distance to the projected point in the model.
* `pcl::registration::CorrespondenceEstimationNormalShooting` computes correspondences as points in the target cloud which have minimum distance to normals computed on the input cloud

It would be up to the user to utilize an implemented tool in order to avoid the problem of introducing a lot of false information and wrong data that definitely would compromise the overall registration. Figure 2.11 shows all correspondences found with `CorrespondenceEstimation()` in a sample saved scene cloud. The correspondences shown in green lines connect two points which the estimation function has paired. Also the piece of code performing this task is represented which is a simple procedure of accepting descriptors for both model and scene and outputting the pairs.

```
pcl::registration::CorrespondenceEstimation
                    <pcl::FPFHSignature33, pcl::FPFHSignature33> cor_est;
cor_est.setInputSource (model_descriptors);
cor_est.setInputTarget (scene_descriptors);
pcl::CorrespondencesPtr model_scene_corrs (new pcl::Correspondences ());
cor_est.determineCorrespondences (*model_scene_corrs);
```

2.6.2 Filtering Correspondences

Scene features usually contain a lot of data, which makes it difficult to perform a robust search and match, it is better to use a divide and conquer strategy. If all the scene features are to be used, the results would be very bad, since the model features can be coupled with any scene feature, at any distance, for example two features that are really close to each other in the model, can be coupled respectively with two features that are very far from each other in the world, that is obviously impossible and gives a wrong corresponding set. Figure 2.11 clearly shows this phenomena. Due to this fact, founded correspondences need to be filtered by some carefully specified conditions to avoid participation of non relevant ones in the main clustering phase.
`pcl::registration::CorrespondenceRejector` class is inherited by all following

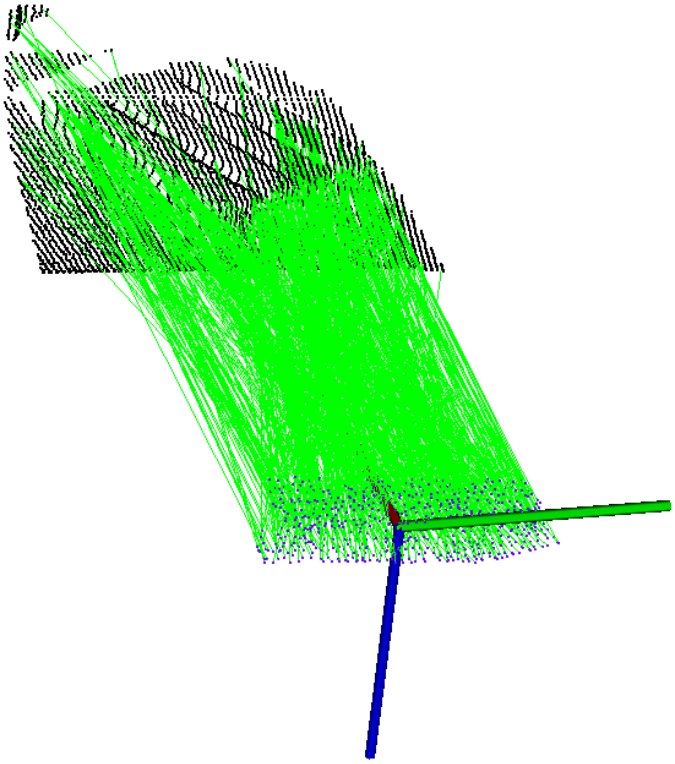


Figure 2.11: All correspondences found between the model and the scene with correspondence estimation shown in green lines

classes as of PCL 1.7:

* `pcl::registration::CorrespondenceRejectorDistance` rejects all the correspondences that exceed a distance threshold. The correspondence distance is the measure between corresponding key points of model and scene clouds.
* `pcl::registration::CorrespondenceRejectorMedianDistance` computes the median distance between all the correspondences, then all the correspondences that exceed a deviation threshold from that mean value are rejected.
* `pcl::registration::CorrespondenceRejectorFeatures` implements a correspondence rejection method based on a set of feature descriptors. Given an input feature space, the method checks if each feature in the model cloud has a correspondence in the scene cloud, either by checking the first k (given) point correspondences, or by defining a tolerance threshold via a radius in feature space.
* `pcl::registration::CorrespondenceRejectorOneToOne` implements a correspondence rejection method based on eliminating duplicate match indices in the correspondences. Correspondences with the same match index are removed and only the one with smallest distance between query and match are kept. That is, considering match at a query, one to many correspondences are removed leaving only one to one correspondences.

* `pcl::registration::CorrespondenceRejectionOrganizedBoundary` class implements a simple correspondence rejection measure. For each pair of points in correspondence, it checks whether they are on the boundary of a silhouette. This is done by counting the number of NaN dexels in a window around the points (the threshold and window size can be set by the user).
* `pcl::registration::CorrespondenceRejectorPoly` implements a correspondence rejection method that exploits low-level and pose-invariant geometric constraints between two point sets by forming virtual polygons of a user-specifiable cardinality on each model using the input correspondences. These polygons are then checked in a pose-invariant manner (i.e. the side lengths must be approximately equal), and rejection is performed by thresholding these edge lengths.
* `pcl::registration::CorrespondenceRejectorSampleConsensus` implements a correspondence rejection using Random Sample Consensus to identify inliers and reject outliers. RANSAC is a non-deterministic iterative method to estimate parameters of a mathematical model, from a set of observed data which contains outliers.
* `pcl::registration::CorrespondenceRejectorSampleConsensus2D` implements a pixel-based correspondence rejection using Random Sample Consensus to identify inliers and reject outliers.
* `pcl::registration::CorrespondenceRejectorSurfaceNormal` implements a simple correspondence rejection method based on the angle between the normals at correspondent points.
* `pcl::registration::CorrespondenceRejectorTrimmed` implements a correspondence rejection for ICP-like registration algorithms that uses only the best k correspondences where k is some estimate of the overlap between the two point clouds being registered.
* `pcl::registration::CorrespondenceRejectorVarTrimmed` implements a simple correspondence rejection method by considering as inliers a certain percentage of correspondences with the least distances. The percentage of inliers is computed internally.

Figure 2.12 shows the remaining correspondences after rejecting the bad ones out of all correspondences in Figure 2.11 using `CorrespondenceRejectorSampleConsensus` class. The correspondences shown in green lines in this figure are the ones which the transformation matrix of the registration algorithm will be computed based on. This will be the class used to reject correspondences in our proposed algorithm, thus the code piece is pasted. The procedure starts with creating an object of that class here called `sac`. It then accepts keypoints for model and scene in addition to their previously found correspondences.

This algorithm needs a threshold with which it will consider points to be inliers or outliers. Points will be counted as inliers if they have less distance than the

threshold to the proposed model and vice versa.

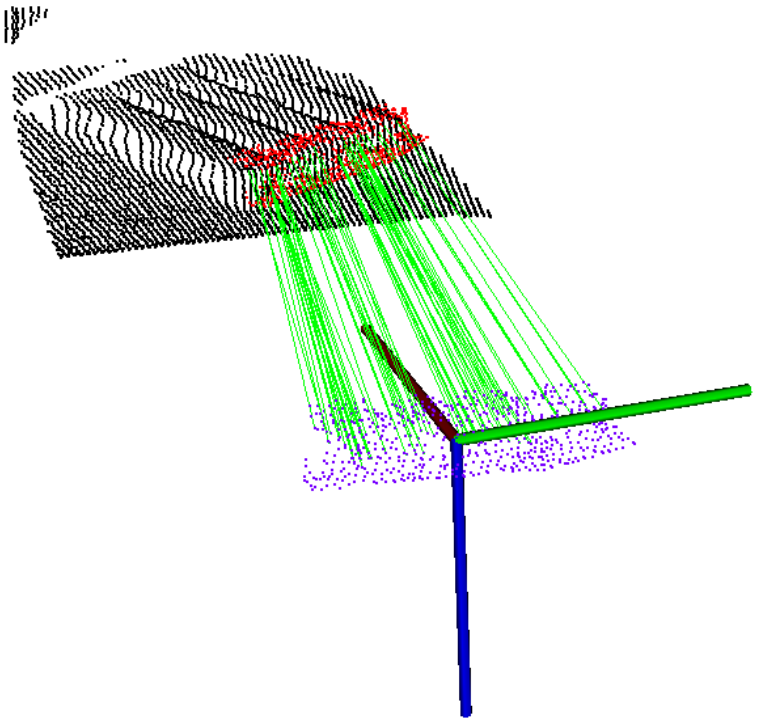


Figure 2.12: Remaining correspondences between the model and the scene after rejection shown in green lines

```
double epsilon_sac = 0.03;
int iter_sac = 200;
pcl::registration::CorrespondenceRejectorSampleConsensus<pcl::PointXYZ> sac;
sac.setInputSource (model_keypoints);
sac.setInputTarget (scene_keypoints);
sac.setInlierThreshold (epsilon_sac);
sac.setMaximumIterations (nr_iterations);
sac.setInputCorrespondences (model_scene_corrs);
pcl::Correspondences inliers;
sac.getCorrespondences (inliers);
Eigen::Matrix4f T = sac.getBestTransformation();
```

2.7 Validation Functions

In order to verify the correctness of any performable registration, we need a robust validation function, that is capable of telling a good alignment to a bad one. For different needs three different validation functions have been developed:

1. Euclidean distance validation function: it is used in the final registration to find out the best match using the square distances between points.
2. Percent of outliers validation function: it is used in the initial registration to

recognize occlusions using the number of outliers points.
3. Normal angles validation function: it has been tried together with the Euclidean distance validation function, but results are much worse, spending much more computation time.

2.7.1 Euclidean Distance Validation Function

Euclidean Distance is a new specific-purpose fitness score that uses squared distances and it is used while performing the initial rough registration. This function accepts the scene cloud (consisting of n points), and the model cloud (consisting of m points) as inputs and outputs a single float score that represents the grade of alignment of the object into the scene cloud according to the formula

$$score = \frac{1}{m}\sum_{i=0}^{m-1} min_{j=0}^{n-1}(\|model.i - scene.j\|^2)$$

where $model.i$ and $scene.j$ represent the 3D vectors containing the coordinates of the points in the model and scene point clouds respectively. From the result score, zero represents a perfect alignment while $3,40282 * 10^{38}$ means the worst alignment. This formula appears to be a very good method to roughly validate the achieved results, since it is based on squared distances that penalizes even the smallest distances between model and the scene.

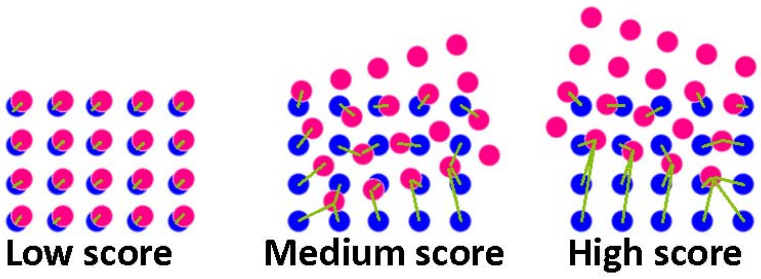


Figure 2.13: Calculating the euclidean score, blue: model cloud ; red: scene cloud taken from [4]

The drawbacks of this approach are:
✻ The possible presence of occlusions: in this case the score grows very fast, compromising the entire effectiveness of the registration process. The key to avoid this is based on choosing the right model cloud.
✻ A wrong alignment may still maintain small distances between object and world. In this case the score remains low and we cannot tell if the alignment is good enough or not.

Algorithm 4 Pseudo Code For The Euclidean Distance Validation Function

1. For each point i in the model cloud, find the nearest point j in the scene cloud
2. Calculate the squared distance between these two points
3. Calculate the mean value of all squared distances

2.7.2 Percent Of Outliers Validation Function

The percent of outliers validation function is a simple yet more effective fitness score to recognize occlusions and bad matching which is used while performing the final fine registration. This function accepts the scene cloud (consisting of n points), and the model cloud (consisting of m points) as inputs and outputs a single float score that represents the percent of outliers based on a small distance threshold according to the formula:

$$score = \frac{100}{m} \sum_{i=0}^{m-1} \begin{cases} 1 & if min_{j=0}^{n-1} \|model.i - scene.j\| > DistanceThreshold \\ 0 & otherwise \end{cases}$$

where $model.i$ and $scene.j$ represent the 3D vectors containing the coordinates of the points in the model and scene point clouds respectively. In other words, no matter if the distance is high, each outliers counts as one. From the result score, zero represents a perfect alignment (with 0% of outliers) and 100 represents the worst alignment with 100% of outliers).

This function is developed since the previous one is not sensible to particular

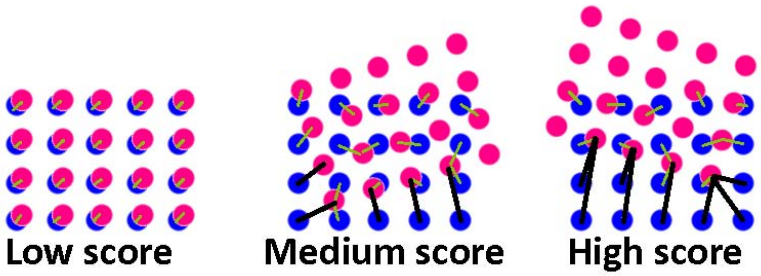


Figure 2.14: Calculating percent of outliers score, blue: model cloud ; red: scene cloud, taken from [4]

cases, where euclidean distances are small but the alignment is wrong. Whereas this function is able to tell a wrong alignment from a right one, while the euclidean distance function cannot discriminate a false positive.

Algorithm 5 Pseudo Code For The Percent Of Outliers Validation Function

1. For each point i in the model cloud, find the nearest point j in the scene cloud
2. Calculate the distance between these two points,

if it is greater than the distance threshold **then**
  it is an outlier
else
  it is not an outlier
end if

3. Calculate the percent of outliers by dividing the sum by m and multiplying it by 100

2.7.3 Normal Angles Validation Function

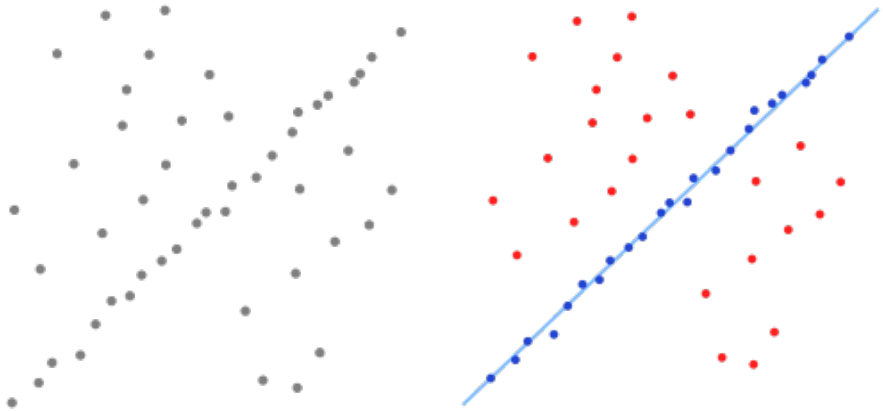
This function has been developed to take the place of the percent of outliers validation function in order to score the final registration, but the low performance of this procedure made it a bad choice if a good performance is required. This function is in many ways similar to the euclidean distance validation function, but it is based on the squared differences of angles between the normals of the model cloud and the normals of the scene cloud. More precisely, the normals taken into account are calculated on each point in the model cloud and its closest neighbours.
This function also accepts the scene cloud (consisting of n points), and the model cloud (consisting of m points) as inputs and outputs a single float score that represents the grade of alignment of the model into the scene, with zero meaning a perfect alignment (0 degrees of difference between normals for all points) and 180^2 meaning the worst alignment (meaning that all the normals are opposite). It has been proven that in comparison to the other two functions, the normal angles function is not a good choice, both for quality of the results and time consumption.

Algorithm 6 Pseudo Code For The Normal Angles Validation Function

1. Compute normals for each point of the model and scene clouds with a fixed given radius
2. For each point i in the object:
(a) find the nearest point j in the scene cloud
(b) calculate the squared distance of the angle between normal of the point i in the model cloud and normal of the point j in the scene cloud
3. Calculate the mean value of all squared distances

2.8 Random Sample Consensus (RANSAC)

RANSAC is an abbreviation for "RANdom SAmple Consensus" which is an iterative method to estimate parameters of a mathematical model from a set of observed data



(a) Data set (b) Fitted line

Figure 2.15: Random sample consensus line fitting, (a):input cloud; (b):blue dots indicate inliers and red dots indicate outliers

which contains outliers. RANSAC is a non-deterministic algorithm in the sense that it produces a reasonable result only with a certain probability, with this probability increasing as more iteration are allowed. The algorithm was first published by Fischler and Bolles at SRI International in 1981.
This is a general model fitting algorithm that uses the minimum number observations (data points) required to estimate the underlying model parameters. A basic assumption in RANSAC is that the data contains two types, "outliers" and "inliers", with inliers being the relevant data of the model and the outliers coming mostly from noise sources or undesired data.
As this context will be an important part in our project, for better understandings we go through an example in 2D. Assume a set of input data points and the goal is to fit a line with the most number of lying points. At least two points are needed to individually clarify a line in 2D space, thus the algorithm randomly chooses two points in the data set. It assumes this line as the model and classifies the remaining points as either inliers or outliers. Each point is assumed to be an inlier if the distance between that point and the line is less than a certain threshold. The fraction of inlier points to outliers must be at a predefined accepting level. After saving the data derived from these processes the algorithm takes two other random points and repeats the procedure. A visualization is given in Figure 2.15.

This procedure is repeated a fixed number of times, each time producing either a model which is rejected because too few points are classified as inliers or a refined model together with a corresponding error measure. In the latter case, we keep the refined model if its error is lower than the last saved model. The number of iterations, N is chosen high enough to ensure that at least one of the sets of random samples does not include an outlier.

Extending of RANSAC algorithm to 3D space and in our specific application of template matching involves getting a number of correspondences or points at each time and transforming the model accordingly. Then data points are categorized as inliers and outliers with respect to a user defined threshold and a score is assigned to this transformation. The whole procedure is repeated a user given number of times and the best value along with its respective transformation matrix will be returned. PCL holds different RANSAC based methods and models that can be combined freely in order to detect specific models and their parameters in point clouds. One was mentioned among correspondence rejection functions as `pcl::registration::CorrespondenceRejectorSampleConsensus2D` which uses a RANSAC based approach to categorize correspondences as inlers and outliers.
In addition there is another class implemented as the `Sampled Consensus_Initial Alignment` algorithm. SAC-IA performs fast searches in an exhaustive correspondence space to find a good alignment solution which can be further refined using a non-linear optimization method [14]. This algorithm estimates and rejects correspondences between two clouds in one single step, as opposed to the other method that performs this task in two separate steps. This class as its name suggests only provides an initial guess and for a better alignment the results are usually refined with other available algorithms.

Algorithm 7 Pseudo Code For RANSAC Based Alignment Algorithm

1. Select s sample features from the scene cloud which their pairwise distances are greater than a user defined minimum distance, MinSampleDistance.
2. For each sample feature find a list of similar features in the model cloud. Select one of them to be considered that sample features correspondence which their distance is less than a user defined maximum distance, MaxCorrespondenceDistance.
3. Compute the rigid transformation defined by the sample features and their correspondences and compute an error metric for the point cloud that computes the quality of the transformation.
4. If the number of iterations is less than a user defined number, MaximumIterations, repeat the procedure.

The piece of code performing this algorithm is represented below. The process begins with defining the three above mentioned parameters that should be given by the user. Next an object of this class is created, here called `sac_ia`. The parameters are passed to the object with relative functions `setMinSampleDistance()`, `setMaxCorrespondenceDistance()`, and `setMaximumIterations()`. The scene cloud and its features are passed by calling `setInputTarget()` and `setTargetFeatures()` functions respectively. The same is done for the model cloud with `setInputSource()` and `setSourceFeatures()` functions. After defining a point cloud to hold the re-

sults, function `align()` will perform the main task of alignment. Finally the fitness score is printed out with a call to function `getFitnessScore()` and the result is copied into a pointer for next uses.

```
float min_sample_distance = 0.2;
float max_correspondence_distance = 0.3f;
int nr_iterations = 200;
pcl::SampleConsensusInitialAlignment <pcl::PointXYZ, pcl::PointXYZ,
                                      pcl::FPFHSignature33> sac_ia;
sac_ia.setMinSampleDistance (min_sample_distance);
sac_ia.setMaxCorrespondenceDistance (max_correspondence_distance);
sac_ia.setMaximumIterations (nr_iterations);
sac_ia.setInputTarget (scene_keypoints);
sac_ia.setTargetFeatures (scene_descriptors);
sac_ia.setInputSource (model_keypoints);
sac_ia.setSourceFeatures (model_descriptors);
pcl::PointCloud<pcl::PointXYZ> registration_output;
sac_ia.align (registration_output);
cout << "sac-ia fitness score:" << sac_ia.getFitnessScore
                    (max_correspondence_distance) << endl;
if (sac_ia.getFitnessScore (max_correspondence_distance) < 0.00002)
cout << "sac-ia fitness score is good! (less than 0.00002)" << endl;
pcl::PointCloud<pcl::PointXYZ>::Ptr rotated_model
          (new pcl::PointCloud<pcl::PointXYZ>());
*rotated_model = registration_output;
```

2.9 Iterative Closest Points (ICP)

The Iterative Closest Points (ICP) method is also a general model fitting algorithm performing alignment of two models called source and target. This algorithm proposed in [22] is motivated by minimizing the difference between two clouds of points. Algorithm 8 presents a pseudo code related to this method. After ICP executes successfully, the source will be aligned to the target. The algorithm requires the source cloud to be already close to the correct match, as such ICP is used for refinement, while other methods (such as SAC-IA) can be used to perform an initial alignment. The ICP algorithm works by estimating how well the two point clouds match. This can be done in different ways, but the preferred and simpler method is to use point-to-point errors; for each point in the first cloud, the algorithm searches for the closest point in the second cloud and computes its distance. The transformation that aligns the source cloud to the target cloud is then the one that minimizes the total error between all point pairs. This alignment step is performed iteratively until a good match is found. The standard ICP algorithm has three alternative termination conditions:

1. The number of iterations has reached the maximum specified by user
2. The epsilon change value between the two last iteration is smaller than a value specified by the user
3. A fitness function computed internally has reached a threshold specified by the user

The choice of this two step alignment (initial guess plus fine alignment) lies on the fact that ICP performs well when its input clouds are relatively already close enough, while SAC-IA is less accurate but more robust to large translations and rotations. Employing both methods we obtain the robustness of SAC-IA and sharpness of ICP at the same time. In addition to that, there is one other advantage of combining the two. SAC-IA is a relevantly slow process and it will consume extremely large amount of time to give a fine alignment, but ICP is perfectly fast, so we decrease the number of iterations of SAC-IA and interrupt its process, and let ICP do the fine and sharp final alignment which will be much faster.

The code piece performing ICP is represented below. The process begins with defining an object of this class, here called `icp`. The scene cloud is passed by calling `setInputTarget()` and the model cloud by calling `setInputSource()` functions. After defining a point cloud `Final` to hold the result of alignment, function `align()` will perform the main task. Finally the fitness score is printed out with a call to function `getFitnessScore()`.

```
pcl::IterativeClosestPoint <pcl::PointXYZ, pcl::PointXYZ> icp;
icp.setInputSource (rotated_model);
icp.setInputTarget (scene_keypoints);
pcl::PointCloud<pcl::PointXYZ> Final;
icp.setEuclideanFitnessEpsilon (0.01);
icp.align(Final);
cout << "icp fitness score: " << icp.getFitnessScore() << endl;
```

Algorithm 8 Pseudo Code For ICP Based Alignment Algorithm

1. For each point in the model cloud, find the closest point in the scene cloud.
2. Find the geometric centroid of accepted matches.
3. Find out rotation and translation based on Singular Value Decomposition (SVD).
4. Calculate new average distances using the rotation and translation matrix.
5. Terminate the loop if any of the termination conditions is reached, otherwise repeat the procedure.

2.10 Work Flow Of General Registration Process

The algorithmic steps for a general registration problem are simple. At the first step both datasets should be preprocessed to remove outliers, distant points, and noise, then key points should be extracted. The remaining points should be matched some how between tow clouds. This can be done by assigning features to each point representing data of the surface and finding the best feature matches. These matches are called correspondences. Correspondence between two points means that they represent the same point on the same surface in two different point cloud views.
Not all correspondences are correct due to different reasons such as noise or similarities between surfaces, thus a rejection method is essential. This step is so important since bad correspondences affect the registration result negatively. This task is performed during the correspondence rejection step. Final and desired rotation and translation matrices are then calculated according to the remaining correspondences. Figure 2.16 shows a flowchart representation of these steps.
A simple registration application will end at this point, but in our case we do not stop here. As it was mentioned before, the final goal of our project is for the mobile machine to track the pallet and safely approach it for lifting. This involves additional steps and specially a closed loop control system for the visual servoing part. In the flowchart we have added a conditional step in which we choose next steps depending on whether we are in the open loop stage or not.
In our proposed algorithm we run the illustrated simple registration phase only once. This is performed at the beginning of the program when there is no data about the position of a pallet or even its existence. Hence data is acquired from a wider laser angle interval. When the first step has finished processing, the output is the position and orientation of the pallet. At this point the program switches to the closed loop, limiting the laser sweep interval to the front area of the pallet. Since in the closed loop phase we only use ICP for registration, we have added a conditional step before computing the key points and after a scene cloud was acquired. If we are in the open loop we proceed to key points extraction and using of RANSAC algorithm, but if in closed loop phase, we jump to the ICP step as shown in the flowchart.
As it was described in the few previous sections, there exists two main methods to perform the registration task in PCL. One will include estimating correspondences and using one of the correspondence rejection functions to acquire the best correspondences. The other method involves the use of SAC-IA to estimate an initial alignment guess which will estimate and reject correspondences at the same time and with one single step. In both cases the results will be refined with ICP. We will implement both these methods and compare their results in detail in the results chapter.

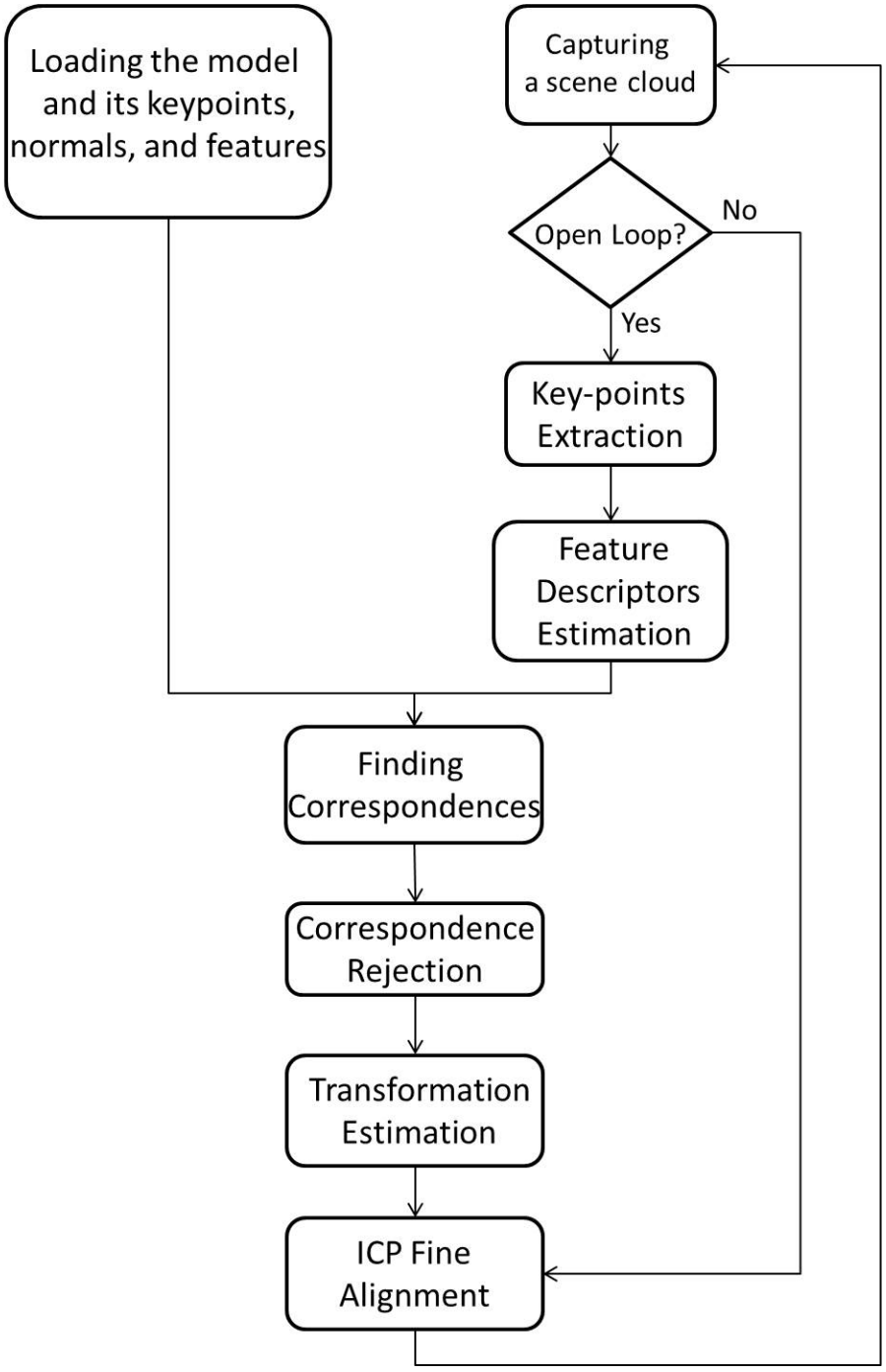


Figure 2.16: Registration process flowchart

3. IMPLEMENTATION

3.1 Overview

This chapter will cover different aspects of implementing the project and use of different previously introduced tools to make an executable project. This starts with getting PCL installed and executed on a compatible system, along with other necessary libraries. This topic is the subject of sections 3.2 and 3.3.
The communication platform of the project is UDP. This is an extremely important part of the implementing phase and needs to be done clearly and accurately. UDP protocol is the bridge between different modules, some running C++ codes on Linux, and some running on xPC target. Thus in section 3.4, UDP communication basics in these environments is covered. Sending and receiving data would also be addressed there.
One of the most important practical aspects is the production of a point cloud. This will affect all the later calculations since it will relate to the define coordinate frames and how range data from the laser system will be transformed to 3D coordinate information. We will tackle this topic in section 3.5. The topic for section 3.6. is realizing a control signal to supervise the laser servo system, and saving point cloud data in different desired intervals. This section will as well cover the servo controller method used in different phases.
We will be testing all different parts of our designed model on the GIMsim simulator machine where we have the laser scanner, laser servo controller, pallet, and machine's mechanic and hydraulic, and the environment simulated accurately. The simulator system in IHA simulation lab is composed of several PCs running IO and transducer models, terrain and collision model, machine dynamics, simulation GUI, and Autonomous functions. A clear view of the whole system implemented in Simulink is represented in section 3.7. At this point we will be having a completely implemented working project ready to be tested and analysed for performance.

3.2 Getting PCL

Getting PCL and installing it on your system has been made easy by documenting step by step guidance to the process[1]. It runs on many operating systems, and pre-built binaries are available for Linux, Windows, and Mac OS X. In addition to installing PCL, it is necessary to download and compile a set of 3rd party libraries that PCL requires in order to function. By selecting the operating system of your choice you may continue and download PCL for your system. PCL also provides the 3D processing pipeline for ROS, so it is possible to get the perception stack as well and still use PCL standalone.
As PCL is continuously developing and new features are dynamically added to its new versions, or if it does not provide pre-built binaries for a specific operating system, it is also possible to compile the library from source. Our choice of operating system is the latest version of Linux Ubuntu 13.04 and we have downloaded PCL source code and compiled it according to instructions.

3.3 Programming Software and Dependencies

PCL also depends on Boost, Eigen, FLANN, and VTK, which are required in order for that particular PCL library to compile and function. All the modules and algorithms in PCL pass data around using Boost[2] shared pointers, thus avoiding the need to re-copy data that is already present in the system. Boost provides free peer-reviewed portable C++ source libraries. It emphasizes libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use.
Most mathematical operations in PCL are implemented with and based on Eigen[3], an open-source C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. The backbone for fast neighbour search operations is provided by FLANN[4] (Fast Library for Approximate Nearest Neighbours). FLANN is a library for performing fast approximate nearest neighbour searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbour search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. FLANN is written in C++ and contains bindings for C, MATLAB and Python.

[1] http://pointclouds.org/downloads/
[2] http://www.boost.org/
[3] http://eigen.tuxfamily.org
[4] http://www.cs.ubc.ca/research/flann/

PCL also comes with its own visualization library, based on VTK[5]. The Visualization Tool Kit (VTK) is an open-source, freely available software system for 3D computer graphics, image processing and visualization which consists of a C++ class library and several interpreted interface layers including Tcl/Tk, Java, and Python. VTK supports a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods; and advanced modeling techniques such as implicit modelling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. VTK has an extensive information visualization framework, has a suite of 3D interaction widgets, supports parallel processing, and integrates with various databases on GUI tool kits such as Qt and Tk. In addition VTK is cross-platform and runs on Linux, Windows, Mac and Unix platforms.
More over PCL provides support for OpenMP[6] and Intel Threading Building Blocks (TBB) library[7] for multi-core parallelization. From the execution point of view PCL relies on CMake[8], a cross-platform, open-source build system as a build tool. CMake is a family of tools designed to build, test and package software, and it is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. Use of this build tool just requires placing a file called CMakeLists.txt in the same project folder and running the "make" command to compile path.

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(MY_GRAND_PROJECT)
find_package(PCL 1.3 REQUIRED COMPONENTS common io)
include_directories($PCL_INCLUDE_DIRS)
link_directories($PCL_LIBRARY_DIRS)
add_definitions($PCL_DEFINITIONS)
add_executable(pcd_write_test pcd_write.cpp)
target_link_libraries(pcd_write_test $PCL_COMMON_LIBRARIES
$PCL_IO_LIBRARIES)
```

Figure 3.1: CMakeLists.txt example

Having installed all the necessary depending libraries and CMake, the last interface needed is any suitable programming language editor of choice to start and write the main program. In this project the widely used programming environment

[5] http://www.vtk.org/
[6] http://openmp.org
[7] http://threadingbuildingblocks.org/
[8] http://www.cmake.org/

of Eclipse[9] is used, which is provided by the Linux software center as well. The output of this part is a file with the extension .cpp, the standards format for C++ code programs. Now it is enough just to place the CmakeLists.txt file along with the .cpp file in the same folder and use a single simple Linux command to compile the program and make an executable application.

3.4 UDP Communication

UDP is the main communication platform that needs implementation in this project. With the User Datagram Protocol (UDP) computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network without prior communications to set up special transmission channels or data paths.
UDP uses a simple transmission model with a minimum of protocol mechanism. It has no handshaking dialogues, and thus exposes any unreliability of the underlying network protocol to the user's program. As this is normally IP over unreliable media, there is no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram. UDP is suitable for purposes where error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

3.4.1 UDP Communication In MATLAB Simulink

As described previously the simulation system consists of different modules that need to communicate and share data. In this specific project we need to communicate between different modules to read in point cloud data, navigation data, laser servo, and to send control signals to the laser servo and detection results to the controller, The controller is implemented in a MATLAB Simulink model, and our main program is running on a Linux machine, thus we need UDP communication programming both in Simulink environment and C++ under Linux. In the Simulink part this is done rather straight forward, since it only suffices to insert the corresponding blocks shown in Figure 3.2 and to set parameters in the windows shown in Figure 3.3.

The receive block has two outputs, one to hold the received data and the second one is an acknowledge signal sending a small pulse whenever a new data has

[9] http://www.eclipse.org/

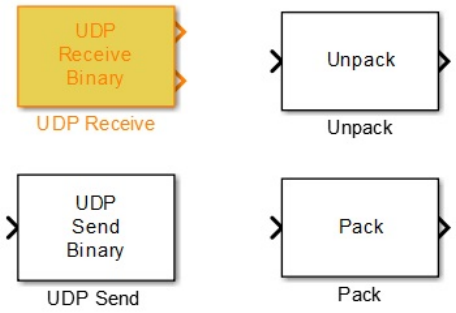
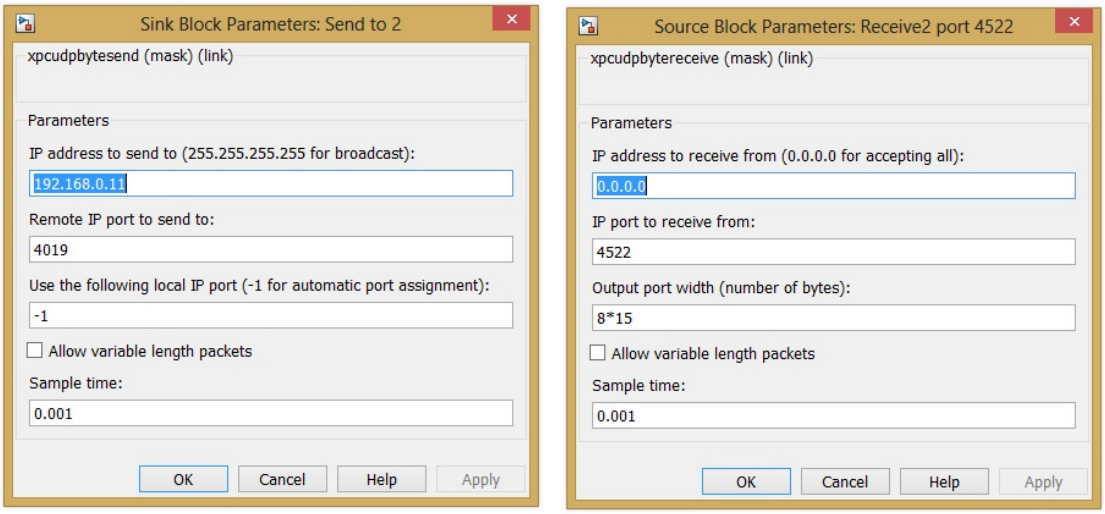


Figure 3.2: UDP send/ receive blocks in simulink



(a) Send block parameters (b) Receive block parameters

Figure 3.3: UDP blocks parameters

been received. The sending block only needs one input port to carry the data that needs to be sent. As mentioned UDP sends short messages called datagrams which comprise one message unit. To combine several doubles as one datagram, a "Pack" block is needed before the send block to pack all the data inside one package. The same applies to the other end, in order to extract individual data from a packet, the "Unpack" block is utilized after a receive block. In addition there are parameters for each block that should be set in order to complete the send or receive function. Figure 3.3 shows parameter setting windows for both blocks. The left window corresponds to send block, which accepts an IP for the remote application (for single cast) while providing a choice for broadcasting as well. Assigning a port number is of great importance, since the end application will know where to read the data from. This is specially important in UDP protocol since as noted this protocol is connection less, and the ports are the only means to provide a rout between two end points. Finally there are two fields for the IP of the local PC and the sample time,

which indicates the frequency for sending datagrams. As for the receiving block there also exists a field to choose between accepting data from a specific IP or from any user who uses the specified port number in the second field. Finally comes the sample time which indicates how frequently to read from or write to the port, and before that the length of the received datagram is specified in bytes for the receive block. UDP discards messages which don't match this length, so it is necessary to know the length of the transmitted datagrams in order to read them in correctly.

3.4.2 UDP Communication With C++ On Linux

In the C++ context, we need to define sockets and set desired specifications for them. In our program two types of sockets are needed, one for sending and one for receiving data. Defining a socket has a typical syntax and involves a few steps to create and identify it. A socket, `s`, is created with the socket system call `int s = socket(domain,type,protocol)` in which all the parameters as well as the return value are integers. Domain is the communication domain in which the socket should be created. Some of address families are `AF_INET` (IP), `AF_INET6` (IPv6), `AF_UNIX` (local channel, similar to pipes), `AF_ISO` (ISO protocols), and `AF_NS` (Xerox Network Systems protocols). Type is the type of service which is selected according to the properties required by the application; `SOCK_STREAM` (virtual circuit service), `SOCK_DGRAM` (datagram service), `SOCK_RAW` (direct IP service). One should check with his address family to see whether a particular service is available or not. Finally the protocol indicates a specific protocol to use in supporting the sockets operation. This is useful in cases where some families may have more than one protocol to support a given type of service. The return value of this call is a file descriptor (a small integer).
For UDP/IP sockets, we specify the IP address family as `AF_INET` and datagram service as `SOCK_DGRAM`. Since there is only one form of datagram service, there are no variations of the protocol, so the last argument, protocol, is `IPPROTO_UDP`.
Next is the identification of the socket. When we talk about identifying a socket, we are talking about assigning a transport address to the socket which is defined in a socket address structure. In sockets, this operation is called "binding" and the bind system call is utilized as:

```
int bind(int socket,const struct sockaddr *address,socklen_t address_len)
```

The first parameter is the `socket` that was created with the socket system call. The third parameter specifies the length of that structure, which is simply the size of the internet address structure, `sizeof(struct sockaddr_in)`. For the second parameter, the structure `sockaddr` is a generic container that just allows the oper-

ating system to be able to read the first couple of bytes that identify the address family. The address family determines what variant of the `sockaddr` struct to use that contains elements that make sense for that specific communication type. For IP networking, we use struct `sockaddr_in`, which is defined in the header `#include <sys/socket.h>`. This structure is defined as:

```
struct sockaddr_in {
                      __uint8_t sin_len;
                      sa_family_t sin_family;
                      in_port_t sin_port;
                      struct in_addr sin_addr;
                      char sin_zero[8];
                    };
```

Before calling bind, we need to fill out this structure in which there are three key parts to set. `sin_family` is the address family we used when we set up the socket. In our case, it is `AF_INET`. `sin_port` is the port number which we may explicitly assign or allow the operating system to assign one. However, instead of simply copying the port number to this field, it is necessary to convert this to network byte order using the function htons() which converts a port number in host byte order to a port number in network byte order. The third field of `sockaddr_in` is a structure of type struct `in_addr` which contains only a single field unsigned long `s_addr`. This field contains the IP address of the host. For server code, this will always be the IP address of the machine on which the server is running, and there is a symbolic constant `INADDR_ANY` which gets this address. It is also needed to call the `memset` function for the socket to fill a block of memory. According to these descriptions a small part of our code creating a socket to read point cloud data from port number 4599 looks like this:

```
int pointcloud_read_socket=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
sockaddr_in pointcloud_read_client;
socklen_t pointcloud_read_client_size = sizeof(pointcloud_read_client);
if( pointcloud_read_socket < 0 )
cout<< "Creating point cloud read socket error" << endl;
memset(&pointcloud_read_client, 0, pointcloud_read_client_size);
pointcloud_read_client.sin_family = AF_INET;
pointcloud_read_client.sin_port = htons(4599);
pointcloud_read_client.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(pointcloud_read_socket, (sockaddr*)&pointcloud_read_client,
                                  sizeof(pointcloud_read_client)) < 0)
cout<< "Binding point cloud read socket error" << endl;
```

This piece of code is repeated in our code four times to create one socket named `laser_send_client` to send the initial values for laser angle to port number 4019, one named `control_signal_socket` to read in the control signal from port 4545 sent from Simulink every $5ms$ (this will be discussed in the next section), one named `pointcloud_read_socket` to read in the actual point data from port 4599 as described above, and finally a socket named `registration_send_client` to send the registration results on port 4522.
Up to this point there is no difference between sockets that are going to be used for sending or receiving, except for two minor modifications: first, for sending function it is necessary to know the address of the machine which is receiving the data (if not broadcasting), so we used the IP address of the destination instead of `htonl(INADDR_ANY)`, as follows:

```
registration_send_client.sin_addr.s_addr = inet_addr("192.168.0.39");
```

And the second modification is about `bind()` function call. It is possible to call `bind()` after the call to `socket()`, if we wish to specify which port and interface should be used for the sending socket. However, this is almost never necessary since the system will decide what port and interface to use. So the call to this function has been skipped for those sockets that we want to use for sending.
Now what makes the difference is the use of two important functions,`sendto()` and `recvfrom()` which are used to send and receive data respectively. The `sendto()` function has the following syntax:

```
int sendto(int socket,const void *buffer,size_t length,int flags,
           const struct sockaddr *dest_addr,socklen_t dest_len);
```

The first parameter, `socket`, is the socket that was created with the socket system call and named. The second parameter provides the message we want to send followed by the third parameter which it it's length. The flags parameter is 0 and not useful for UDP sockets. The `dest_addr` defines the destination address and port number for the message and uses the same `sockaddr_in` structure that we used in bind to identify our local address. As with bind, the final parameter is simply the length of the address structure: `sizeof(struct sockaddr_in)`. The return value from `sendto` only indicates if the packet was successfully sent from the local computer and no information about whether or not the packet was received by the destination or not.
Once you have a UDP socket bound to a port, any UDP packets sent to your sockets IP address and port are placed in a queue. To receive packets just loop and call `recvfrom` until it fails indicating there are no more packets left in the queue. The `recvfrom()` function has the following syntax:

```
int recvfrom(int socket, void *restrict buffer, size_t length,
             int flags,struct sockaddr *restrict src_addr,
                       socklen_t *restrict *src_len)
```

The first parameter is the socket that we created. The port number assigned to that socket via the bind call indicates on what port `recvfrom` will wait for data. The flags parameter allows processing out-of-band data, peek at an incoming message without removing it from the queue, or block until the request is fully satisfied. We can safely ignore these and use 0 instead, specially in this application we are not in need of flags. The `src_addr` parameter is a pointer to a `sockaddr` structure that we allocate and will be filled in by `recvfrom` to identify the sender of the message. The length of this structure will be stored in `src_len`. If it is not required to identify the sender, it is possible to set both of these to zero, but then there is no way to reply to the sender. The `recvfrom` call returns the number of bytes that were read into buffer, or -1 in case of failure.
There exists one other important behaviour of the `recvfrom` function, and that is the behaviour which is called "blocking" or "non-blocking". Blocking refers to the case in which the receiving function will block and wait for a packet to arrive. On the other hand, in the non-blocking mode it will wait for a user defined amount of time and then returns with a value 0 if there was no data available to read, or if there is no time specified, it will immediately return in this case. When the argument flags of `recvfrom()` is set to `MSG_NOBLOCK`, the function does not block if there is no data to be read, but returns immediately with a return value of 0 bytes. It is notable that in this application we did not use this behaviour and all the reading sockets are in blocking mode as in their default.

3.4.3 Saving Point Cloud Data Through UDP

Now that we have explained the basics of socket programming in C++ for a UDP communication, it is time to look at an important piece of our code which uses a receive function iteratively to read and save the point cloud data. This code has been pasted from the closed loop phase. The open loop phase is exactly the same, but without the while loop to run only once. The reading and saving works as follows; first a single double which is the control signal is read in. This signal is described in detail in the next section, but for now it is enough to know that this is a condition signal to control the read and save of data. If this value is one we proceed to read one row of the point cloud data, and we do not read if otherwise.

```
while (true)
```

```
{
    recvfrom (control_signal_socket, control_signal_value,
              sizeof(control_signal_value), 0,
              (sockaddr*)&control_signal_client,
              &control_signal_client_size);
    if (control_signal_value[0] == 1)
        {
            recvfrom(pointcloud_read_socket, &row_package1,
                     sizeof(row_package1),0,
                     (sockaddr*)&pointcloud_read_client,
                     &pointcloud_read_client_size);
            udpVectorData1.push_back(row_package1);
            number_of_rows1 ++;
        }
    if (control_signal_value[0] == 0 & number_of_rows1!= 0)
        {
            cout << "number of rows:" << number_of_rows1 << endl;
            break;
        }
}
```

Note that both receive functions are in blocking mode, so the first function which is reading the control signal, waits there until a data is available. This signal is coming every $5ms$, so if we are lucky not to loose it, this waiting should not take more than that. If this value is equal to one, another receive function is called. This function is in blocking mode as well, but there is a significantly important difference in between. The control signal is sent to this socket as unicast, and unicast data will be placed in a structure so called a queue. Once read the package will be removed from the queue and there will not be any data available before another $5ms$. In contrast the point cloud data are sent as broadcast and they will not be removed when read. In fact there is no meaning for a queue for broadcasting, since every user should be able to log and receive the broadcast data. This means that even though these data are coming every $20ms$, but the second receive will not need to wait this time to read a data, the previous data is always available in between these 20 milliseconds and will not be removed even if it is read several times. With a simple division it can be seen that we will be reading the same data four times repeatedly before the new data is available after a $20ms$.
This is a good strategy we have implemented to compensate the unreliable nature of UDP. In other words, although the reading process is much faster than $20ms$, but this is a limitation since no new data will be available before this time. Experience shows that reading only once during this $20ms$ will decrease the chance of receiving the package, and a great number of data were lost. It is notable that this way the reliability of reading data correctly is increased without loosing any time, which is

desirable.
One row of point cloud data consists of 721 double values, consisting of a single "sweep" value, 180 doubles of "variance", and 180 "point data" each having three values for x, y, and z. We have defined two structures to save in data and to ease accessing of the elements. The first structure which is called LaserData holds coordinate values, and the second structure, UDPData, is a structure of structure and holds the value of sweep, variances and its third element is 180 of LaserData type, as shown below. An object of type UDPData is then defined and called `row_package1` which is basically one row which will be read at each call of receive function. Finally a vector called udpVectorData1 of the type UDPData will hold all the rows to form a complete point cloud, and this is done by the `push_back` function. In addition an integer `number_of_rows1` will decrease by one each time a row is saved in that vector.

```
struct LaserData
    {
        double x;
        double y;
        double z;
    };
struct UDPData
    {
        double sweep;
        double variance[180];
        LaserData data[180];
    };
vector <UDPData> udpVectorData1;
UDPData row_package1;
```

There still is another step to complete. You can see that the while loop which iterates and saves data in a vector has a condition of always true, so we need to give it a condition to break. This is satisfied when both `control_signal_value` is zero and `number_of_rows1` is not zero. This guarantees that the control signal has had a falling edge, meaning it has fallen to zero after a one and saving some data, and not being in zero before saving any points when it has not even reached there. At this point the reading from UDP socket is finished and all data which we were allowed to read during the high period of control signal is saved in a vector, ready to be transformed to a PCL usable format. The relative piece of code which transforms these data to a PCL compatible point cloud form is pasted here. This is a simple iteration to go through the vector and access the coordinates and save them in a pointer called cloud which will be usable by PCL for further processing.

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud
   (new pcl::PointCloud<pcl::PointXYZ>());
int index = 0;
cloud -> width    = number_of_rows * 180;
cloud -> height   = 1;
cloud -> is_dense = false;
cloud -> points.resize (cloud -> width * cloud -> height);
for (int i = 0; i < udpVectorData.size(); i++)
 {
    for (int u = 0; u < 180; u++)
        {
           cloud -> points[index+u].x = udpVectorData.at(i).data[u].x;
           cloud -> points[index+u].y = udpVectorData.at(i).data[u].y;
           cloud -> points[index+u].z = udpVectorData.at(i).data[u].z;
        }
    index = index + 180;
 }
```

Figure 3.4 summarizes all UDP communication performed in our algorithm in a schematic representation.

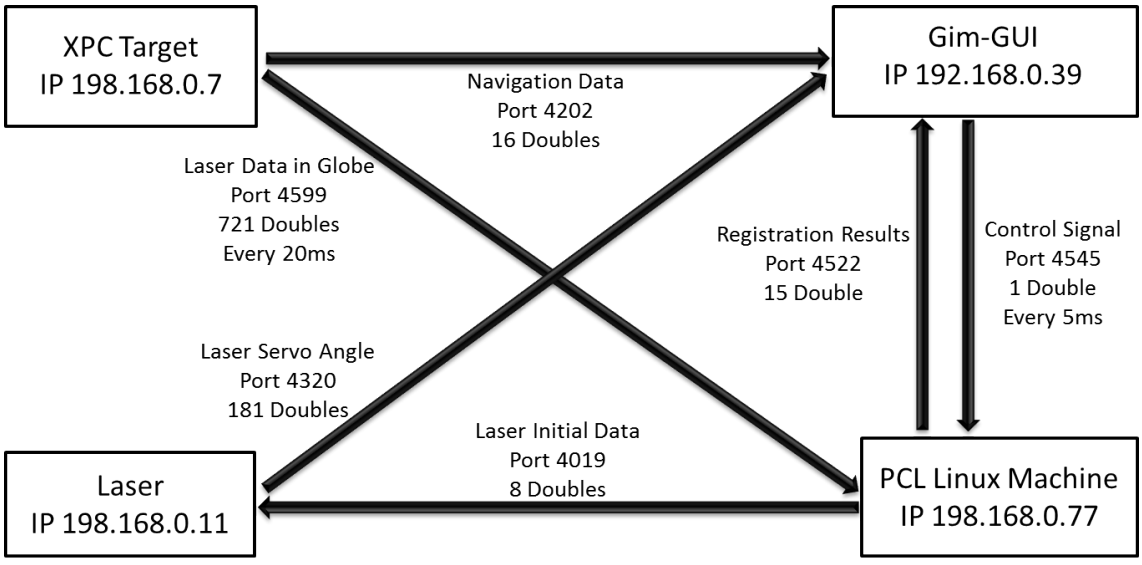


Figure 3.4: UDP communication schematic diagram of the system

3.5 Point Cloud Production

Up to this point, we have always considered that a ready point cloud in some format is available from a port, holding points coordinates with respect to a predefined coordinate system. We have also mentioned that what a laser scanner measures is the distance between the beam origin and the detected point. So it is obvious that these ranges must undergo some important processes in order to get point coordinates. Here we will discuss how this is implemented in our project.
Recall section 1.5 where we stated that the LMS scans the perimeter of its surroundings in a plane with the aid of laser beams and it measures its surrounding in two-dimensional polar coordinates. This measuring principle is shown in Figure 3.5.

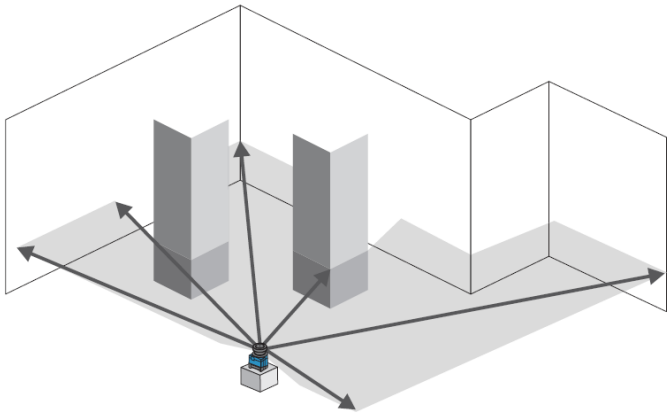


Figure 3.5: Measurement principle of LMS111 laser scanner

This type of laser measurement systems are inherently 2D, in the sense that they usually combine a laser range finder with some other techniques to measure distances on a plane. To obtain a 3D point cloud, they are placed on a rotating unit. Using the kinematics of this unit, we can obtain a multitude of such 2D measurement planes, which can then be converted into a consistent 3D representation. In this case we have a servo system controlling the tilt of the laser beam, and enabling the sweep. Figure 3.6 shows our laser scanner with a fixture which will mount to the machine body. The fixture is fixed to the machine and the sensor will rotate along the Y axis.

In the system converting range data to a 3D cloud we have a set of assigned coordinate frames that needs to be clearly described. The set starts with a frame called F_{Sensor} assigned to the laser beam rotating around its Y axis while sweeping. Another frame is assigned to the fixture called F_{Laser}. This frame is fixed to the fixture, and F_{Sensor} rotates with respect to that. These two frames are clearly shown in Figure 3.6.
Another frame is assigned to the machine body which is called F_{Body}, and finally the

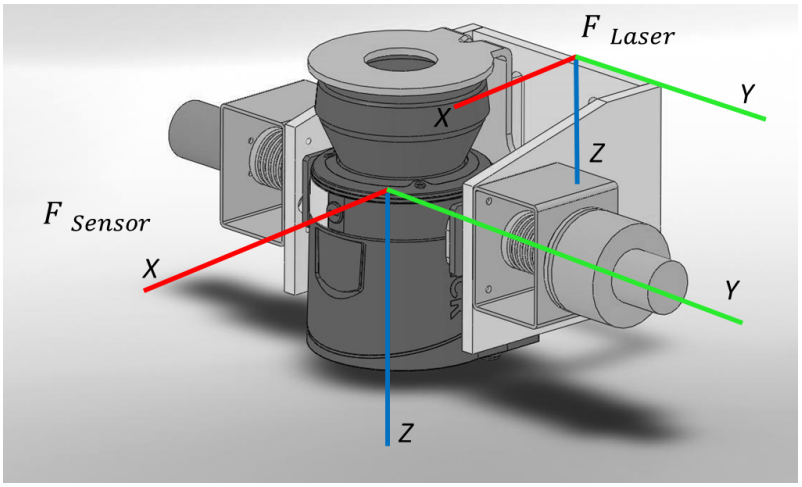


Figure 3.6: Laser beam and body coordinate frames

global frame F_{World} which is fixed to a certain non moving point and we would like to have our data in that frame eventually. In Figure 3.7 all these frames are shown with respect to one another. Note the agreement in this document, x axis will be shown in red, y will be in green, and z axis will be blue. This is in accordance with the PCL visualizer and will also apply to feature point clouds we represent in the results chapter.

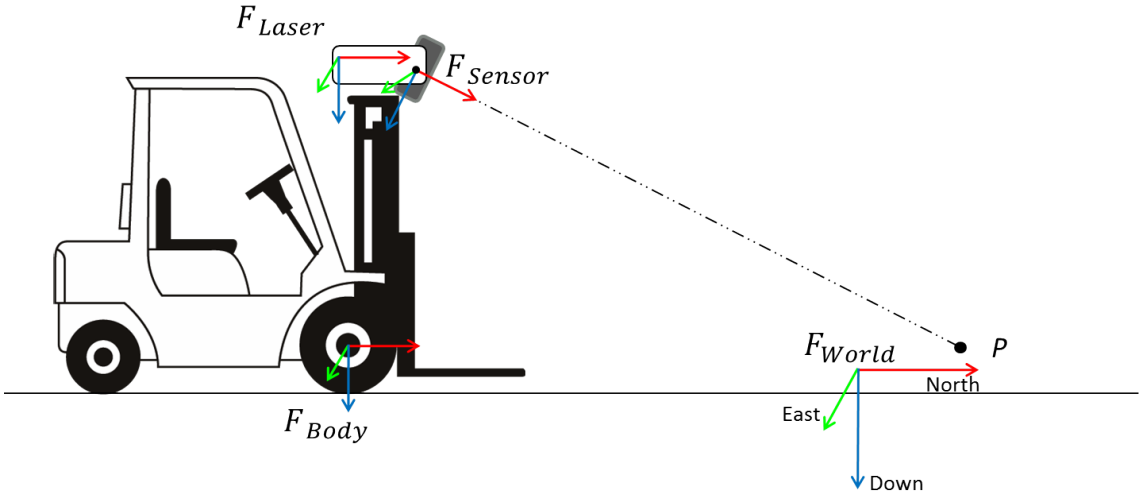


Figure 3.7: All assigned frames

Each point P_{Sensor} with coordinates measured in the sensor frame can be transformed to the global world frame with a transformation matrix as follows

$$P_{World} = T^{World}_{Sensor}(t)\ P_{Sensor}$$

Now let us see how a point coordinate is found in this system with defined frames. The laser sensor emits 180 laser beams with each two subsequent beams having

0.0087 degree in between as it is shown in Figure 3.8. For each single emitted beam with known angle α, the distance d is measured as described in section 1.5. It is then a simple geometry calculation as done in the figure to get two coordinates x and y in the laser beam frame. The z coordinate value will obviously be zero for all points measured in the sensor coordinate frame.

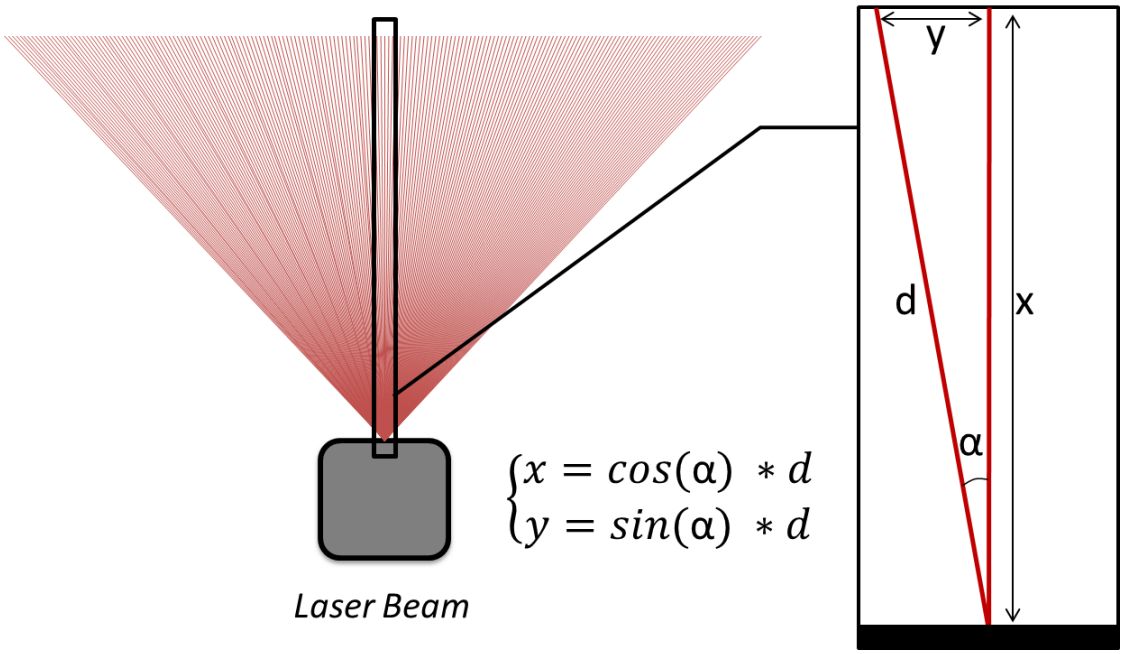


Figure 3.8: Laser beam and point coordinates measured in the sensor coordinate frame

The coordinate values $(x, y, 0)$ in the laser beam frame can now be transformed to the global frame with a series of transformation matrices through the laser body, and machine coordinate frames. In other words this will be a series of matrix products. Assume that the transformation matrix from the laser beam to the laser body is denoted as T^{Laser}_{Sensor}, the one from laser body to the machine is denoted as T^{Body}_{Laser}, and the one from the machine to the global frame is denoted as T^{World}_{Body}. Thus a transformation matrix from the laser beam to the global frame can be written as

$$T^{World}_{Sensor}(t) \;=\; T^{World}_{Body}(t)\; T^{Body}_{Laser}\; T^{Laser}_{Sensor}$$

In the above equation, T^{World}_{Body} is known based on the navigation data, and T^{Body}_{Laser} is a fixed matrix in accordance with the implementation. Also T^{Laser}_{Sensor} will be calculated based on the rotation angle of the servo unit. The laser beam frame rotates with respect to the laser body frame and. Now each point P_{Sensor} with coordinates $(x, y, 0)$ measured in the laser beam frame can be transformed to the global world frame as follows

$$P_{World} = T_{Sensor}^{World}(t)\ P_{Sensor}$$

P_{Sensor} is a 3*1 matrix, while T_{Sensor}^{World} is 4*4. Thus a row of ones is added to the end of P_{Sensor} for the production to be done correctly.

3.6 Servo Control

Designing a controller to control the laser scanner servo motor to sweep as desired, is an important task to be done at this stage. To simply describe the complete scenario, assume that a program calls for the procedure to execute, which is in fact a higher level control mechanism. As a result gathering required data for the first time will start.

In this first scan, which we call it "open loop" phase, the aim is to acquire information from a wider angle of view, since there is no information about the possible position of a pallet or even the existence of one. This point cloud is saved during a complete laser sweep and fed to the main program written using PCL. Here is where all the previously described procedures and algorithms will take place, and assuming there is at least one object present in the captured scene, the result will be a 4*4 translation matrix showing the relative position of the found object in the scene to the model in its data base.

At this point the second step commenced which we will name "closed loop" detection phase. In this phase the result matrix of the open loop phase is given to the servo controller and it is used there to control the movement of the machine towards the pallet. As in this step there is relatively accurate information of the pallet position with respect to the machine, we will be able to control the laser servo system to focus on a specific desired area. This results in smaller scan angle intervals, smaller point clouds and less processing time. As the machine starts moving towards the pallet, we continue focusing on the goal area, capturing point clouds, feeding it to PCL program, and feeding the result back to the controller in order to close our feedback control loop and correct the machine's path towards the goal.

The already existing simulated model accepts a package of eight double values to control the servo mechanism; the first five values are zero, the sixth value is at 1.85 and the seventh value is set at -0.6. The first value is the current height and the second one is the current position of the laser with respect to the machine frame. These configurations allows to change the location of laser on the body, thus enabling study the effect of calibration accuracy. The laser servo can be in position mode or velocity mode control as well.

Finally the last value will be the actual angle we want the laser to be positioned at. There is however an exception; if the constant -2000 is given, the laser will start

sweeping continuously and periodically. This pack of eight has to be send to port number 4019 and to the machine with IP address "192.168.0.11". As the code we are writing should be in accordance with the hardware and interface design, we should make sure to pass relevant data in each stage.

3.6.1 Open Loop Phase

At the beginning of our code, we send the constant -2000 which causes continuous periodic sweep on the laser scanner. Since we are not focusing on any specific area, it will suffice to record data gathered during one sweep which means only one sweep needs to be masked out. This masked sweep will be the only period in which we need to record the point cloud data during the open loop phase. We will call this signal the "control signal" since it will control the point cloud read and saving process and plays a significant role in the designed system. Of course this signal needs to be appropriately merged with the closed loop phase control signal as well, in order to enable reading and saving point clouds in both stages. The logic implementation and generation of this signal will be clearly discussed in section 3.6. But for now this signal remains one during one complete sweep after the program has commenced.
The control signal will be send to the Linux machine every $5ms$ where we will be reading it inside our code and check its value. The case is then investigated, if the value is one we proceed to read and save a single row from the point cloud, and if it is zero we skip reading and saving. This procedure is repeated inside a while loop until a zero is detected after a one, which indicates a falling edge in the control signal and the loop is then terminated.

3.6.2 Closed Loop Phase

In the previous section we created the control signal for open loop phase. When the program starts running, it will read in the value of this signal and saves the point cloud when it is high. The point cloud processing is then started using PCL and the resulting 4*4 translation matrix is returned.
Now in the closed loop detection phase we need to focus the laser sweep over a small portion of the the pallet. This is a wise decision to avoid capturing unnecessary data, as it is clear that for detecting the position and orientation of the pallet, it will be enough to locate the front of the pallet. Since the machine is moving at this stage, motion dynamics of the machine should also be considered. This motion of the machine means that the lower and upper limits of the servo angle will change with time, as the distance between the pallet and the machine changes with time.

For a better understanding, consider Figure 3.9 which shows the relative position of the machine, pallet and laser scanner. In this schematic, b indicates the distance between the machine and the pallet. From a simple geometric calculation we have

$$\begin{cases} \alpha = Arctan(1.85/(b)) \\ \\ \beta = Arctan(1.85/(b+0.4)) \end{cases}$$

in which $1.85m$ is the height of the laser scanner and $0.4m$ is the focus area. It can be seen from the above equations that both α and β depend on the variable b. The position of the machine in the world frame is known from odometers and the position of the pallet in the world frame is also known from each registration, thus the distance between the pallet and the machine is known for all the times. According to the above equation, knowing this variable will result in knowing the upper and lower limit angles in every instance. Comparing the current servo angle with the upper limit (when moving upwards) or lower limit (when moving downwards) we find the direction switching points.

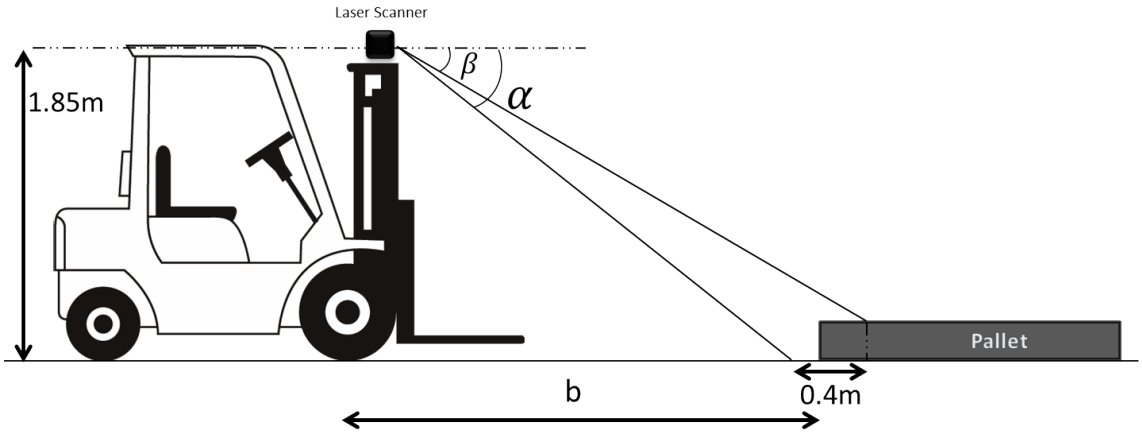


Figure 3.9: Relative position of the machine, pallet and laser scanner

3.7 Overview Of The Simulink Model

At this point all the implementation steps are done and our Simulink model is complete and in accordance with all hardware requirement and communication protocols. Now we may have a look at the final model and gain a more clear view, before proceeding to our tests. The most upper level view of the whole system is shown in Figure 3.10. The system consists of four main subsystems. All the controllers are implemented in the subsystem called controller.

The PCL registration subsystem holds blocks for receiving registration results from

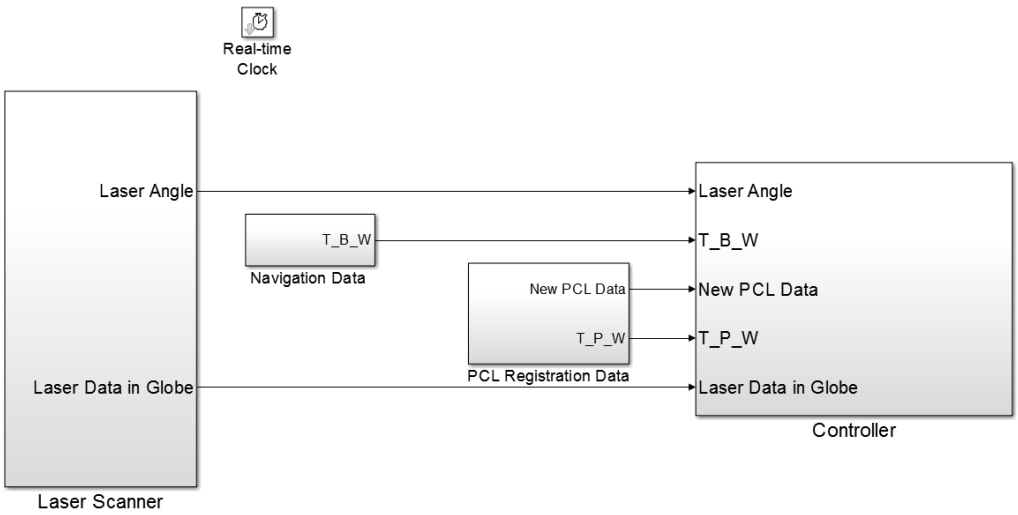


Figure 3.10: Complete model overview

the PCL machine and extracting the rotation and transformation matrices out of them. It has two outputs, the first is an acknowledge signal giving a short pulse whenever a new data has been received, and the second one named `T_P_W` which is basically the translation matrix of the pallet to the world coordinate frame. Figure 3.11 shows the contents of this subsystem, containing a laser vision block shown in Figure 3.12. Recall the package sent from PCL to this model has 15 double values with the first and the two last values being zero. Unpacking and extracting matrices R and P take place in the laser vision block and the MATLAB function in the PCL registration data subsystem just concatenates them to form a desired 4*4 matrix `T_P_W`.

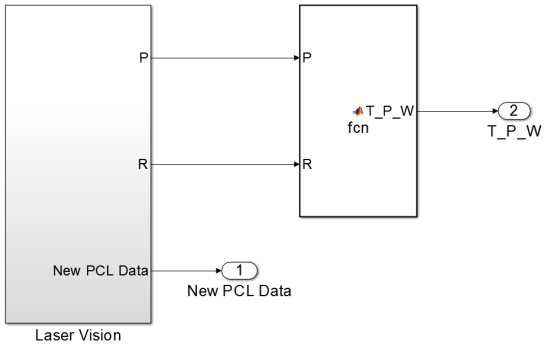


Figure 3.11: PCL registration data subsystem

The second subsystem in the model is the laser scanner subsystem shown in Figure 3.13. It holds two UDP receive blocks to receive point cloud data in the global coordinate frame, and also some other information about the laser angle, being processed in a subsystem to extract the exact laser angle.

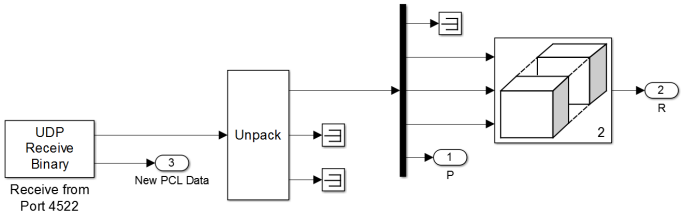


Figure 3.12: Laser vision subsystem

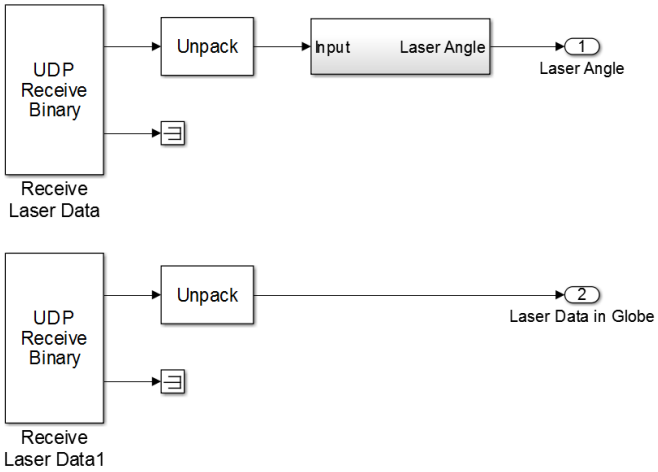


Figure 3.13: Laser scanner subsystem

The third block in the main system is the navigation data subsystem, which is shown in Figure 3.14. This package of navigation data comes from port number 4202 and it contains 16 doubles in the form of:
$\{North, East, Down, Roll, Pitch, Yaw, v_x, v_y, v_z, w_x, w_y, w_z, odo_x, odo_y, odo_z, Time\}$
received from a UDP receive bock and extracted by unpacking. What is desired for us is to extract the complete translation matrix for the machine body to the world coordinate frame. For this goal, we are going to be using $\{North, East, Down\}$ values to get the position vector, and $\{Roll, Pitch, Yaw\}$ values to get the rotation matrix of the machine, and the rest of the data is of no use to us. In the MATLAB function, the R matrix is generated from three angles according to the following formula:

$$\begin{bmatrix} cos(p)cos(y) & cos(p)sin(y) & -sin(p) \\ sin(r)sin(p)cos(y) - cos(r)sin(y) & sin(r)sin(p)sin(y) + cos(r)cos(y) & sin(r)cos(p) \\ cos(r)sin(p)cos(y) + sin(r)sin(y) & cos(r)sin(p)sin(y) - sin(r)cos(y) & cos(r)cos(p) \end{bmatrix}$$

This matrix is then concatenated with the position. The process is shown in Figure 3.14 and finally the 4*4 matrix of `T_B_W` is outputted.

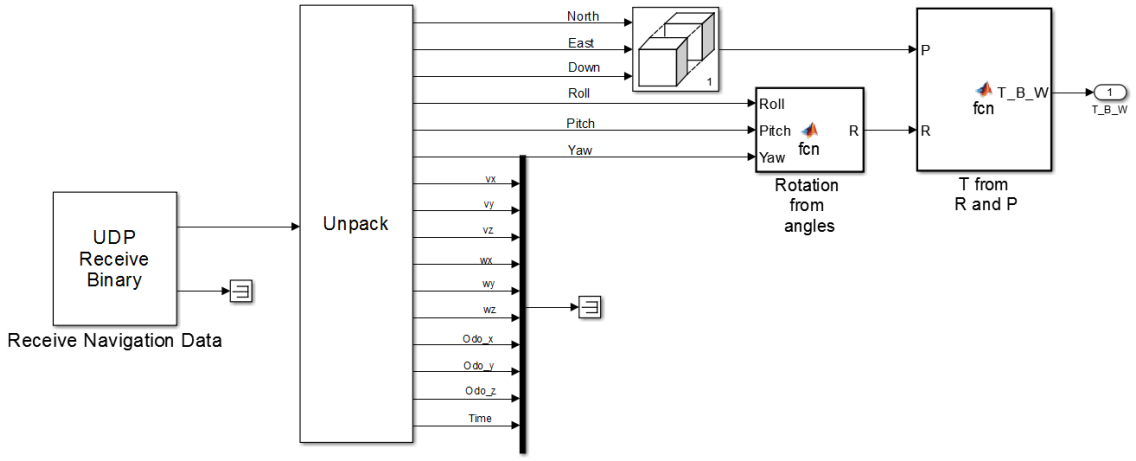


Figure 3.14: Extracting body to world transformation matrix from navigation data

The fourth and final subsystem shown in the main model is the controller which holds our designs for the previously described control signal. This block accepts the outputs of all three previously mentioned subsystems to perform its functionality. Figure 3.15 shows inside of this subsystem. The logic starts with getting the transformation matrix from the body to the world(`T_B_W`) from the navigation data, and the transformation matrix from the pallet to the world(`T_P_W`) from recognition result, and computing the transformation matrix from the pallet to the body(`T_P_B`). In order to get the correct result, the following formula should be applied:

$$T_B^P(t) \;=\; T_W^P \; T_B^W(t)$$

in which the t argument indicated the dependency on time. It can be seen from the above formula that the transpose of the `T_B_W` matrix is needed. The transpose of a 4*4 translation matrix is calculated according to the following formula:

$$M = \left[\begin{array}{ccc|c} & R & & P \\ \hline 0 & 0 & 0 & 1 \end{array}\right]^{-1} = \left[\begin{array}{ccc|c} & R^T & & -R^T\ P \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Element $(2, 4)$ of this matrix is the distance between the machine and the pallet. In section 3.6.2 we described how we need this distance value to dynamically focus on a small area of the pallet front, and here we can see the actual implementation in Simulink.
This distance value is also used to distinguish between open and closed loop phases. When the simulation starts, this distance is equal to zero, due to the fact that there is no information about the existence of a pallet. This value will only change if any detection result from the PCL machine causes this to happen. Thus comparing this

value to zero we can enable two different subsystems. Inside these two subsystems we produce the control signal and the output is logically "OR"ed to enable the control signal if the value of an of these signals is high.

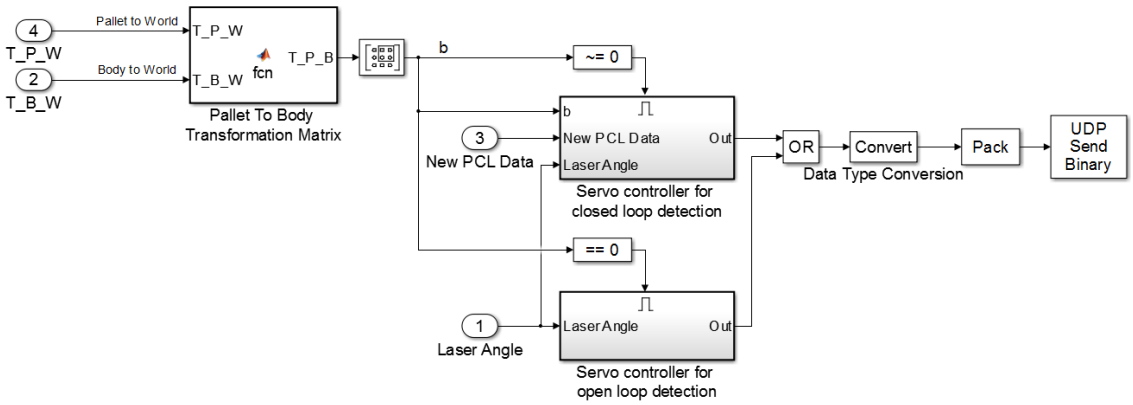


Figure 3.15: Controller subsystem

The subsystem shown lower in Figure 3.15 is related to the open loop phase. In section 3.6.1 we discussed how we need this signal to behave in order to mask only one cycle during an ordinary periodic sweep of the laser. Figure 3.16 shows the underlying logic and its implementation which has a simple logic and does not require any further explanation. To test the system Figure 3.17 shows the output of this subsystem to a periodic simple sinusoidal signal. The signals are plotted as value versus time. It can bee inferred that one sweep is masked in the lower signal and its value is high in that interval.

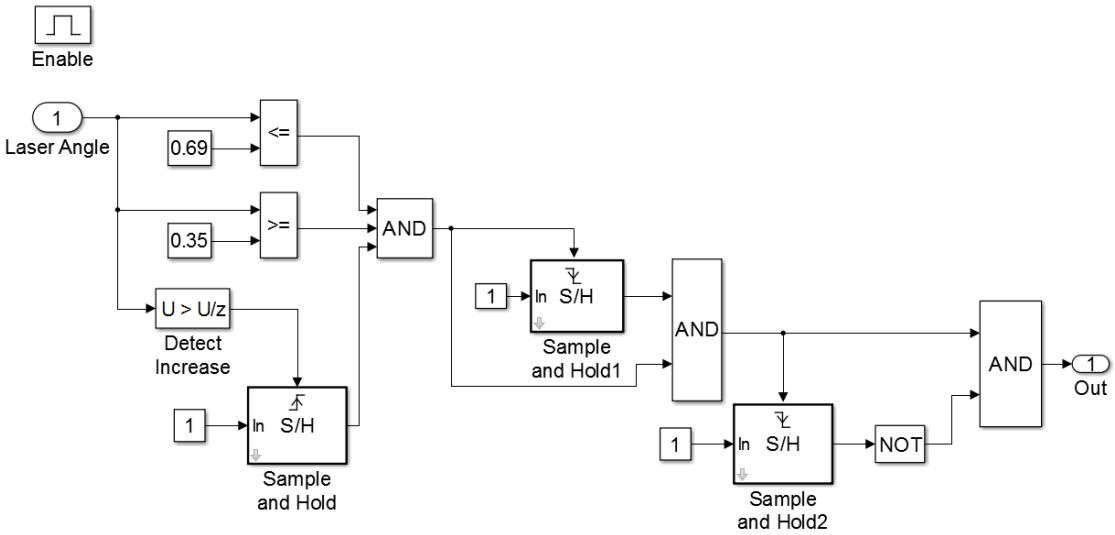


Figure 3.16: Servo controller for open loop detection

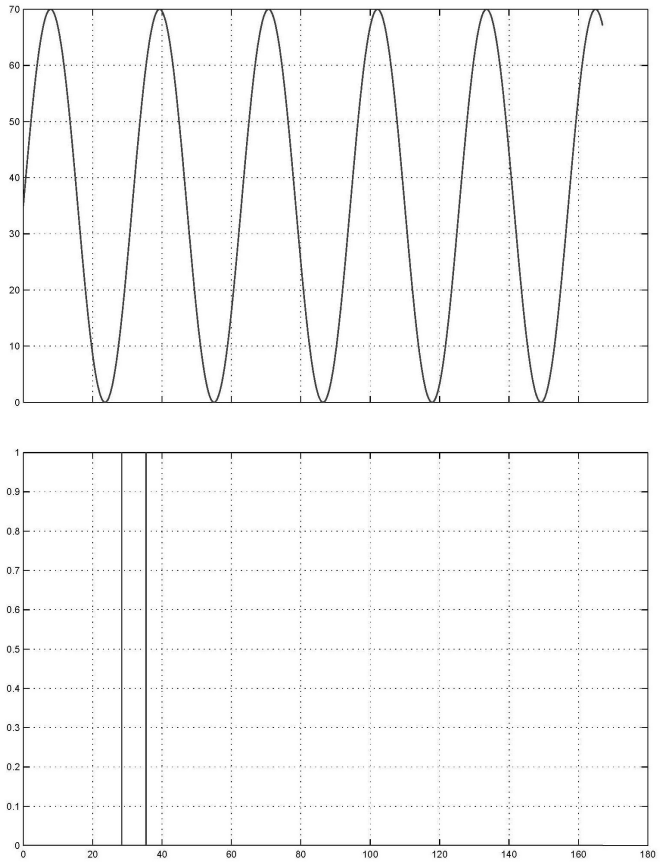


Figure 3.17: Open loop control signal

The other subsystem shown upper in Figure 3.15 is related to the closed loop phase discussed in section 3.6.2, and its contents are shown in Figure 3.18. One important note here is that in this phase we also need to produce the servo angle signal and send it to the laser servo. This is because here the angle should be dynamically controlled in accordance to the relative distance of pallet and machine. Thus there are two subsystems shown in Figure 3.18.

The goal is to make the laser sweep between two angles discussed in section 3.6.2; α and β. Hence a MATLAB function is added to produce these two values using the distance b and the previously discussed equations.

To produce the actual servo angle signal, we use a discrete time integrator. The idea is to provide two values to the integrator, one positive and one negative to cause increasing and decreasing in its output. For this purpose a controlled switch is used. Also the initial condition of the integrator is the exact laser angle at the time when the system switched from open loop to closed loop detection, to prevent any discontinuity in the laser motion.

As shown in Figure 3.19, investigating the output of the switch, we will know whether the angle is decreasing or increasing and based on that we compare the

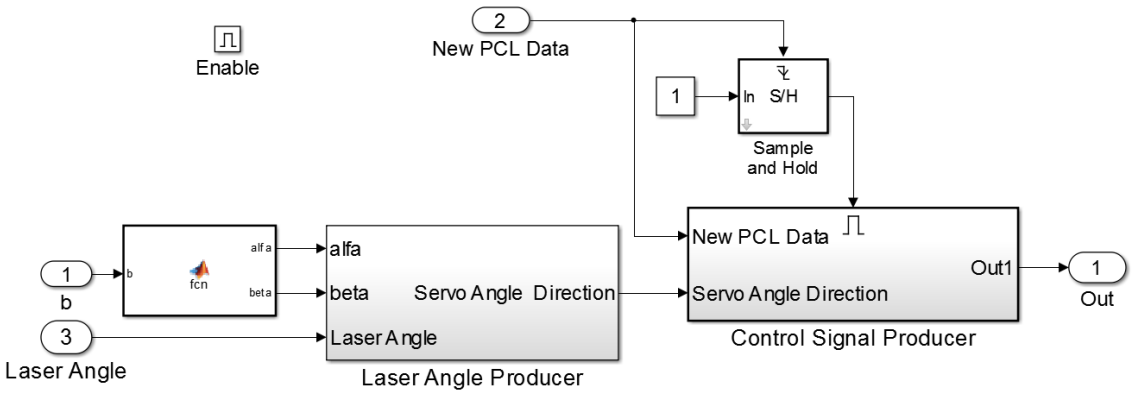


Figure 3.18: Servo controller for closed loop detection

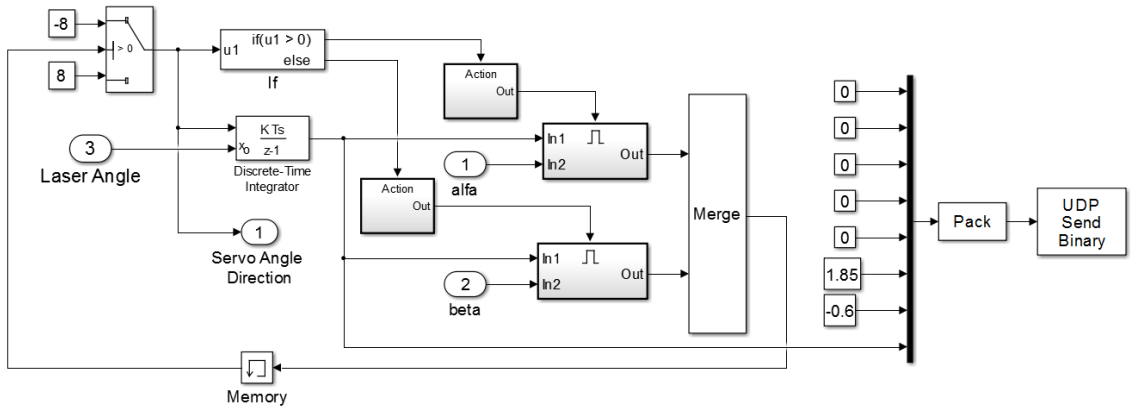


Figure 3.19: Laser angle producer subsystem

current servo angle either to α or β. So the two enabled subsystems inputting α and β are basically comparators detecting whether the integrator output has reached any of these two values. In the case of comparing to α we know that the servo angle is in increasing mode, thus the output will be $+1$ to cause the switch to output -8 and hence switch sweep direction. The same story applies when we are comparing the servo angle to β which means we are in the decreasing mode, so to change the direction the output is set to be -1. Then these two outputs are merged to be updated whenever a change has happened and used to control the switch. The memory block is added to cause a small delay in order to break a conditional infinite loop. Finally using the same package format discussed in section 3.6 this signal is packed and sent to the servo system.

The last part that needs explanation is the subsystem generating the control signal in this phase, and for this we will use the previously described servo direction in addition to the acknowledge signal of PCL new data. The implementation is shown in Figure 3.20.

We have made the servo angle to sweep between our desired values (which are set dynamically), and now the idea is to mask one complete sweep (either increasing or

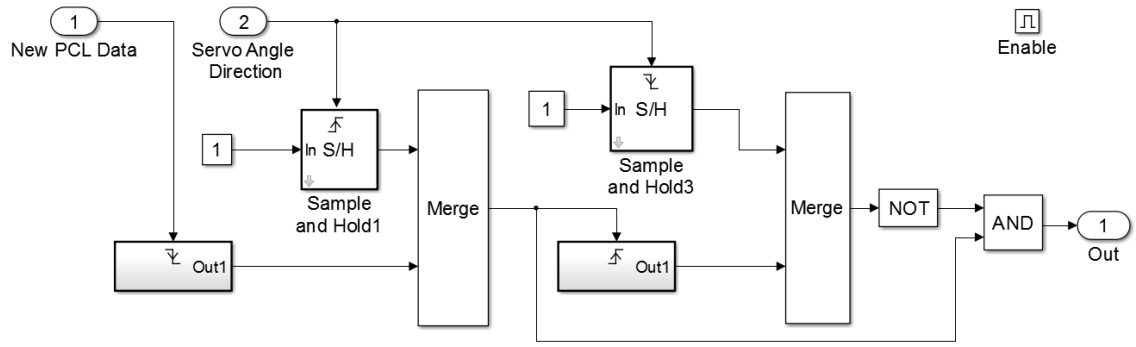


Figure 3.20: Control signal producer subsystem

decreasing) each time after a new data has been sent from PCL. This will ensure that if the registration result has come in the middle of a cycle, we gather data from a complete increasing or decreasing cycle. As this will repeat every time a registration result is received, it also works in accordance with the final while loop in our C++ code, and guarantees iteration.

The logic behind this should be understood from the block diagram and for a deeper understanding Figure 3.21 provides the resulting control signal. The first signal is the direction of the laser servo which switches between +8 and -8 for two sweep directions. The second signal shown is the PCL acknowledge signal having a small pals whenever a result is sent from the PCL machine. Finally it can be seen from the control signal (shown in the third row) that it will wait until a result is sent, then it masks the first complete sweep. Note that the two triggered subsystems force a zero output each time a falling or rising edge is detected.

Figure 3.21: Closed loop control signal

4. TESTS AND RESULTS

4.1 Overview

This chapter describes the evaluation of the implemented method for pallet recognition. Choices taken when designing the algorithm previously described were due to what PCL had to offer. Now to see the results and investigate whether or not our choices were correct, several tests have been conducted and the output results are presented. The chapter is divided into four sections. In section 4.2 the input model is introduced with pictures. Details on parameters used by the implementations are listed in section 4.3 alongside the values we used for those parameters.
Then in section 4.4 the results on the GIMsim simulator are discussed. The most important performance parameters such as recognition time, the alignment fitness score and the position and rotation results are computed. Comparing the average results for ten iterations on each test, we then discuss the results.
Though the time constraint did not allow for complete execution and testing on the machine, but we have gathered some real data from a standard pallet in an open environment to execute some tests. Performed tests on these real data are presented in section 4.5.

4.2 Datasets

It is now time to have a look at data sets used to test our algorithm. There are two point clouds to be fed to the application as input, the previously captured and trained "model" of the pallet front, which we aim to detect and it is saved in the database of the system, and a complete point cloud called the "scene", which is the captured data by the laser scanner in each execution step. Obviously the scene will be different at each time, due to the distance and angle of the laser scanner to the pallet, and the dynamic environment which may change, but the model is the same and it is shown in Figure 4.1.
In order to reveal the position and rotation of the pallet, it will be enough just to see the front view, and no need to give the whole pallet as the model to be detected. This is also due to the fact that we are not usually interested to detect an empty pallet, but rather a loaded one. So when a pallet is loaded the only visible part will

be the front. Note that a specific coordinate frame is assigned to the model, which its origin is exactly at the middle of the pallet front.

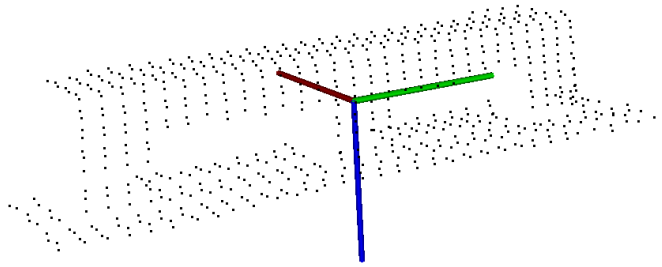


Figure 4.1: Pallet front model

Another rather important time reduction strategy applied here is called the "training" phase. In addition to the model point cloud, we also compute key points, normals and features for the model and pass them to the algorithm as inputs, rather than computing them at the same time with the source cloud. This is a rational and beneficial decision, since the model and all its descriptors remain the same during the whole process, and there is no need to compute them over and over again. We will see the effect of this step later in this chapter. If this process and pipeline is to be used for another model, these data sets should be replaced with the desired ones. Figure 4.2 represents computed key points using uniform sampling technique with a search radius of $4cm$. Figure 4.3 shows normals estimated at key points with a search radius of $0.1m$. The reason to choosing these values is given in section 4.3.2 and 4.3.3.

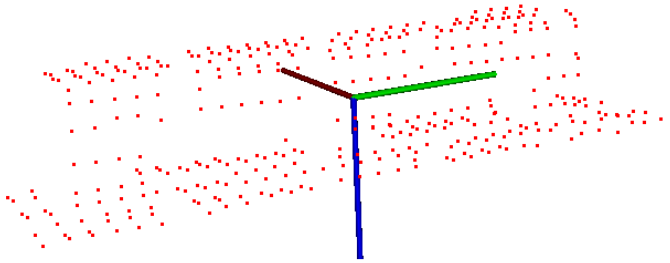


Figure 4.2: Model uniform key points extracted with bin size 0.04m

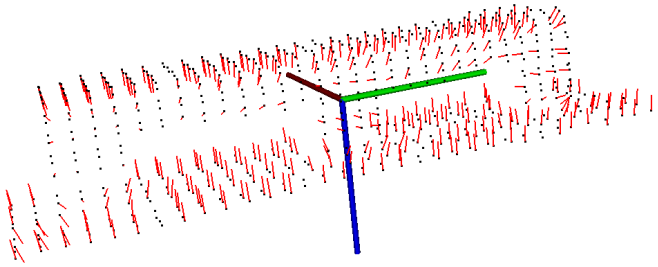


Figure 4.3: Model normals extracted with search radius 0.1m

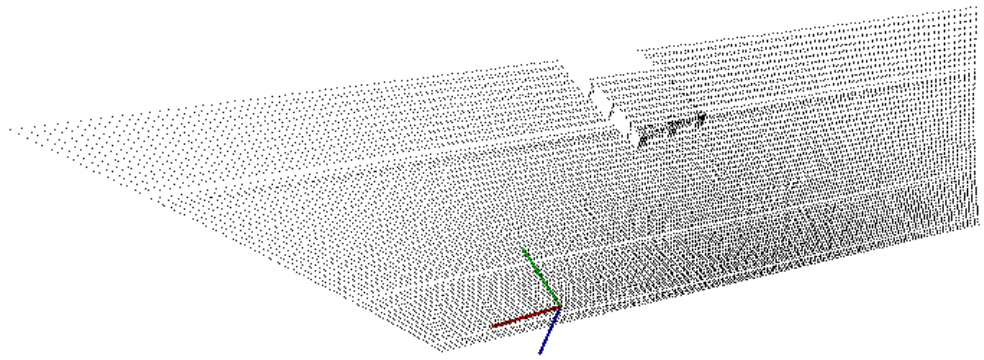
4.3 Determining Parameters

While introducing the algorithm step by step and illustrating the theoretical background for each step, we also presented and mentioned some parameters to be defined by the user for that function. For instance, in the section where we discussed surface normals, it was mentioned that there is no meaning for a normal for a point, but we rather need a small neighbourhood of that point to be considered as a surface, and then to compute the normal. This raised a need for a parameter of the radius of that neighbourhood to be determined. Additionally it was mentioned that the value of these parameters, play a vital role in the recognition algorithm and attention needs to be paid to their setting. Choosing the best parameters can be a little tricky sometimes, and it often depends on the nature of the data.
According to our C++ code, parameters that need setting include the radius of key point selection, radius for normal estimation, radius for feature extraction, minimum distance of sample correspondences, maximum correspondence length, number of iterations for RANSAC algorithm, and finally the euclidean fitness epsilon for ICP algorithm. Through the next subsections we will investigate settings for each of these parameters.

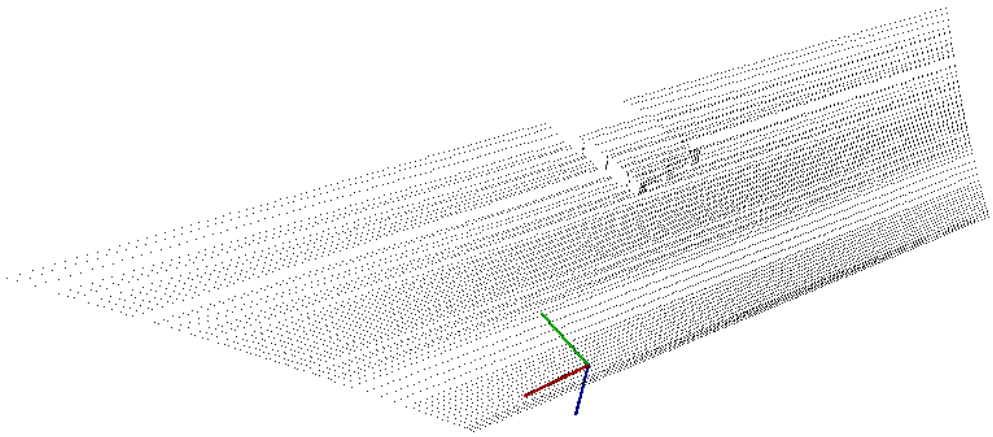
4.3.1 Control Signal Sending Frequency

Recall the section where we described how the scene cloud is saved with the aid of a control signal sent every $5ms$. A sample scene saved with this signal is shown in Figure 4.4(a). To see the effect of this value, the frequency was changed to $15ms$ and the scene was saved accordingly. The result is shown in Figure 4.4(b). Pay attention to the number of rows not saved correctly and missed. Since this data missing is random, it might happen to loose a lot of data on the front of the pallet, and this will destroy the matching result. The results with $5ms$ were satisfying and we decided to set the parameter to this value.

Note here the position of the pallet in the global coordinate frame. The ground is positioned $3m$ above the zero level. This is due to the simulation implementation, and it will not affect any process in the recognition algorithm. But the distance between correspondences in RANSAC should be adjusted with this in mind, as will be shown later. In addition in all the following simulation tests, the left bottom corner of the pallet is manually positioned at $(0, 0, -3)$ in the world coordinate frame, thus we always know what the results should be and calculate the error from that.



(a) Control signal sent every 5ms



(b) Control signal sent every 15ms

Figure 4.4: Comparing a sample scene saved with two different control signal frequencies

4.3.2 Uniform Key points Sampling Size

Extracting key points from a point cloud is the first step through the process and setting the right value for this function is critically important and will affect all the upcoming steps. This radius will affect the number of remaining points from the original cloud after down sampling. Usually there is no strict routine for determining the best value, but it depends a lot on the application, the data set, the minimum speed, and the amount of details needed. Thus one should try some rational values and perform a trade off between the advantages and disadvantages.
This value will also affect the result of the recognition, in addition to the time. If the sampling size is too high, there will be great loss of meaningful data, and might even cause the object shape to be non recognizable. For the model cloud we set this value to $0.04m$ and visualizing them shows that the shape is still recognizable. Keeping this size fixed for the model, the scene sampling size should be found at a point where the accuracy and time lines meet. In other words one should run

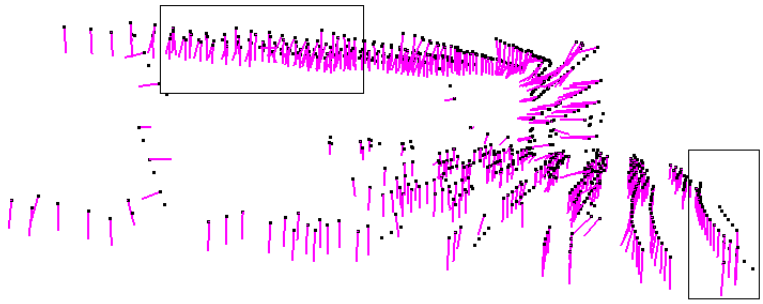
multiple tests and figure out the size which will give both satisfying computation time and accuracy. The value was found to be the same for the scene as well.

4.3.3 Normal Estimation Radius

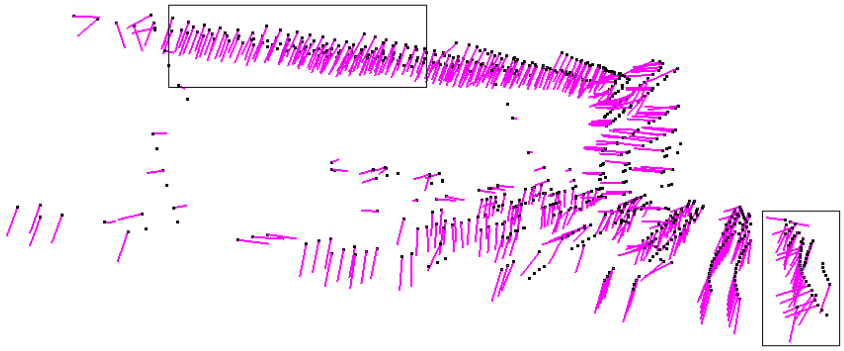
After extracting the key points, it is time to estimate normals for each remaining point. This radius should be chosen to contain enough points in the neighbourhood of a query point. From the key points extraction radius we will roughly know how far points are located from one another, and if the normals radius is less than this distance there will not be any points in the sphere to form a surface and the resulting normal will be NaN. Usually it will be fine to have some NaN in the results, specially at the corners and they can be removed, but the ratio should be really small.
Another option is to use the function `setKSearch()` instead of `setRadiusSearch()` which will use k neighbours no matter the radius, instead of all neighbours in a sphere of radius r. It might be the case that the k neighbours search returns a completely different neighbourhood size than radius search, and obviously different neighbourhoods will result in different normals. The radius search is of particular interest for 3D feature estimations, because it attempts to capture the data on the same surface patch, independent of the number of points. It is advised by the PCL developers to use the radius search with a carefully chosen radius to get more accurate normals [1], and this is exactly what we have done. Additionally it is a wise decision to visualize the normals with provided classes in PCL and perform a visual inspection and choose the best value accordingly.
Fortunately this value is not so sensitive as the concept of a surface remains valid apart from the point data. Changes in the key points extraction size (at least in centimetres order) will not affect the accuracy of normals. Remember as long as we remain on the same surface the normal would be the same, either it is calculated from 5 points or 20 points. We just have to make sure this is more than the sampling size value.
After some trials we chose the radius of $0.1m$ for both model and scene clouds and the test results were also satisfying. Figure 4.5 shows normals for the scene calculated with radius $0.1m$(a) and $0.05m$(b). Pay special attention to the marked areas where the normals are not perpendicular to the surface due to a small radius choice.

4.3.4 Feature Extraction Radius

The feature extraction radius has to be larger than the radius selected to estimate the normals, otherwise the features will be very similar, and thus they will not con-



(a) search radius 0.1m



(b) search radius 0.05m

Figure 4.5: Comparing scene normals estimated with two different radius sizes

tribute to the registration process. Unfortunately features are just float values and can not be visualized to perform any visual inspection. But the procedure to find the best value for this parameter is to start with a value certainly greater than the normals radius, then run the recognition algorithm, and find a value that gives the best result with some test and trial. This value will greatly depend on the algorithm as will be seen later.

4.3.5 RANSAC Parameters

The two parameters `MinSampleDistance` and `MaxCorrespondenceDistance` are related to the RANSAC based recognition algorithm. The first one indicates the minimum distance between two randomly selected correspondences. The value for this parameter should be chosen in accordance with the object size. If the sampled correspondences are too close to each other, the result might not be accurate enough. The second parameter gives a hint about the distance between two corresponding points in the two clouds. This will tell the algorithm that the point we are looking for should not be further than some distance, and should be set to a little bit larger than the average distance between corresponding points in two clouds. In our simu-

lation environment we know that the scene is 3 meters above the model in the global frame and also that the model coordinate frame has 90 degrees rotation around z axis. Having these in mind an estimate of $3.5m$ is a good value for the maximum distance between two corresponding points.
Whereas in the real environment we have to adjust this parameter accordingly. The high level control guides the machine towards the pallet with a good precision based on navigation and odometer data. Thus we will not be performing visual detection from a distance more than $5m$. According to these assumptions the correspondence distance between two corresponding points will not be more than $5m$ in this case.
The number of iterations for this algorithm depends some how on the rotation of the two clouds with respect to each other. If the two clouds have great rotations, more iterations are needed. This value should be set to the minimum value with which the algorithm gives a satisfying result. In the simulation environment the two coordinate frames for model and scene have 90 degrees of rotation, while in the real system the point cloud comes in the same coordinate frame as the model. Thus the number of iterations would be much less in this case. There is no rule to set this parameter, but one should try different values and inspect the results and set the value accordingly.

4.3.6 ICP Euclidean Fitness Epsilon

The intention of ICP is to keep minimizing the euclidean distance between all the points (or a subset of them) selected in the point clouds and this is done iteratively. One termination condition is to set the maximum allowed Euclidean error between two consecutive steps in the ICP loop, before the algorithm is considered to have converged and stops. The error is estimated as the sum of the differences between correspondences in an Euclidean sense, divided by the number of them.
With this parameter we indicate that if the sum of Euclidean squared errors is smaller than this value, we want to terminate the iteration. We have set this value in accordance with our desired resolution to $0.01m$.

4.4 Simulation Results

Time is one of the most important performance metrics in any project which becomes even more important in real time applications involving autonomy. In several different places within this document it was mentioned that some strategies were applied in order to decrease the execution time and to increase the quality of our proposed algorithm. It is now time to investigate the validation of those claims. In

this section we will have a closer look at time consumption of different steps within our algorithm, in addition to performing some tests with different parameters to investigate the effect of each on the process time.
The system used for all tests is a lenovo X200 laptop with 4GB RAM, Intel core 2 duo CPU 2.26 GHz, operating on a 32 bit Linux ubuntu 13.04. All tests have been performed on the same system, and also at the same time to decrease the chance of any external factor affection. More over when investigating the effect of one parameter, all the other parameters were kept the same, except if otherwise mentioned. To perform the tests, the machine was driven manually to a position near to the pallet and facing its front. Some scenes were taken while the machine was moving and some were taken with the machine standing still. All these conditions are described in the following pages.
It was mentioned before that the uniform key points sampling size has a great effect on the final result and the recognition time as well. Thus several tests have been conducted to investigate the effect of this parameter. First we have tried our algorithm with a sampling size of 0.001 ($1mm$) which is a small size compared to the pallet size and data resolution. The machine was placed at the nearest possible distance to the pallet and it was not moving during the data acquisition, then the number of original scene cloud, number of selected key points, total registration time, time consumption of each step, ICP fitness score and the rotation and position of the final matrices were calculated. Note that since we are using a simulated model and the environment is ideal, there was no rotations around x and y axis, and the results also indicated this correctly, so we do not repeat these and only record the rotation angle around z axis. Table 4.1 shows the results. In the next test the pallet and machine positions were kept the same but we changed the sampling size to 0.05 ($5cm$) and the results are represented in Table 4.2.

Table 4.1: Near Distance, With Training, Sample Size 0.001, Machine Stationary

	ICP Score	Total Points	Key Points	Recognition Time(ms)	Angle (degree)	Pos. X	Pos. Y	Pos. Z
1	0.000685	15424	3717	1389.70	93.43	-0.553	-0.133	-3.059
2	0.000410	15080	3721	1230.43	92.51	-0.537	-0.169	-3.059
3	0.000650	13469	3337	1129.31	91.89	-0.561	-0.144	-3.062
4	0.000594	15573	3665	1327.84	94.50	-0.558	-0.144	-3.059
5	0.000326	14108	3420	1162.82	90.05	-0.569	-0.151	-3.058
6	0.000595	14002	3610	1222.86	93.88	-0.582	-0.143	-3.059
7	0.000521	14971	3571	1234.68	95.46	-0.641	-0.144	-3.058
8	0.000529	13906	3512	1176.73	88.98	-0.625	-0.146	-3.059
9	0.000321	15423	3802	1328.12	89.07	-0.620	-0.159	-3.058
10	0.000345	14472	3457	1205.48	96.84	-0.627	-0.143	-3.058
Av	0.000497	14643	3581	1240.79	92.66	-0.587	-0.147	-3.058

Table 4.2: Near Distance, With Training, Sample Size 0.05, Machine Stationary

	ICP Score	Total Points	Key Points	Recognition Time(ms)	Angle (degree)	Pos. X	Pos. Y	Pos. Z
1	0.000566	12794	1701	592.357	88.21	-0.631	-0.095	-3.062
2	0.000749	14670	1885	634.040	87.00	-0.564	-0.091	-3.060
3	0.000512	13925	1825	600.673	86.54	-0.598	-0.085	-3.061
4	0.002214	14027	1778	620.687	55.87	-0.663	-0.077	-3.058
5	0.000512	14024	1741	607.865	91.55	-0.594	-0.069	-3.062
6	0.000664	12996	1762	618.269	89.77	-0.656	-0.064	-3.061
7	0.000736	13728	1901	673.732	86.21	-0.567	-0.063	-3.060
8	0.000520	13554	1811	627.370	94.23	-0.602	-0.045	-3.061
9	0.000432	13958	1894	658.331	97.88	-0.632	-0.037	-3.061
10	0.000584	14331	1862	634.239	88.95	-0.594	-0.041	-3.060
Av	0.000749	13801	1816	626.756	86.62	-0.610	-0.066	-3.060

Considering average values from these tests, with $1mm$ sampling size the total recognition time was 1240.79 and with $5cm$ it was 626.756. At the same time the average of ICP fitness score was 0.000497 in the first test and 0.000749 in the second test. Any ICP fitness score with an order of 10^{-4} indicates a sharp and good result. Comparing ICP fitness scores from their average might not be a good index to consider, but it is better to use the number of bad recognitions out of ten trials. With this in mind we can see that we had only one bad result out of ten in the second test, while this ratio was zero in the first test at the cost of time. We do not repeat the data, but we determined that this ratio increased with increasing the sampling size. This gives a straight result of an inverse relationship between the quality and quantity in the algorithm, and raises the concept of trade-off. We have accepted a 10% of failure and aimed for a recognition time under $1s$. Of course this can be modified by needs. Also a sample scene is presented in Figure 4.6 showing the pallet near position, the original model and the red area is where the algorithm has placed the model as the result of recognition.

The next test is conducted to investigate the difference between detecting a near and a far object. It is a natural behaviour of laser range scanners to produce more dense clouds near the beam origin, and more sparse ones at further distances. Previous figures clearly show this common effect which is due to some divergence effect. There are two divergences, one is the divergence of two adjacent beams, because of ray nature of the beams originating from single point which will diverge along the distance. The other is the divergence of one beam, since real laser is not perfect and beam radius will increase as it travels farther.

As the density of the data has great effect on the number of points and thus on the recognition accuracy and time, we conducted a test at the furthermost distance

between the pallet and the machine where the pallet was still visible to the sensor. While our previous tests have been in the nearest distance we will see the difference as more magnified. With the same sampling size of $5cm$ the machine is situated at a further distance and the results of ten recognition iterations are given in Table 4.3 and a sample cloud is shown in Figure 4.7.

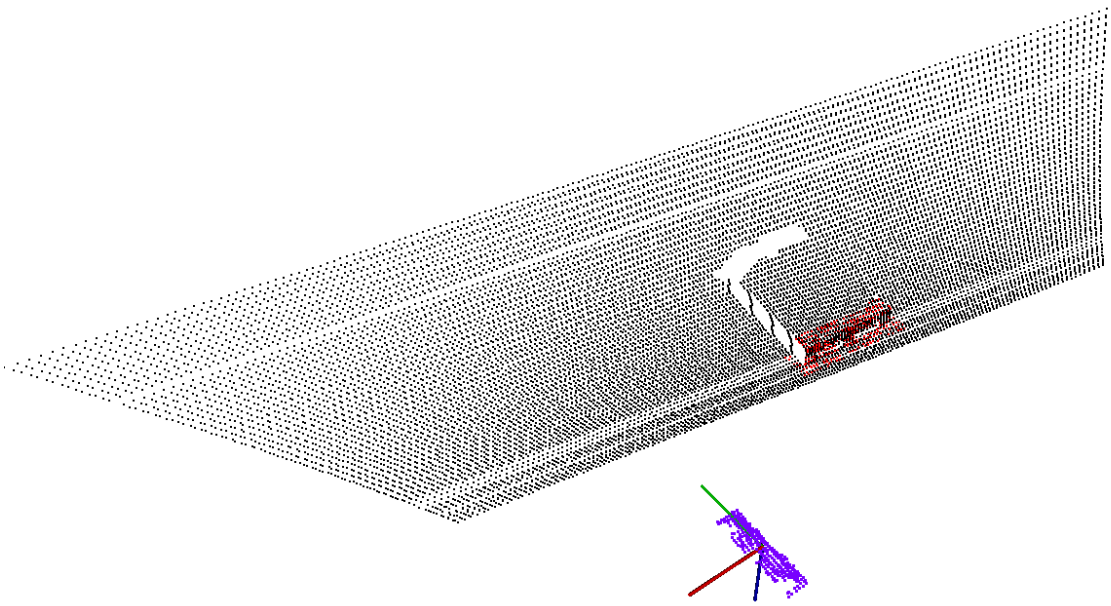


Figure 4.6: Sample saved scene cloud showing the pallet near position in relation to the original model (the red area shows the final match)

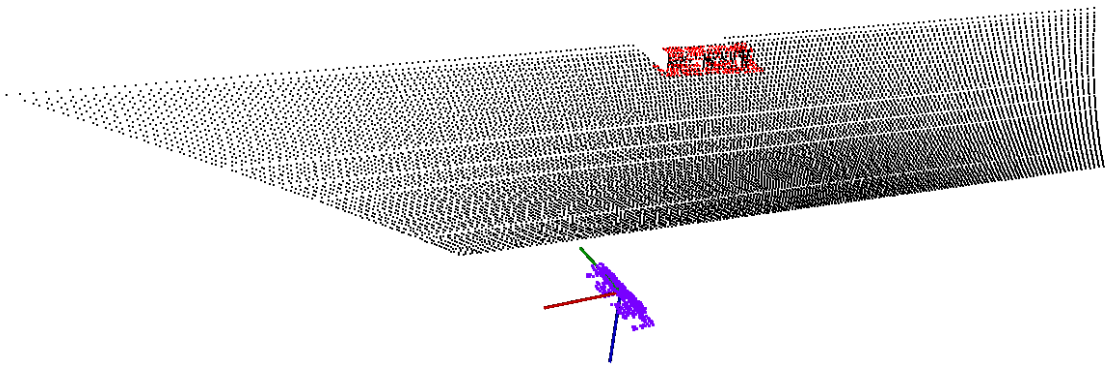


Figure 4.7: Sample saved scene cloud showing the pallet near position in relation to the original model (the red area shows the final match)

Table 4.3: Far Distance, With Training, Sample Size 0.05, Machine Stationary

	ICP Score	Total Points	Key Points	Recognition Time(ms)	Angle (degree)	Pos. X	Pos. Y	Pos. Z
1	0.000753	3541	569	465.612	89.44	-0.636	-0.167	-3.038
2	0.000848	3697	611	464.319	88.12	-0.667	-0.167	-3.041
3	0.000836	3577	590	480.708	-79.92	-0.600	-0.176	-3.038
4	0.001764	3259	592	479.900	122.21	-0.558	-0.119	-3.037
5	0.000723	3763	548	446.417	88.54	-0.648	-0.171	-3.039
6	0.001848	2841	598	491.745	-145.55	-0.783	-0.184	-3.037
7	0.000777	3692	620	471.932	95.22	-0.675	-0.170	-3.041
8	0.000830	3109	566	466.298	188.87	-0.593	-0.176	-3.039
9	0.001090	3675	577	460.859	184.32	-0.580	-0.177	-3.039
10	0.000940	3253	537	449.900	44.05	-0.602	-0.178	-3.039
Av	0.001041	3441	581	467.769	67.53	-0.634	-0.168	-3.038

The very first great difference revealed from the results is the amount of points gathered in two different distances despite the same sampling size. When near, the average total scene points was 13801 and the average of key points left after uniform sampling was 1816. While when far, these values dropped to 3441 and 581 receptively. This clearly shows that we will have more meaningful data (points from the object) when the object is closer. Thus the density and resolution of the cloud should be considered seriously before applying any uniform sampling, to adjust the sampling size accordingly. Above results show that with the same sampling size of $5cm$ the unreliability increases to 20% when the pallet is further. One possible adaptive way is to use a greater sampling size for the first scan, then reduce it as the machine approaches the pallet.

One more rather important test worth reporting was performed with the machine moving during the data acquisition. We drove the machine manually with different speeds and accelerations and during different times, completely randomly. Moreover the movements were both in forwards and backwards directions. Table 4.4 holds the results of ten recognition iterations. Three sample scenes are also shown in Figure 4.8. Note from the table that the amount of total points and key points has no specific order, and this was completely expected. Also the pictures show randomly distributed densities due to random movements as well. Though this will affect the recognition time, but the results showed great accuracy during the tests except from the the times when the pallet was at a far distance and there was not enough points acquired.

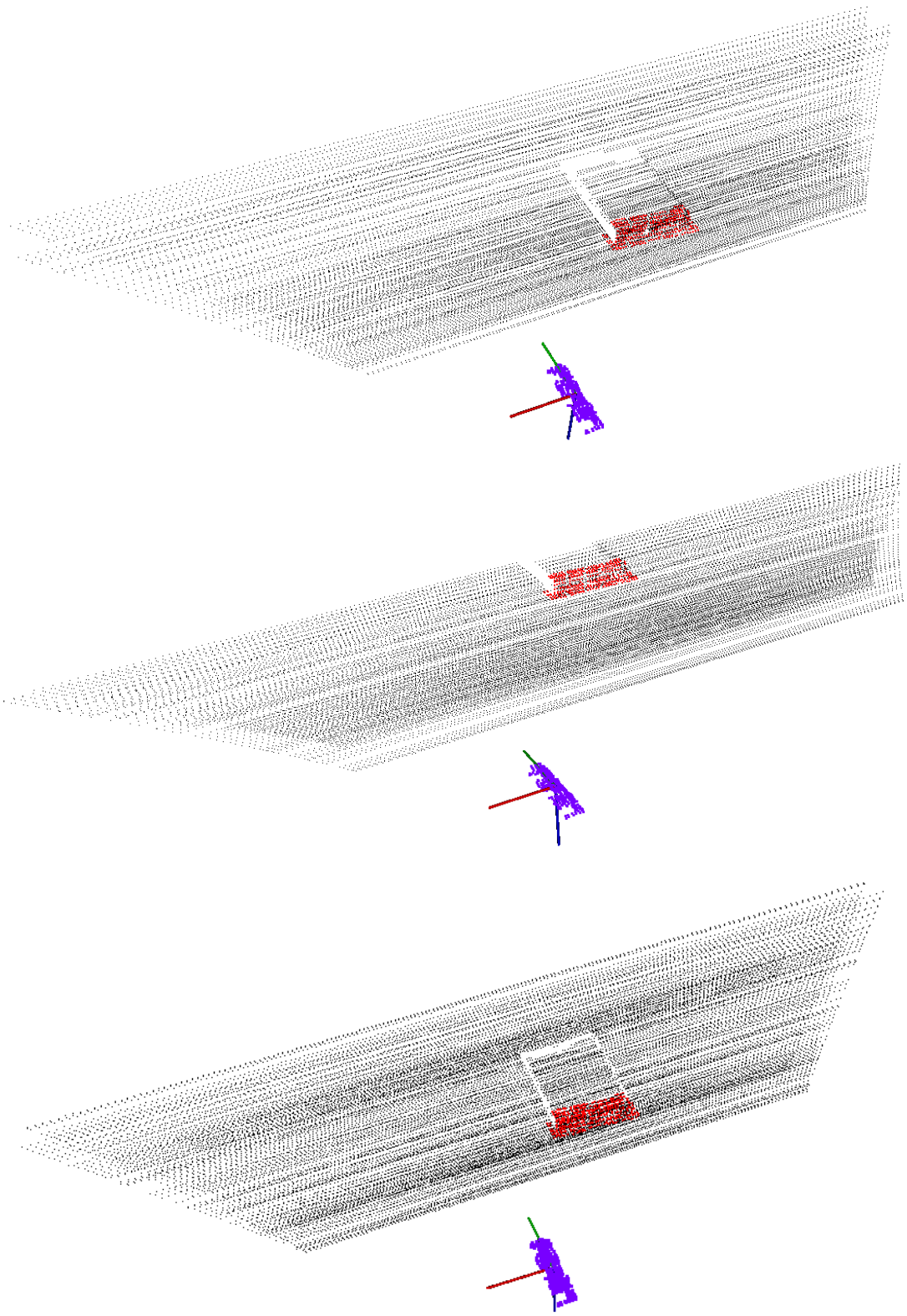


Figure 4.8: Sample saved scene clouds while the machine was moving (the red areas show the matching result)

Table 4.4: With Training, Sample Size 0.05, Machine Moving

	ICP Score	Total Points	Key Points	Recognition Time(ms)	Angle (degree)	Pos. X	Pos. Y	Pos. Z
1	0.000462	13879	2108	805.286	92.04	-0.608	-0.129	-3.055
2	0.000562	5236	1162	534.021	89.88	-0.579	-0.159	-3.048
3	0.006800	4837	1113	557.591	178.04	-0.571	-0.148	-3.045
4	0.000992	9002	1819	640.853	145.52	-0.627	-0.206	-3.050
5	0.002541	9161	2326	826.957	-98.48	-0.644	-0.162	-3.062
6	0.000393	13034	2193	724.094	94.66	-0.579	-0.154	-3.053
7	0.000383	13054	2183	724.084	91.04	-0.589	-0.184	-3.083
8	0.000671	6243	1421	599.831	85.32	-0.561	-0.103	-3.047
9	0.000573	12093	2063	750.838	89.65	-0.519	-0.141	-3.056
10	0.001238	9220	1871	703.200	-154.22	-0.463	-0.178	-3.047
Av	0.001462	9576	1826	686.675	61.34	-0.574	-0.156	-3.054

One of the first recommended strategies was to previously compute key points, normals, and features for the model and feed them as input to the algorithm, rather than computing them at the run time. Although this decision might seem quite obvious at first, but loading data will be time consuming as well. All previous tests were done with training. To put this under test, an experiment was performed with all preprocessing steps for the model at the same time with processing the scene.

Table 4.5: Without Training, Sample Size 0.05, Machine Stationary

	ICP Score	Total Points	Key Points	Recognition Time(ms)
1	0.000733	14210	1854	670.453
2	0.000973	14708	1899	691.501
3	0.000500	13810	1717	682.251
4	0.000508	14951	1983	764.664
5	0.000729	15091	1889	684.083
6	0.001299	14901	1935	702.992
7	0.000477	13757	1934	787.265
8	0.000439	14179	1919	690.751
9	0.000413	15295	1924	699.801
10	0.000449	14015	1860	675.622
Average	0.000652	14492	1891	704.938

Comparing the average recognition times from Table 4.2 and 4.5 it is clear that we have saved $704.938 - 626.756 = 78.182ms$ on training. This could have been anticipated since the model cloud does not contain so many points and it is rather a small cloud. Thus the saved amount of time might not be significant. This time saving however will increase by the size of model cloud.

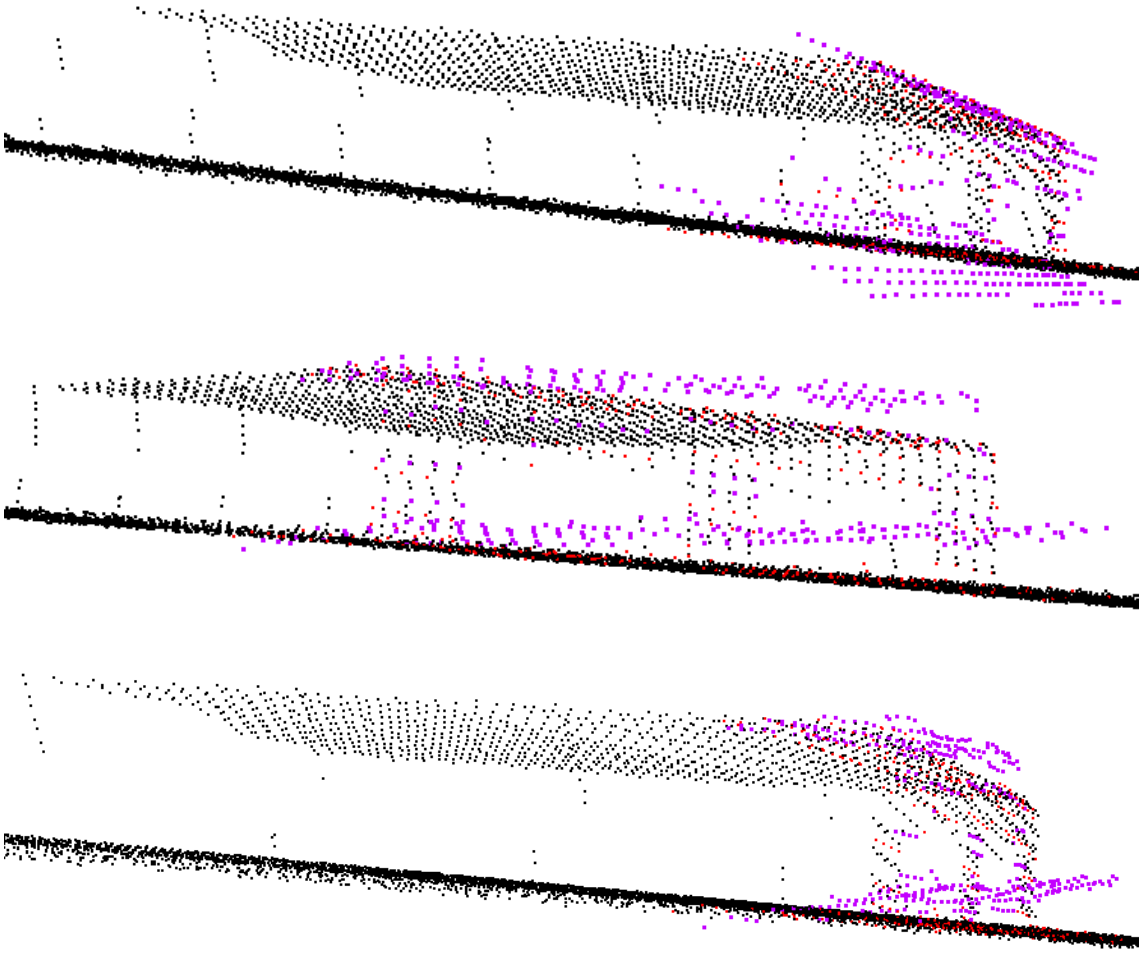


Figure 4.9: A sample recognition result from different view angles showing the effect of iterative closest points alignment, the purple cloud shows the output of random sample consensus algorithm, and the red cloud shows the final result after fine alignment with iterative closest points

As mentioned the final alignment is performed to give a sharp more accurate refinement to the initial alignment result in a faster way. To see the effect of this process a sample output is shown in Figure 4.9 from different angles and closer views. The black cloud is the original scene, the purple cloud is the output of RANSAC, and the red cloud represents the output of the ICP algorithm. It is obvious that ICP fine alignment produces very great and accurate results and without a doubt it will be very beneficial.

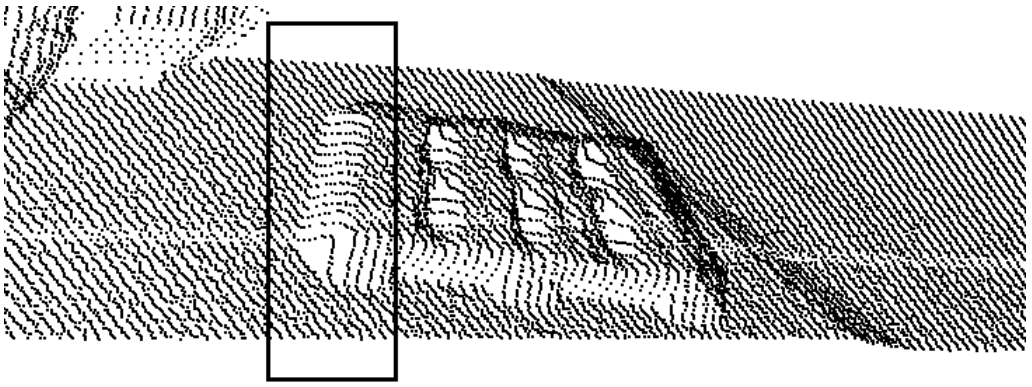
4.5 Real Data Results

Our document would have not been complete without performing any tests on real data. The simulation is a helpful method to test our algorithm, but the real data

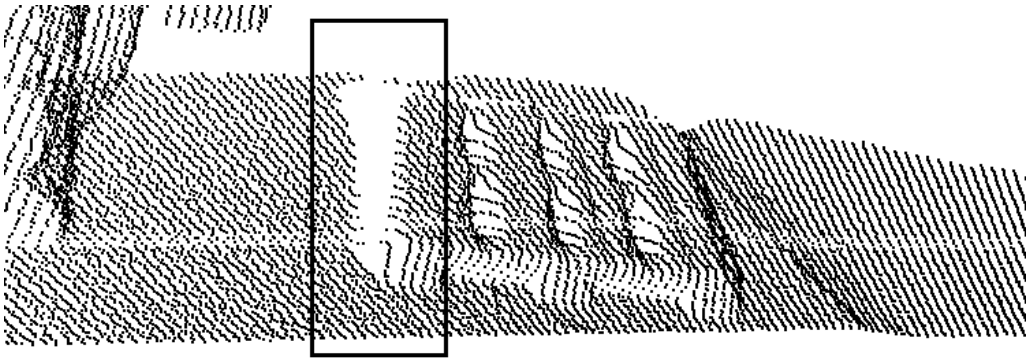
do not always look as in simulated environment. There are many factors affecting the data, laser noise, weather conditions, object edges, and etcetera. Figure shows a close look at a point cloud from a real pallet in outdoor environment and it shows that how previously mentioned factors affect the cloud. These complicate the estimation of local point cloud characteristics such as surface normals or curvature changes, leading to erroneous values, which in turn might cause point cloud registration failures.
Thus applying these data to our algorithm did not give any fine results at all. This data sets suffer greatly from the noise and the effect of sharp edges. PCL provides a very useful filter called "Statistical Outliers Removal" especially for this purpose. Some of these irregularities can be solved by performing a statistical analysis on each point's neighbourhood, and trimming those which do not meet a certain criteria. The sparse outlier removal is based on the computation of the distribution of point to neighbours distances in the input dataset. For each point, it computes the mean distance from it to all its neighbours. By assuming that the resulted distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standard deviation can be considered as outliers and trimmed from the dataset.
The following Figures 4.10 and 4.11 show the effect of the sparse outlier analysis and removal. The original dataset is shown on the top, while the resultant one on the bottom. Note how the points at the side of the pallet are removed and how the bottom pictures look smoother. Also the number of points in the original cloud was 46646, while after removing the outliers the remaining number of points were 33573. Figure 4.12 shows finally what we will be feeding the algorithm as the scene. The outliers are removed and the scene is down sampled as well in this figure. In comparison to the original noisy scene in 4.11(b), see how the front of the pallet is more recognizable and smooth.

To adapt the algorithm, the following piece of code was added to our main program. As the usual routine first an object of this class of filters is created here called `sor`. The function `setInputCloud()` gives the scene cloud as the input to the filter. The two parameters of mean number of neighbouring points and the stadard deviation threshold are set using the functions `setMeanK()` and `setStddevMulThresh()` respectively. Then a call to the `filter()` function will perform the main filtering process. Just adding this filter to the algorithm and changing the crop filter parameters the results were improved greatly.

```
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
sor.setInputCloud (scene);
sor.setMeanK (50);
sor.setStddevMulThresh (1);
sor.filter (*scene);
```

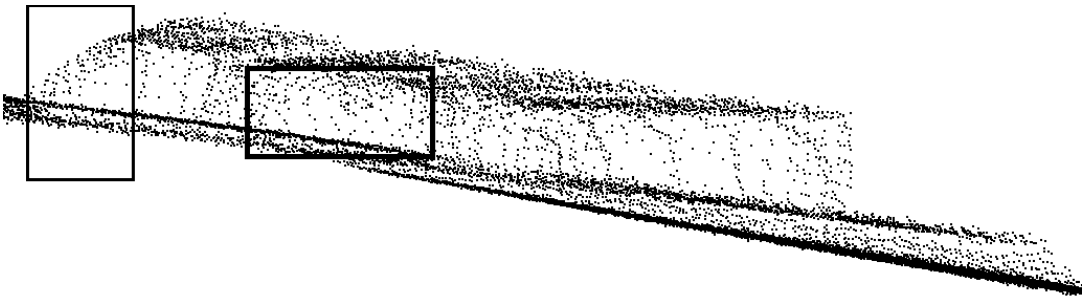


(a) Before

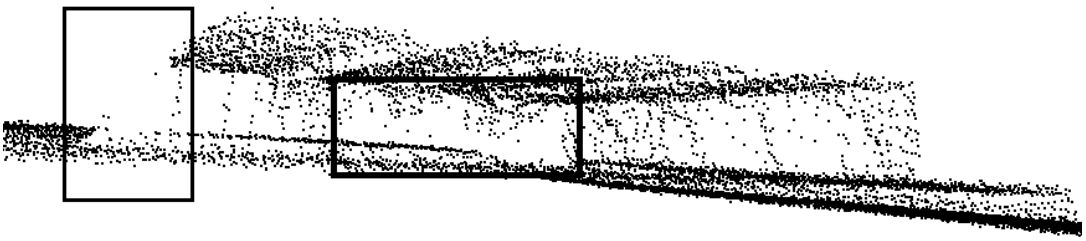


(b) After

Figure 4.10: The effect of statistical outliers removal filter, the noisy data from the side edge is removed



(a) Before



(b) After

Figure 4.11: Applying statistical outliers removal filter, most of the data associated with noise is removed from the side edge and the empty space of the front

Figure 4.12: A noisy scene, filtered with statistical outlier remover and downsampled

For this data set we are interested in the effect of different algorithms and performance of different descriptors in each of them. Thus four sets of tests were performed comparing FPFH and SHOT descriptors in two previously mentioned main algorithms. The first one is finding and rejecting correspondences while the second algorithm uses RANSAC based method.
Five tests have been conducted on five different scenes and the results are presented in Tables 4.6 , 4.7, 4.8, and 4.9 with the calculation of the time consumed by each step and the position and rotation outputs. To capture the scenes the machine was driven manually to the front of the panel in five different positions. In all tests the downsampled model containing 618 points was given as model input and the radius for the scene down sampling was $0.04m$, normal estimation radius was $0.1m$, epsilon distance for rejecting correspondences was $0.03m$, minimum sample distance and maximum correspondence distance for RANSAC based method were $0.02m$ and $4.5m$ respectively, and euclidean epsilon for ICP was $0.01m$ fixed.
Although in this stage we had no ground truth about the position of the pallet like we did in simulation, but PCL provides a transformation validation function using the method described in euclidean validation function section to produce a float representing the level of confidence of the resultant transformation. This class accepts the model and the scene clouds in addition to the transformation matrix, and will calculate how this transformation matrix will get the model close to the scene. Then it returns a float value with which we can perform comparison between different transformation matrices and investigate the correctness of our results. These fitness scores are also given in each table.
For illustrative purposes we have also performed a visual inspection and the results showed a very good estimation as represented in Figure 4.13. The figure also shows the remaining correspondences in green lines, based on which the best transformation was estimated.

Table 4.6: Algorithm 1, FPFH Descriptors, Scene Sample Size $0.04m$, Model Points 618

	Fitness Score	Total Points	Key Points	Total Time(s)	Desc. Time	Corr.Est. Time	Corr.Rej. Time	Desc.Ext. Radius	Number of Iterations
1	0.000533	46646	2763	2.46	0.63	0.04	1.67	0.28	1800
2	0.000438	46646	2776	0.64	0.14	0.02	0.26	0.11	300
3	0.000680	46646	2748	1.30	0.58	0.04	0.43	0.27	500
4	0.000587	46646	3340	1.13	0.46	0.02	0.35	0.22	400
5	0.000428	46646	2396	0.75	0.41	0.01	0.08	0.24	100

Rotation Matrix	Roll(x)	Pitch(y)	Yaw(z)	Translation Vector
$\begin{bmatrix} 0.76 & -0.35 & -0.53 \\ 0.18 & 0.92 & -0.34 \\ 0.61 & 0.17 & 0.77 \end{bmatrix}$	12.45	13.32	-37.99	$\begin{bmatrix} 2.00 \\ -0.34 \\ -0.21 \end{bmatrix}$
$\begin{bmatrix} 0.99 & 0.16 & -0.00 \\ -0.16 & 0.99 & 0.03 \\ 0.01 & -0.03 & 1.00 \end{bmatrix}$	-1.71	-9.18	-0.57	$\begin{bmatrix} 1.98 \\ -0.30 \\ -0.26 \end{bmatrix}$
$\begin{bmatrix} 0.94 & 0.27 & -0.19 \\ -0.20 & 0.93 & 0.31 \\ 0.26 & -0.26 & 0.93 \end{bmatrix}$	-15.62	-12.01	-15.14	$\begin{bmatrix} 2.22 \\ 0.12 \\ -0.19 \end{bmatrix}$
$\begin{bmatrix} 0.88 & -0.27 & 0.39 \\ 0.25 & 0.96 & 0.09 \\ -0.40 & 0.01 & 0.92 \end{bmatrix}$	0.62	15.86	23.62	$\begin{bmatrix} 3.07 \\ -0.13 \\ -0.38 \end{bmatrix}$
$\begin{bmatrix} 0.99 & 0.08 & -0.11 \\ -0.09 & 0.99 & -0.07 \\ 0.11 & 0.08 & 0.99 \end{bmatrix}$	4.62	-5.19	-6.31	$\begin{bmatrix} 2.87 \\ -0.27 \\ -0.33 \end{bmatrix}$

Table 4.7: Algorithm 2, FPFH Descriptors, Scene Sample Size $0.04m$, Model Points 618

	Fitness Score	Total Points	Key Points	Total Time(s)	Desc. Time	Alg. Time	Desc. Radius	Number of Iterations
1	0.000482	46646	2763	0.81	0.15	0.22	0.11	50
2	0.000607	46646	2776	0.82	0.14	0.20	0.11	50
3	0.000637	46646	2748	0.85	0.14	0.22	0.11	50
4	0.000757	46646	3340	0.99	0.19	0.20	0.12	50
5	0.000529	46646	2396	0.80	0.14	0.18	0.12	50

Rotation Matrix	Roll(x)	Pitch(y)	Yaw(z)	Translation Vector
$\begin{bmatrix} 0.97 & -0.12 & 0.20 \\ 0.11 & 0.99 & 0.08 \\ -0.20 & -0.06 & 0.98 \end{bmatrix}$	-3.50	6.47	11.58	$\begin{bmatrix} 1.96 \\ -0.36 \\ -0.26 \end{bmatrix}$
$\begin{bmatrix} 0.99 & 0.01 & -0.12 \\ -0.01 & 1.00 & -0.03 \\ 0.12 & 0.03 & 0.99 \end{bmatrix}$	1.73	-0.58	-6.91	$\begin{bmatrix} 2.02 \\ -0.12 \\ -0.26 \end{bmatrix}$
$\begin{bmatrix} 0.81 & 0.56 & -0.13 \\ -0.51 & 0.80 & 0.29 \\ 0.28 & -0.16 & 0.95 \end{bmatrix}$	-9.56	-32.19	-16.30	$\begin{bmatrix} 2.14 \\ 0.14 \\ -0.20 \end{bmatrix}$
$\begin{bmatrix} 0.90 & -0.32 & -0.29 \\ 0.32 & 0.95 & -0.04 \\ 0.28 & -0.05 & 0.96 \end{bmatrix}$	-2.98	19.57	-16.34	$\begin{bmatrix} 3.04 \\ -0.08 \\ -0.34 \end{bmatrix}$
$\begin{bmatrix} 0.98 & 0.04 & 0.21 \\ -0.03 & 1.00 & -0.04 \\ -0.21 & 0.04 & 0.98 \end{bmatrix}$	2.34	-1.75	12.09	$\begin{bmatrix} 2.89 \\ -0.15 \\ -0.37 \end{bmatrix}$

Table 4.8: Algorithm 1, SHOT Descriptors, Scene Sample Size $0.04m$, Model Points 618

	Fitness Score	Total Points	Key Points	Total Time(s)	Desc. Time	Cor.Est. Time	Cor.Rej. Time	Desc. Radius	Number of Iterations
1	0.000388	46646	2763	1.34	0.15	0.93	0.04	0.11	50
2	0.000366	46646	2776	1.41	0.15	0.97	0.46	0.11	50
3	0.000426	46646	2748	1.49	0.15	0.92	0.17	0.11	200
4	0.000602	46646	3340	1.71	0.17	1.14	0.09	0.11	100
5	0.000657	46646	2396	1.34	0.13	0.80	0.17	0.11	200

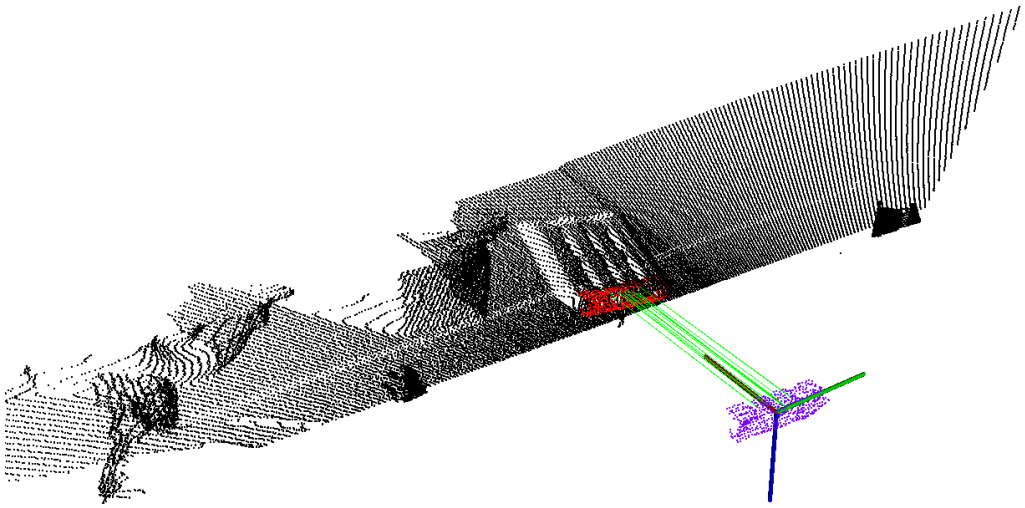
Rotation Matrix	Roll(x)	Pitch(y)	Yaw(z)	Translation Vector
$\begin{bmatrix} 0.98 & -0.20 & -0.02 \\ 0.20 & 0.98 & -0.04 \\ 0.02 & 0.04 & 1.00 \end{bmatrix}$	2.29	11.53	-1.14	$\begin{bmatrix} 1.93 \\ -0.24 \\ -0.24 \end{bmatrix}$
$\begin{bmatrix} 0.99 & 0.06 & -0.11 \\ -0.08 & 0.99 & -0.11 \\ 0.11 & 0.11 & 0.99 \end{bmatrix}$	6.34	-4.62	6.32	$\begin{bmatrix} 1.99 \\ -0.28 \\ -0.27 \end{bmatrix}$
$\begin{bmatrix} 0.67 & 0.54 & -0.47 \\ -0.39 & 0.83 & 0.39 \\ 0.60 & -0.09 & 0.79 \end{bmatrix}$	-6.50	-30.20	-37.74	$\begin{bmatrix} 2.20 \\ 0.22 \\ -0.25 \end{bmatrix}$
$\begin{bmatrix} 0.87 & -0.48 & -0.01 \\ 0.46 & 0.84 & 0.29 \\ 0.15 & 0.25 & 0.95 \end{bmatrix}$	14.74	27.86	-8.67	$\begin{bmatrix} 3.09 \\ -0.14 \\ -0.37 \end{bmatrix}$
$\begin{bmatrix} 0.43 & 0.09 & 0.89 \\ 0.11 & 0.98 & -0.16 \\ -0.89 & 0.17 & 0.42 \end{bmatrix}$	22.04	14.35	63.49	$\begin{bmatrix} 2.95 \\ -0.30 \\ -0.33 \end{bmatrix}$

Table 4.9: Algorithm 2, SHOT Descriptors, Scene Sample Size $0.04m$, Model Points 618

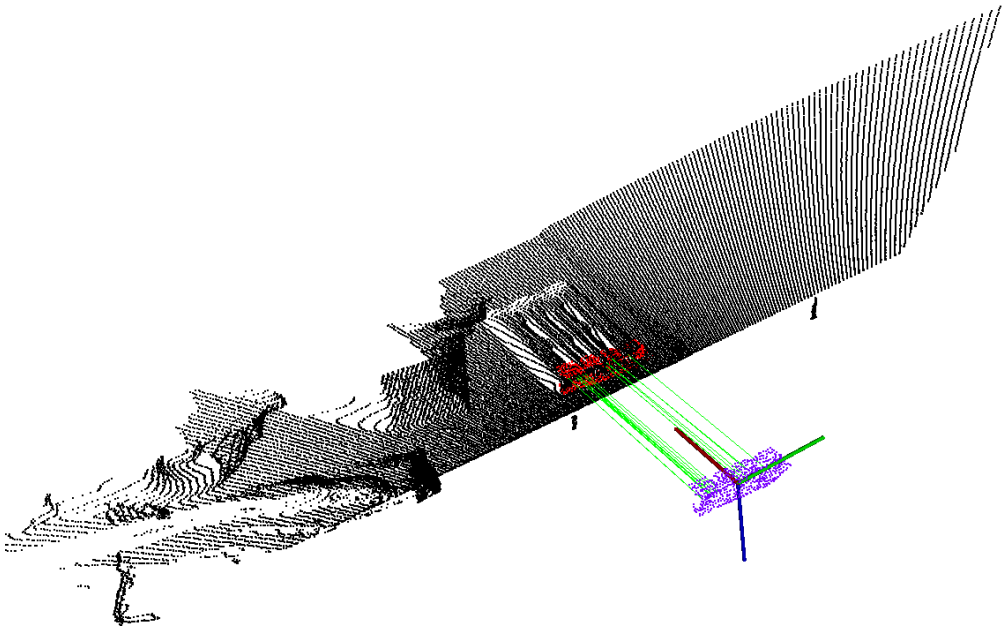
	Fitness Score	Total Points	Key Points	Total Time(s)	Desc. Time	Alg. Time	Desc. Radius	Number of Iterations
1	0.000371	46646	2763	8.26	0.17	7.53	0.14	300
2	0.000337	46646	2776	3.24	0.13	2.51	0.11	100
3	0.000440	46646	2748	7.10	0.16	6.35	0.13	250
4	0.000727	46646	3340	9.73	0.18	8.80	0.13	300
5	0.000891	46646	2396	5.12	0.12	4.43	0.12	200

Rotation Matrix	Roll(x)	Pitch(y)	Yaw(z)	Translation Vector
$\begin{bmatrix} 0.85 & -0.27 & 0.45 \\ 0.22 & 0.96 & 0.16 \\ -0.48 & -0.03 & 0.88 \end{bmatrix}$	-1.95	14.51	28.66	$\begin{bmatrix} 1.98 \\ -0.32 \\ -0.26 \end{bmatrix}$
$\begin{bmatrix} 0.87 & 0.06 & -0.49 \\ -0.08 & 0.99 & -0.03 \\ 0.48 & 0.07 & 0.87 \end{bmatrix}$	4.60	-5.26	-28.78	$\begin{bmatrix} 2.01 \\ -0.24 \\ -0.28 \end{bmatrix}$
$\begin{bmatrix} 0.84 & 0.54 & -0.04 \\ -0.50 & 0.80 & 0.32 \\ 0.20 & -0.25 & 0.94 \end{bmatrix}$	-14.89	-30.76	-11.56	$\begin{bmatrix} 2.22 \\ 0.24 \\ -0.24 \end{bmatrix}$
$\begin{bmatrix} 0.77 & -0.38 & -0.50 \\ 0.30 & 0.92 & -0.24 \\ 0.55 & 0.04 & 0.83 \end{bmatrix}$	2.75	21.29	-33.64	$\begin{bmatrix} 3.05 \\ -0.27 \\ -0.36 \end{bmatrix}$
$\begin{bmatrix} 0.98 & 0.19 & -0.08 \\ -0.19 & 0.99 & 0.04 \\ 0.08 & -0.02 & 0.99 \end{bmatrix}$	-1.16	-10.97	-4.58	$\begin{bmatrix} 2.83 \\ -0.35 \\ -0.35 \end{bmatrix}$

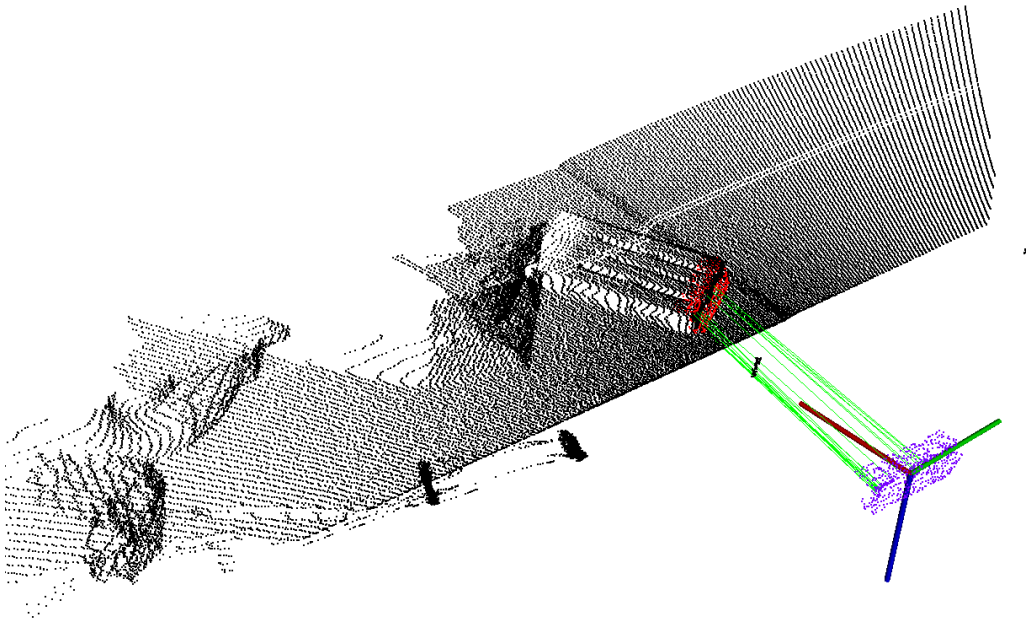
In real data processing we are not only interested in the accuracy of different scene results, but the recognition of an object should be robust regardless of the view angle. Especially in real time applications there is no chance to change the parameters. Comparing the four above tables we can see that only one method with one descriptor type will produce results with the same feature extraction radius. In other three tables regardless of the accuracy of the results and the time consumption, each scene needed a modification in the feature extraction radius, which is not desirable in our application at all. According to these results and tests we suggest that SHOT descriptors along with the first algorithm will be the best choice in such applications that need object recognition from different angles.



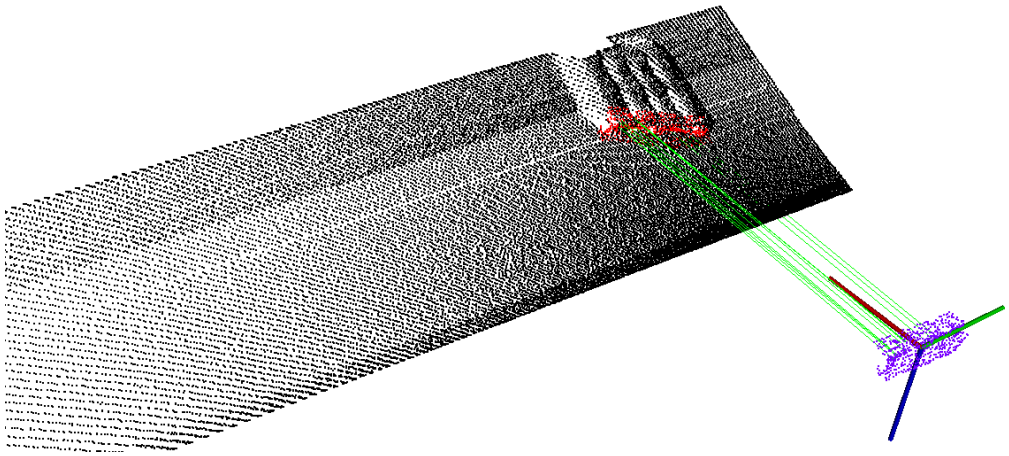
(a) Test1



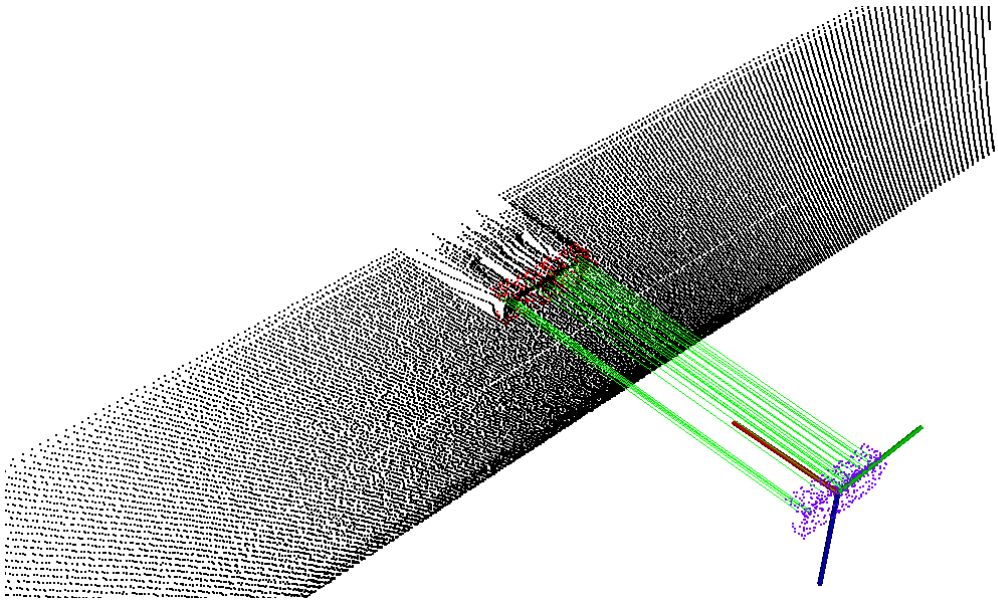
(b) Test2



(c) Test3



(d) Test4



(e) Test5

Figure 4.13: Recognition tests with real data, visual inspection (the red area shows where the pallet has been fitted, the green lines show remaining correct correspondences

5. CONCLUSION

This work covered the problem of object recognition and pose estimation in a point cloud data structure using PCL (Point Cloud Library). The motivation for this work was the need of industries to have standard pallet recognition, since most of the trucks and forklifts are equipped with 3D sensor devices.
We have started with a brief explanation of the problem and the aim of this project. Different software, hardware and tools were introduced step by step. The programming of this project was done with an extensive use of an open source library which also supports users for their problems and issues. The PCL was introduced along with different libraries for various mathematical and graphical operations. We then developed our project step by step by giving necessary theoretical knowledge of basic 3D data processing concepts and what tools PCL had to offer. In each section the piece of relevant code was also presented, thus the main code was described at the same time.
On the implementation side and more practical issues we then addressed communication, servo controller, and the main point cloud production phases. To simply describe the complete scenario, we assumed that an upper level program calls for the procedure to execute, as a result gathering required data for the first time was started. In this first scan, which we called it open loop phase, the aim was to acquire information from a wider angle of view, since we had no information about the possible position of a pallet. This point cloud was then fed to the main C++ program written using PCL. Here all the previously described procedures and algorithms took place, and assuming there was at least one object present in the captured scene, the result was a 4*4 translation matrix showing the relative position of the found object in the scene to the model in its data base.
At this point the second step commenced which we called it closed loop detection phase. In this phase, the result matrix of the detection was fed to the designed controller and was used there to control movement of the machine towards the pallet. As in this step we had relatively accurate information of pallet position with respect to the machine, we were able to control the laser servo system to focus on a specific desired area as well. This resulted in smaller scan angle intervals, smaller point clouds and less processing time. As the machine started moving towards the pallet based on open loop detection phase, we continued focusing on the goal area,

capturing point clouds, feeding it to PCL program, and feeding the result back to the controller in order to close our feedback control loop and correct the machine's path towards the goal.
Such real time applications not only have strict restrictions on time and accuracy, but also require extremely robust procedures. To investigate this issue we compared two main algorithms mostly used in literature and also present in PCL along with two of the main descriptor types. Each descriptor was tested in each of the algorithms and the results were represented. We discovered a valuable result from the experiments not mentioned previously. Not all types of descriptors will perform fine with all types of algorithms. Most of the present research have focused on single scene experiments, object categorization, or semantic maps, whereas in this application we required a procedure to detect one object robustly from different angles and distances.
As all algorithms have parameters to be set by the user, our main problem was to find an algorithm which will perform fine with one setting. Results have proven the use of SHOT descriptors along with finding and rejecting correspondences will be the best choice.

5.1 Feature Work

The time constraint did not allow for a complete implementation on the machine. In this phase we have reached good results on the simulator environment and solved communication issues. The next step for this project is to make an executable of this program and implement it on the machine. It will then be possible to research more on practical issues and investigate on the real aspects which will be a good research topic.

REFERENCES

[1] Radu Bogdan Rusu, "Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments", PhD Dissertation Technische Universitat Munchen, available at files.rbrusu.com/publications/RusuPhDThesis.pdf

[2] Xin Zhang, John Morris, Reinhard Klette CITR, "Volume Measurement Using a Laser Scanner", Computer Science Department, The University of Auckland, September 2005

[3] Andreas Nuechter, "3D Robotic Mapping, Springer Tracts in Advanced Robotics", volume 52, Springer Berlin / Heidelberg, 2009

[4] Stefano Squizzato, "Robot bin picking:3D pose retrieval based on Point Cloud Library", University of Padova, December 2012

[5] Oliver van Kaick, Hao Zhang, Ghassan Hamarneh, Daniel Cohen-Or, " A Survey on Shape Correspondence", Computer Graphics Forum, Volume 30, Issue 6, pages 1681-1707, September 2011

[6] Luis A. Alexandre, "3D Descriptors for Object and Category Recognition: a Comparative Evaluation", Workshop on Color-Depth Camera Fusion in Robotics at the IEEE/RSJ International Conference on Intelligent Robots and Systems(IROS), Portugal, October 2012

[7] R. B. Rusu, A. Holzbach, G. Bradski, and M. Beetz. "Detecting and segmenting objects for mobile manipulation", In Proceedings of IEEE Workshop on Search in 3D and Video (S3DV), held in conjunction with the 12th IEEE International Conference on Computer Vision (ICCV), Japan, September 2009

[8] K. Hammoudi, F. Dornaika, B. Soheilian, N. Paparoditis, "Extracting Wire-frame Models of Street Facades from 3D Point Clouds and the Corresponding Cadastral Map", International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences (IAPRS), vol. 38, France,September 2010

[9] Sitek et al., "Tomographic Reconstruction Using an Adaptive Tetrahedral Mesh Defined by a Point Cloud", IEEE Trans. Med. Imag., 2006

[10] Radu Bogdan Rusu, Steve Cousins, "3D is here: Point Cloud Library (PCL)", Robotics and Automation (ICRA), IEEE International Conference on Robotics and Automation (ICRA), May 2011

[11] Silvio Filipe, Luis A. Alexandre, "A Comparative Evaluation of 3D Keypoint Detectors in a RGB-D Object Dataset, 9th International Conference on Computer Vision Theory and Applications", Portugal, January 2014.

[12] Chavdar Papazov, Sami Haddadin, Sven Parusel, Kai Krieger and Darius Burschka, "Rigid 3D geometry matching for grasping of known objects in cluttered scenes", The International Journal of Robotics Research, 2012

[13] Chavdar Papazov and Darius Burschka, "An Efficient RANSAC for 3D Object Recognition in Noisy and Occluded Scenes", In Proceedings of the 10th Asian Conference on Computer Vision (ACCV'10), November 2010

[14] Radu Bogdan Rusu, Nico Blodow, Michael Beetz, "Fast Point Feature Histograms (FPFH) for 3D Registration", International Conference on Robotics and Automation (ICRA), 2009

[15] R. B. Rusu, N. Blodow, Z. C. Marton, and M. Beetz, "Aligning Point Cloud Views using Persistent Feature Histograms", 21st IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), France, September 2008

[16] http://www.xbox.com/en-US/kinect

[17] Jens Wittrowski, "Furniture Recognition using Implicit Shape Models on 3D Data", Master Thesis in Intelligent Systems, Bielefeld University, October 2012

[18] http://www.pointclouds.org/documentation/tutorials/walkthrough.php

[19] Jens Behley, Volker Steinhage and Armin B. Cremers, "Performance of Histogram Descriptors for the Classification of 3D Laser Range Data in Urban Environments", IEEE International Conference on Robotics and Automation (ICRA), pp. 4391-4398, 2012

[20] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, John Hsu, "Fast 3D Recognition and Pose Using the Viewpoint Feature Histogram", IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2010

[21] F. Tombari, S. Salti, and L. Di Stefano, "Unique signatures of histograms for local surface description", in Proceedings of the 11th European conference on computer vision conference on Computer vision: Part III. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 356-369

[22] Zhengyou Zhang, "Iterative point matching for registration of free-form curves and surfaces", International Journal of Computer Vision, October 1994, Volume 13, Issue 2, pp 119-152