Master’s Thesis

3D Pose Estimation of Objects Using a Graph Based Approach

Author: Muhammad Humza Shakeel
Matriculation No: 21228223

Referees:
Prof. Dr. Florentin Wörgötter
Dr. Minija Tamosiunaite

Supervisors:
Dr. Jeremie Papon
Markus Schoeler

A thesis submitted in fulfillment of the requirements for the degree of Masters in Applied Computer Science in the

Computer Vision Group
Department of Computational Neuroscience
Third Institute of Physics – Biophysics
Georg-August-Universität Göttingen

July 2015
Declaration of Authorship

I, Muhammad Humza Shakeel, declare that this thesis titled, 3D Pose Estimation of Objects Using a Graph Based Approach and the work presented in it are my own. I confirm that:

- I produced the present work myself only with the help of the indicated aids and sources.
- This work has not been submitted or reproduced in any form in any Institute or University.
- Where I have consulted the published work of others, this is always clearly attributed.
- I have acknowledged all main sources of help.

Signed: 

Date:
3D Pose Estimation of Objects Using a Graph Based Approach

Abstract

This thesis deals with the 3D pose estimation problem for rigid objects. Pose estimation means approximation of the six degrees of freedom of the pose of an object in consideration, i.e. the three rotation angles and the three translation components. Pose is always defined relative to a base frame. For example, we have a model object in its own reference frame. The goal of the pose estimation problem would be to find the best transformation $T$ which aligns the model object to the object perceived in a scene (in a scene reference frame). In this work, we assume that objects are isolated and segmented from a scene recording.

The main contribution of the work is the Graph Based Pose Estimation Algorithm (GBPE). GBPE is a novel, robust and embarrassingly parallel pose estimation algorithm. The core of the algorithm is to use surface graphs to estimate poses. The method aims at the minimization of wrong correspondence matching by segmenting both point clouds into small geometrically aligned patches and then finding similarities only between relevant segmented areas.

The algorithm starts by segmenting both point clouds into small geometrically aligned patches using the Difference of Normal and Region Growing Based Segmentation. The next step is to draw surface graphs of segmented patches. This is followed by calculating features on every node of both graphs and classifying edges as convex or concave is the next step of the algorithm. Finally, the algorithm finds subgraphs and applies the Graph Based Iterative Closest Point Algorithm (GICP) to the remaining subgraphs. The best subgraph is the one which has the least mean squared error.

We find that GBPE works quite well even when the objects have no color. We show quantitatively that our simple graph based pose estimation approach is able to compete with more complex state of the art methods; these methods use sophisticated mathematical approaches to compute poses. Furthermore, we observe that our algorithm is highly parallelizable. Feature matching and applying GICP are the two easily parallelizable steps of the algorithm.

Keywords: Computer Vision, 3D Pose Estimation, Surface Graphs, Segmentation
## Contents

**Declaration of Authorship**

**Abstract**

**Contents**

**List of Abbreviations**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>The Pose Estimation Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>State of the Art</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Overview and Organization</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Generation of Dataset</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Hidden Point Removal for Data Set Generation</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Back-face Culling</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Ray-tracing for Hidden Point Removal</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Limitations in the Algorithm</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>Segmentation of a 3D Point Cloud and Generation of Surface Graphs</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Supervoxel Clustering</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Difference of Normal based Segmentation</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Difference of Normal and Region Growing based Segmentation</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Generation of Surface Graphs of Segmented Patches</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Discussion</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Feature Extraction on Segmented Patches and Subgraph Matching</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Feature Extraction</td>
<td>27</td>
</tr>
<tr>
<td>4.1.1</td>
<td>SHOT Feature</td>
<td>28</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Surface Curvature</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Feature Matching</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Subgraph Matching</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td>3D Point Cloud Alignment for Pose Estimation</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Graph based Iterative Closest Point Algorithm for Point Cloud Alignment</td>
<td>38</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Results and Comparison with the State of the Art Algorithms</td>
<td>42</td>
</tr>
</tbody>
</table>
Contents

5.2 Discussion ................................................................. 48

6 Conclusion and Future Work ........................................... 49
  6.1 Summary of Contribution ............................................. 49
  6.2 Limitations and Direction of Future Work ....................... 50
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_s$</td>
<td>Model point cloud</td>
</tr>
<tr>
<td>$P_t$</td>
<td>Scene point cloud</td>
</tr>
<tr>
<td>GBPE</td>
<td>Graph based pose estimation algorithm</td>
</tr>
<tr>
<td>GICP</td>
<td>Graph based iterative closest point algorithm</td>
</tr>
<tr>
<td>SICP</td>
<td>Standard iterative closest point algorithm</td>
</tr>
<tr>
<td>AP</td>
<td>Alignment prerejective</td>
</tr>
<tr>
<td>CG</td>
<td>Correspondence grouping</td>
</tr>
<tr>
<td>VCCS</td>
<td>Voxel cloud connectivity segmentation</td>
</tr>
<tr>
<td>$R_{voxel}$</td>
<td>Voxel resolution</td>
</tr>
<tr>
<td>$R_{seed}$</td>
<td>Seed resolution</td>
</tr>
<tr>
<td>$R_{search}$</td>
<td>Search radius</td>
</tr>
<tr>
<td>DoN</td>
<td>Difference of normal</td>
</tr>
<tr>
<td>SHOT</td>
<td>Signature of histogram and orientation</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean squared error</td>
</tr>
<tr>
<td>PC</td>
<td>Point cloud</td>
</tr>
<tr>
<td>BFC</td>
<td>Back-face culling</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The part of our brain which is responsible for processing visual information is called the visual cortex. It receives immense amount of information from the retina, filters it and enables us to take decisions without even thinking about them. Moreover, this complex machinery also helps us to solve many visual problems in every second of our lives such as identifying and differentiating between objects, avoid collisions, navigating in an environment etc.

The science that duplicates the visual cortex using computer-aided solution to solve complex visual problems in robotics is called computer vision. The goal of computer vision is to make machines intelligent so they can assist us in our daily activities. Although, research in computer vision has been progressing since many decades but recent advances in technology and availability of cost efficient 3D sensors like Microsoft Kinnect have opened a new paradigm for researchers in the field. Some applications of computer vision can be found in the field of automation of transport, packaging, rescuing and in countless different industries.

Yet, unlike mankind, these robots lack intelligence. Programming these robots and making them work for us requires not only computing power and memory but also intelligent and innovative programming methodologies. In this thesis, we focus on the sub-problem of computer vision, which is called Pose Estimation.
1.1 The Pose Estimation Problem

The problem of pose estimation is very essential in computer vision with many applications in augmented reality, mobile robotics etc. Formally, pose estimation is defined by Grimson [16] as follows:

**Definition 1.** By Pose we mean the transformation needed to map an object model from its inherent coordinate system into agreement with the sensory data.

In other words, 3D pose estimation means approximation of the six degrees of freedom of the pose of an object in consideration, i.e. the three rotation angles and the three translation components. Pose is always defined relative to a base frame. For example, we have a model object in its own reference frame as depicted in figure 1.1. The goal of
the pose estimation problem would be to find the best transformation $T$ which aligns the model object to the object perceived in a scene (in a scene reference frame) as depicted in figure 1.3. However, in this work, we assume that objects are isolated and segmented from a scene recording as depicted in figure 1.4.
While estimating poses in robots many difficult questions arise which needs to be answered, some of which can be found below:

- What type of features do we have to calculate?
- What type of transformation is needed?
- What type of objects are considered?

We will discuss about features, different transformations, object types etc throughout this thesis. But, following we will show some state of the art works in the field.

### 1.2 State of the Art

Pose estimation is a very important research area in computer vision and has been investigated for the past five decades. The PhD thesis of W. Szczepanski [40] gave an overview of early works in the field, beginning in the year 1879. In our thesis, we will also present our variant of iterative closest point (ICP) algorithm for point cloud alignment, that is why we will also shed some light on different alterations of ICP algorithms. In order to have a better understanding of the literature, we have separated the work into the following different topics:

1. **Improvements in Iterative Closest Point Algorithm:** In this topic we will present an overview of different variants of iterative closest point algorithm and will
show how they have improved over time in terms of accuracy and computational efficiency.

2. Different Pose Estimation Algorithms: In this topic we will give an overview of how different features can be used to estimate the poses of 3D objects. Moreover, we will also shed some light on how they have improved over time and what are their limitations.

1. Improvements in Iterative Closest Point Algorithm:

The iterative closest point algorithm for point cloud alignment was first proposed by Besl and McKay [3]. Since then the algorithm has been modified several times as shown in figure 1.6. Overall the algorithm can be divided into three steps: 1) Finding the similarity between the model point cloud \( P_s \) and the scene point cloud \( P_t \). 2) Finding the transformation matrix \( R \) in such a way that it minimizes the mean squared error. 3) Applying \( R \) to \( P_s \) and calculating mean squared error.

![Graph showing number of research papers published from 1992 to 2010](image)

**Figure 1.6:** Graph showing number of research papers published from 1992 to 2010 [30] [28].

If we analyze the current literature, different strategies can be adopted for the selection of points. Besl and McKay [3] suggested the use of all the points for ICP. On the other hand, Turk et al. [44] proposed to apply uniform sampling before selecting the points. Whereas applying random sampling on every iteration has been proposed by Masuda et al [20].
Different approaches while point matching can also be adopted. For example, the nearest neighbor strategy by Besl and McKay [3] is a good and fast way to match the points, which can be achieved using kd-tree or closest point caching (Simon et al. [37]). Another way of finding similar points is referred to as ”normal shooting”, which is presented by Rusinkiewicz [30] [8]. In normal shooting, the algorithm finds an intersection between the destination surface and the ray which begins from the source point in the direction of the source point’s normal [8]. Based on color and angle between normals, a compatibility matrix can also be used to find similar points as discussed in [29] by Rosenhahn et al.

Moreover, different strategies can also be adopted for rejecting points. One way is to reject all those points whose distance is higher than the user specified value. Another way which is demonstrated in [29] by Rosenhahn et al. rejects worst n% of points. Those points can also be rejected whose multiple of the standard deviation distance is higher than the corresponding point error (Masuda et al. [20]). Points that are geometrically not aligned with neighboring points can also be discarded as discussed in [10] by Dorai et al.

2. Different Pose Estimation Algorithms:

Gall et al. [12] presented a method for pose estimation which uses not only 3D shape of an object as described in [5] by Brox et al. but also its texture. The information about texture makes correspondences strong which leads to a robust pose estimation algorithm. However, this method failed to estimate the poses when models were homogeneous and distinctive features were difficult to extract (Tomas et al. [4]).

Another method which is presented by Tomas et al. [4] tried to solve the problem of Gall et al. [12]. They have combined two features: One type solves the shortcomings of the other. The above pose estimation algorithm [12] is only suitable when unique features are easily extractable. When objects are homogeneous, keeping track of pose from one frame to another is also challenging. In [4] they have used object contours as one feature to match the free form object surfaces. Object contour works well if two conditions are satisfied: 1) To estimate the pose, the object’s silhouette has to contain enough information. 2) For pose tracking, the motion should be slow enough to avoid a local minima. The second feature which they suggested is the use of features from optical flow i.e. correspondences from 2D points from successive images. For pose tracking, if the pose in the first image is known, then it allows the construction of the 2D-3D correspondences in the second image.
A descriptor that can only be computed on feature points is proposed by Chen et al. [7]. Feature points are those points in which the shape variation is large. This is an integrated local surface descriptor for object recognition and pose estimation. Next, Chen et al. calculated the local surface properties which includes: Centroid, local surface type and 2D histogram. The surface patch has been classified into different categories based on the mean and Gaussian curvature of the feature point. Mean and standard deviation have been calculated for every shape index and used as indexes in a hash table. Models containing similar surface descriptors are cast by votes. Models are then hypothesized based on potential correspondences between local surface patches. Finally, the iterative closest point algorithm has been applied to check the validity of the algorithm.

1.3 Overview and Organization

We have seen different pose estimation techniques in the previous section 1.2. Some of them uses different features to make pose estimation robust, some modifies ICP to make pose estimation less error prone, but none of them have minimized the chances of wrong correspondences by reducing the area. In our work, we use a graph based approach to estimate the poses of 3D rigid point clouds. We call our algorithm the Graph Based Pose Estimation Algorithm (GBPE).

In our pose estimation algorithm we assume that objects are isolated and segmented from a scene recording. GBPE begins by taking two point clouds. One is a scene point cloud whose properties are fully known to the algorithm and we denote it by $P_t$. The second point cloud is called a model point cloud which is self-occluded and randomly transformed and we denote it by $P_s$. The goal of the algorithm is to estimate the pose and align both of them. To estimate the pose, the first step in our algorithm is to decompose both point clouds into small geometrically aligned patches. After a segmentation, the algorithm generates surface graphs of segmented patches of both point clouds. We now calculate features on every node of both graphs. We assume that $P_s$ is a subset of $P_t$, by taking this assumption we find all the possible valid subgraphs of $P_s$ graph in $P_t$ graph. On the remaining subgraphs, we apply the graph based iterative closest point algorithm and choose only one subgraph which has the least mean squared error.

We benchmark our algorithm using 47 different objects (scene point clouds), and for every object we generate four different random transform objects (model point clouds) after applying random transformation $T$. In total, the size of the dataset which includes
all points clouds (scene point clouds + model point clouds) is 235. Since we are dealing with the synthetic data, and to make it act like a real world scenario, we have to remove all those points that are not visible given a certain viewpoint after applying random transformation $T$. We apply hidden point removal algorithm and generate the dataset in chapter 2.

The work is organized as follows: First, in chapter 2 we present a hidden point removal algorithm for dataset generation. Second, in chapter 3 we present two segmentation algorithms for point clouds and generate surface graphs. Third, in chapter 4 we compute features on segmented patches of both point clouds and find all possible valid subgraphs. In chapter 5 we apply a graph based iterative closest point algorithm (GICP) on the remaining subgraphs to check the validity of our pose estimation algorithm. Finally, in chapter 6 we conclude our thesis and give the direction of future work.
Chapter 2

Generation of Dataset

In this chapter, we generate the dataset that would be used to benchmark our pose estimation algorithm. In this thesis, we assume that objects are isolated and segmented from a scene recording. Since we are dealing with the synthetic data and to mimic it act like a real world scenario we have to remove all those points of a model point cloud that are not visible given a certain viewpoint as shown in figure 2.1. To remove hidden points of point clouds, we apply hidden point removal algorithm as discussed in the section 2.1.

![Figure 2.1: Left: A point cloud after hidden points are removed. Top right: A complete point cloud. Bottom right: A point cloud after hidden points are removed from another viewpoint.](image)

2.1 Hidden Point Removal for Data Set Generation

The ability of identifying and removing points that would be hidden if the object were solid is called hidden point removal in computer graphics and vision. Prior to
understanding the detection of hidden points of a point cloud, we need to understand what a point cloud is and how it is different from a polygon mesh. A point cloud is a combination of many points in a three dimensional coordinate system as shown in figure 2.2 on the left. A polygon mesh consists of edges, vertices and faces that defines the shape of a polygon mesh in three dimensions as depicted in figure 2.2 on the right. Since point clouds consists of only points not surfaces, the problem of visible point detection arises as shown in figure 2.3.

Hidden point removal algorithms can be classified into two main categories [1]: Surface triangulation based methods and voxel based methods. In the first approach, the algorithm has to create the triangles between the points of a point cloud to make it like a polygon mesh, this method is computationally expensive and requires a lot of memory. This approach further consists of three main algorithms: Testing the surface normal direction, ray-triangle intersection, z-buffering method. In the second approach, points are represented as voxels and does not require creation of triangles. This approach can also be classified into three major categories: Voxel-ray intersection, buffering techniques, ray-tracing technique. Another state of the art method which is proposed by Katz et al. [1, 14] considers only those points of a point cloud that are located on the convex hull after transformation. In this thesis, we use a ray-tracing based hidden point removal algorithm due to its robustness as compared to other algorithms discussed before.

2.1.1 Back-face Culling

Back-face culling (BFC) is an efficient pre-processing step before applying any other hidden point removal algorithm. It removes all those points that are facing backward by
just computing the dot product between the point normal and the viewpoint as shown in figure 2.6.

After applying BFC to a point cloud, we apply a ray-tracing based hidden point removal algorithm as discussed in the section 2.1.2.

2.1.2 Ray-tracing for Hidden Point Removal

Ray tracing is a standard method for generating high definition images in computer graphics. The idea of ray-tracing is inspired by nature and very close to how we see things as mentioned by Ibn al-Haytham (c. 965-1039) in his theory of vision [13]. He argued that we see objects in the real world due to light, and if there is no object in the world then we will not be able to see the light as well. The concept of ray tracing is to render only those parts of the model which are visible on the screen. This is performed
by casting a ray from a viewpoint to the object, and take only those points which are closest to a viewpoint after intersection. Depending on the nature of the object, rays might be reflected or refracted. This concept is called backward ray tracing [17].

Originally ray tracing has been used for meshes, but for the first time the concept of ray tracing for point geometry has been used by Péroche et al., 2000 [35] without converting points into meshes. In our implementation of hidden point removal, we have taken the concept from [1, 14, 17, 21, 35].

Let $P$ be a point cloud consisting of $N$ number of points in a three dimensional space. We generate $N$ number of rays from a viewpoint $V$ to all the points of a point cloud and we divide each ray into $m$ equal parts using the formula:

\[
\tilde{x} = m \times x_n + k \times x_1
\]

\[
\tilde{y} = m \times y_n + k \times y_1
\]

\[
\tilde{z} = m \times z_n + k \times z_1
\]

where $\tilde{x}, \tilde{y}, \tilde{z}$ are the new dividing points on the line, $m$ and $k$ are the ratios. For example, if we have to divide a line into hundred equal parts, we start the algorithm by setting $m = 100$ and $k = 1$ and iteratively increase $k$ and decrease $m$ by 1. $x_1, y_1, z_1$ are the coordinates of a viewpoint while $x_n, y_n, z_n$ are the coordinates of the $n^{th}$ point of a point cloud $P$.

\[\text{Figure 2.5: Red points are the points of a point cloud, blue lines are the ray casted from a viewpoint to every point of a point cloud, yellow bars are the dividing points on the lines.}\]
The total number of $m$ depends on the curvature and density of a point cloud. In dense point clouds we have to choose the small value of $m$ while in sparse point clouds large value of $m$ has to be chosen. However, finding the accurate number of $m$ is a non-trivial task. After dividing the line into $m$ equal parts, radius has to be drawn around $m_i$. The next step is to find all the neighboring points that lies within the radius. If the total number of the neighboring points are greater than the prescribed threshold, then we mark that point as hidden else we move to $m_{i+1}$. This is a two step iterative algorithm: 1) we have to do it for $N$ number of points and 2) we have to divide every line $m$ number of times.

![Figure 2.6: The yellow point is the point of consideration, red points are the other points of a point cloud, blue line is casted from a viewpoint to the yellow point, orange bars are the dividing points on the line. Black color indicates the radius, sky blue points are the points that lies within the radius while green point is responsible for hiding the yellow point.](image)

We have tested our algorithm on 47 different point clouds. For every point cloud we have generated four different random transformed point clouds, and then we removed their hidden points (188 in total). The time taken to remove the hidden points varied from one point cloud to another. Some point clouds that we have used were very densely and consisted of more than 1,800,000. One point cloud which consisted of 1,891,725 points took around 54 minutes to remove all the hidden points. The average time for generating 188 point clouds out of 47 complete point clouds is 42 minutes. All measurements were recorded on an Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz processor.
2.1.3 Limitations in the Algorithm

A key limitation of this algorithm is that we have to choose an optimal radius size, but there are cases where no radius size works as shown in figure 2.9. Another limitation of this algorithm is that it is computationally expensive because for every point we have to cast a ray, every ray has to be divided \( m \) times and for every \( m_i \) we have to find nearest neighbors.
Figure 2.9: **Left**: The size of the radius is big and it is removing one point (on the blue line) which is visible. **Right**: To avoid this problem, we make the radius size small but it is not removing the hidden point (on the green line) because it is not coming within radius and line is passing between points.

Our data set is available online at:

Partial Point Clouds: [https://dl.dropboxusercontent.com/u/4876897/University%20of%20Goettingen/Thesis/DataSet/DataSet.zip](https://dl.dropboxusercontent.com/u/4876897/University%20of%20Goettingen/Thesis/DataSet/DataSet.zip)

Full Point Clouds: [https://dl.dropboxusercontent.com/u/4876897/University%20of%20Goettingen/Thesis/DataSet/DataSet_FullPointClouds.zip](https://dl.dropboxusercontent.com/u/4876897/University%20of%20Goettingen/Thesis/DataSet/DataSet_FullPointClouds.zip)
Chapter 3

Segmentation of a 3D Point Cloud and Generation of Surface Graphs

The process of decomposing a point cloud into meaningful patches is called a point cloud segmentation. The aim of this process is to cluster similar voxels to get a better understanding of a point cloud. Segmentation is one of the most studied topic in computer vision with many applications in medical imaging, object detection, face detection, video surveillance etc. Since we have to subdivide a point cloud into meaningful segments, some questions that immediately arise are:

- What is the required level of segmentation?
- Where to stop the subdivision process? etc

Determining the granularity of any segmentation algorithm is a non-trivial task and there is no definite segmentation algorithm which exists for all types of applications with the same segmentation level. In other words, all segmentation algorithms are application dependent. For example, we have to design an application that detects the movement of cars on road. The goal of this application is to detect only cars, so the first level of segmentation is to extract the road from a full point cloud. Second, we have to extract cars from already segmented road point cloud. In this particular application we have to stop our subdivision process once we have identified all the vehicles.
Segmentation algorithms are divided into two main categories: A discontinuity based approach and a similarity based approach. In a discontinuity based segmentation approach [15], the goal is to detect isolated points, edges, lines etc. While in a similarity based approach [27] the main aim is to cluster those voxels which are similar in some sense, e.g. color, intensity etc. Some popular techniques in this category are region based approach and threshold based approach. In a region based approach, the algorithm takes a voxel at random and groups all other neighboring voxels which are similar in intensity. In a threshold based approach, all voxels that are within a particular threshold value are grouped together.

In this chapter, we propose two segmentation algorithms which uses Voxel Cloud Connectivity Segmentation (VCCS) [26] as a pre-processing step. VCCS not only downsamples a point cloud by imposing a 3D grid but also clusters similar voxels into perceptually meaningful regions. In the first approach, we try to segment a point cloud using only normal difference between supervoxel clusters and we call this segmentation algorithm difference of normal based segmentation. In the second approach, we combine the region growing technique with the previous approach and try to solve the limitations of the first. The reason why we have used difference of normal and region growing based segmentation because it is computationally efficient and also robust as compared to other segmentation algorithms. We also compare the results of our segmentation algorithm with one state of the art algorithm which is also implemented in PCL. Finally, we generate surface graphs of all segmented patches of both point clouds i.e. $P_s$ and $P_t$. 

Figure 3.1: Left: A complete point cloud. Right: A point cloud after segmentation: The desired output.
3.1 Supervoxel Clustering

A segmentation of a point cloud using all data points is a computationally expensive task. Several pre-processing methods are available which downsamples a point cloud before applying any segmentation algorithm. In our work, we have used the Voxel Cloud Connectivity Segmentation (VCCS) [26], which is a state of the art method for generating supervoxel clusters. It downsamples a point cloud by imposing a voxel-grid filter, also known as voxelization. It also maintains the neighborhood information between voxels which is achieved by using adjacency octree. There are three different ways of storing adjacencies in 3D space — 6-, 18- and 26- —. However, if 26- adjacency is used the chances of missing connections drops substantially.

The algorithm for generating supervoxel clusters starts by selecting a number of seed points. This is achieved by dividing a space into a voxelized grid with a defined seed resolution $R_{seed}$ which is higher than the voxel resolution $R_{voxel}$. The small $R_{seed}$ value leads to more supervoxel clusters on a point cloud. Points that are isolated in space are considered as noise, small search radius $R_{search}$ has been drawn around each seed, and keep only those seeds which have enough supervoxels. The algorithm grows iteratively and combines the neighboring supervoxels with the minimum distance to create supervoxel clusters. It also ensures that supervoxel clusters do not flow across object boundaries. In this thesis, we are only interested in centroid normal of supervoxel clusters and the adjacency between them.
3.1.1 Difference of Normal based Segmentation

After clustering supervoxels into perceptually meaningful regions, the next step is to merge supervoxel clusters that are on the same surface to perform a segmentation. Thus, we have to divide a point cloud into small patches that are geometrically aligned with each other as we have depicted in figure 3.1. In our approach, we have used supervoxel cluster centroid normals to merge all supervoxel clusters that are within the prescribed threshold. The idea is very simple, it is an iterative algorithm which starts by taking one supervoxel cluster and calculates its normal difference with all other supervoxel clusters of a point cloud. All supervoxel clusters that are within the range would be merged with the initial supervoxel cluster. After the first iteration, all supervoxel clusters that are within the range would become one surface as we have shown in figure 3.4. This whole concept of segmenting a point cloud is inspired by the concept called difference of normal based segmentation which is computationally efficient and also very effective for unorganized point clouds.

![Figure 3.4: Left: A complete voxelized point cloud. Right: The difference of normal based segmentation algorithm has successfully segmented the plane after first iteration][41].

Our approach of segmentation is unique from other difference of normal based segmentation in two ways: Firstly, instead of using every voxel, we use supervoxel clusters which makes our implementation fast because we need less comparisons. Secondly, since we are heavily relying on the quality of the VCCS, which is efficient even when we have noisy data. This makes our algorithm robust and allows us to provide single threshold value for segmentation.

However, our method is very basic and takes into account only those point clouds that have surfaces pointing in different directions. This is because our algorithm takes one supervoxel cluster, compares it with all supervoxel clusters instead of comparing it with those supervoxel clusters that are on the same surface. Unfortunately, mathematically there is no solution available that helps us to distinguish between two supervoxel clusters that are far from each other but pointing in the same direction. It is like figuring out a point which lies on the line without knowing the line equation. We have to devise
Chapter 3. Segmentation of a 3D Point Cloud and Generation of Surface Graphs

Figure 3.5: Angle between two normal vectors: \( \cos \theta = \frac{(\vec{n}_i \cdot \vec{n}_j)}{||\vec{n}_i|| \cdot ||\vec{n}_j||} \).

an algorithmic solution that would only take supervoxel clusters which are on the same plane, and distinguish it with all the other supervoxel clusters that are on some other plane but pointing in the same direction.

Figure 3.6: Top left: A complete voxelized point cloud. Top right: The difference of normal based segmentation algorithm failed to segment a point cloud: depiction after first iteration. Bottom: The difference of normal based segmentation algorithm failed to segment a point cloud: depiction after first iteration from another viewpoint.

3.1.2 Difference of Normal and Region Growing based Segmentation

Region growing is a well known segmentation technique in computer vision. In this algorithm, one pixel is selected at random as an initial starting point and iteratively combines itself with similar neighbors. In our case, the splitting of a point cloud begins by taking a single supervoxel cluster as an initial starting point, and iteratively computes difference of normal with other neighboring supervoxel clusters. If the difference is within the limit, then it merges with the initial supervoxel cluster and the size of the initial supervoxel cluster would increase. If the normal difference with any of its neighboring
supervoxel cluster is greater than the given range, then that supervoxel cluster would be considered in the next iteration. After each iteration, one surface would be segmented, which means, if a point cloud consists of five surfaces in total, a complete point cloud would be segmented after five iterations.

**Figure 3.7:** Left: A complete voxelized point cloud. Right: The difference of normal and region growing based segmentation algorithm successfully segmented a point cloud into small patches.

Since, our algorithm is highly dependent on the quality of VCCS [26], there are times when it generates supervoxel clusters at the edges of a point cloud which leads to failure of a segmentation algorithm. For instance, if we have a point cloud which has one supervoxel cluster at the edge between two surfaces and the normal difference of this supervoxel cluster with both surfaces is within the threshold. This scenario would lead to failure of the algorithm because the supervoxel at the edge would not allow two surfaces to split.

**Figure 3.8:** Depiction of supervoxel clusters between two surfaces of a point cloud.

To avoid the above mentioned problem, we have modified our algorithm and made the threshold value very small. By doing this, after performing segmentation, all surfaces would be segmented and we would have some extra surfaces which contains only supervoxel clusters at the edges. Now to get rid of these supervoxel clusters at the edges, we merge it with any of the connected surfaces. This step of the algorithm is
Chapter 3. Segmentation of a 3D Point Cloud and Generation of Surface Graphs

Figure 3.9: The difference of normal and region growing based segmentation failed to segment because of the supervoxel cluster at the edges.

called merging step. The threshold value for our dataset is $36^\circ$.

Figure 3.10: Left: A complete voxelized point cloud. Right: The difference of normal and region growing based segmentation algorithm successfully segmented a point cloud even after having supervoxel clusters at the edges.

Now we will compare the results of our difference of normal and region growing based segmentation algorithm with region growing segmentation algorithm which is implemented in the Point Cloud Library (PCL). We have compared the results in terms of time taken to segment a point cloud and in terms of accuracy. In terms of time, our algorithm is faster than the PCL based region growing segmentation algorithm. There are two reasons why our algorithm is better in terms of time: First, we apply VCCS [26] which reduces the total number of points of a point cloud. Second, instead of comparing
all the points with each other we compare only supervoxel cluster centroid normals. The average time taken by the PCL based segmentation algorithm is 23 seconds while our algorithm took around 9 seconds in total. We have compared and demonstrated the results using all 47 point clouds (see section 2.1.3) of our dataset.

![Figure 3.11: A dense point cloud consists of 1,891,725 points in total.](image)

In figures 3.12 and 3.13 we have demonstrated the results of all point clouds and the time taken to perform a segmentation. Using PCL based segmentation 28 point clouds took between 5 to 20 seconds while our algorithm segmented 45 point clouds by taking only between 4 to 14 seconds. Remaining 11 point clouds took around 25 to 40 seconds and 4 point clouds took around 43 to 60 seconds using PCL based segmentation. High density point clouds takes more time to perform a segmentation. A point cloud depicted in figure 3.11 took 101 seconds using PCL based segmentation and 26 seconds using our algorithm to do the same segmentation. Overall, in terms of time, our algorithm performs segmentation very fast.

We have compared the results in terms of accuracy as well. Our algorithm successfully segmented 39 point clouds out of 47. The reason of failure can be found in supervoxel clusters at the edges of a point cloud which at times joins two non-geometrically aligned surfaces as one. The PCL based segmentation algorithm performs little better than our algorithm in this regard and successfully segmented 41 point clouds out of 47.

All the measurements were recorded on an Intel(R) Core(TM) i7-3610QM CPU @ 2.30 GHz processor.
Figure 3.12: Point clouds in bins and their time taken to perform a segmentation using region growing based segmentation which is already implemented in PCL.

Figure 3.13: Point clouds in bins and their time taken to perform a segmentation using difference of normal and region growing based segmentation.
3.2 Generation of Surface Graphs of Segmented Patches

After segmenting both $P_s$ and $P_t$ into small geometrically aligned patches, the next step is to generate surface graphs. In computer science and mathematics, graphs can be used to model the relationship between objects. In graph theory, objects that are in consideration are called nodes and the lines joining them are called edges.

In our attempt of pose estimation, we have used surface graphs to model the relationship between segmented patches of both point cloud i.e. $P_s$ and $P_t$. The overall idea is to decompose (see section 3.1.2) both point clouds into small geometrically aligned patches. After a segmentation, the algorithm draws surface graphs of segmented patches and finds all possible structures of $P_s$ graph in $P_t$ graph (see section 4.3). Finally, the algorithm trims wrong subgraphs and applies the graph based iterative closest point algorithm (see section 5.1) to the remaining subgraphs.

We have used a Boost Graph Library (BGL) to create surface graphs. It is an open source library for C++. It provides a generic interface to access a graph structure but hides the implementation from the programmer. The method add_edge has been used to create a graph. It has three parameters: The first and second parameters are the nodes that we need to connect. The third parameter is a graph itself which stores the whole structure.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3_14.png}
\caption{\textbf{Left:} A complete voxelized point cloud. \textbf{Center:} Segmented patches of a point cloud. \textbf{Right:} A surface graph of segmented patches.}
\end{figure}
Chapter 3. Segmentation of a 3D Point Cloud and Generation of Surface Graphs

Code snippet of the method add_edge:

```cpp
add_edge(nodeOne, NodeTwo, myGraph);
```

One more scenario that needs to be considered here is related to those model point clouds that are not fully connected. Every model point cloud is self-occluded as shown in figure 3.15 but some are fully connected and some are disjoint. This is due to the hidden point removal algorithm as discussed in section 2.1. Hidden point removal algorithm removes points in such a way that from a certain viewpoint it seems that all surfaces are connected, however, if we rotate them, we realize that it contains two or more sub point clouds which are disjoint as depicted in figure 3.15. The graph for these types of point clouds would look something like figure 3.16.

![Figure 3.15: Left: A voxelized partial point cloud from one viewpoint. It seems that all the surfaces are connected. Right: A voxelized partial point cloud from another viewpoint. In reality, this point cloud now contains two sub-point clouds.](image1)

![Figure 3.16: Left: Segmented patches of a point cloud depicted in figure 3.15. Right: A graph containing two sub-graphs.](image2)

3.3 Discussion

In this chapter we segmented both point clouds i.e. \( P_s \) and \( P_t \) using difference of normal based segmentation. Difference of normal based segmentation was very naive and only suitable when point clouds containing surfaces that are facing in different directions. We improved segmentation algorithm by incorporating region growing approach with difference of normal based segmentation. In the end, we generated surface graphs of both point clouds.
Chapter 4

Feature Extraction on Segmented Patches and Subgraph Matching

In this chapter, we compute feature on all segmented patches of both $P_t$ and $P_s$. The feature includes: SHOT feature (see section 4.1.1) and surface curvature (see section 4.1.2). The next step is to match (see section 4.2) all curved surfaces of $P_s$ with all curved surfaces of $P_t$ and flat surfaces of $P_s$ with flat surfaces of $P_t$. Moreover, one curved surface of $P_s$ might correspond to one or many curved surfaces of $P_t$, but not to any flat surface. This way we reduce the total feature matching time of the algorithm. Once we receive all the correspondences between all segmented surfaces of both point clouds, we generate all possible structures of $P_s$ graph. Finally we validate $P_s$ graph structures in $P_t$ graph using the subgraph matching algorithm (see section 4.3).

4.1 Feature Extraction

A feature is any important point on a surface which is distinguishable in nature such as a corner, a blob or an edge. A good feature should be computationally efficient and robust at the same time. Feature extraction has been a significant research area for the past three decades. In short, features based on their size can be categorized into two main categories: Global and local. A global feature takes complete object into consideration at the same time while local feature computes a feature at point $p$ using its $k$ nearest neighbors. Local descriptors can deal with the challenges of occlusion, viewpoint variation etc. Some important features and their types are shown in table 4.1.
Every local and global feature can further be categorized into two main categories: Signature based and histogram based methods. The first category computes feature on every point $p$ of a point cloud by defining a local reference frame, while histogram based methods accumulate local topological or geometrical measurement into histograms. Histogram based methods are less robust because they compress information into bins. Some features based on their taxonomy are shown in table 4.2 ([42]).

Table 4.2: Features and their types based on the descriptive power [42].

In our work, we use Signature of Histogram of OrienTation (SHOT) [42] which is an intersection of histogram and signature based feature. The purpose of using SHOT in our work is that it is a local feature, robust and at the same time exploits both the power of histogram and signature based methods. We will discuss more about SHOT in the next section 4.1.1. Another feature that we use is called surface curvature which is also computationally efficient and enables us to reduce the total feature matching time (see section 4.1.2).

### 4.1.1 SHOT Feature

Signature of Histogram of OrienTation or SHOT [42] is a local, pose invariant, robust, multidimensional 3D descriptor. It also takes into account point density variation in the data which is due to the different sensors used. SHOT computes a unique and an unambiguous local reference frame by estimating the total least squares by eigen value decomposition of the covariance matrix $M$ as a weighted linear combination:
Chapter 4. Feature Extraction on Segmented Patches and Subgraph Matching

\[ M = \frac{1}{\sum_{i: d_i \leq R} (R - d_i)} \sum_{i: d_i \leq R} (R - d_i)(p_i - p)(p_i - p)^T \]  

(4.1)

where \( R \) is the radius, \( p \) is the point in the spherical neighborhood, \( p_i \) is the index point in the spherical neighborhood, \( d_i = ||p_i - p||^2 \) is the distance. This way they have given the distant points lesser weights. Finally, they disambiguated the signs of \( x \)-axis as:

\[ S_{x^+} = \{ i : d_i \leq R \wedge (p_i - p).x^+ \geq 0 \} \]  

(4.2)

\[ S_{x^-} = \{ i : d_i \leq R \wedge (p_i - p).x^- \geq 0 \} \]  

(4.3)

\[ x = \begin{cases} 
  x^+, & |S_{x^+}| \geq |S_{x^-}| \\
  x^-, & \text{otherwise}
\end{cases} \]  

(4.4)

same procedure also applies to \( z \)-axis, and \( y \)-axis can be obtained as: \( x \times z \).

They have also computed a set of local histograms. This is done by imposing a 3D grid on a 3D volume and computing a set of local histograms. For every local histogram, they have assembled the total number of points into bins by computing the angle \( \theta_i \) between the normal at each feature point \( n_u \) and the normal at each point within the corresponding part of the grid \( n_v \). The next step is to combine all the histograms to make it act like a real descriptor.

The structure of SHOT uses isotropic spherical grid which is divided into radial, azimuth and elevation axis as shown in figure 4.1.

\[ \text{Figure 4.1: Structure of a SHOT descriptor [42].} \]

4.1.2 Surface Curvature

The degree that tells us the bend in any surface is called a surface curvature. It is a computationally efficient and easily computable feature which only requires point normals for computation. In our work, we only need to know whether segmented patches are flat or curved which allows us to match curved patches of \( P_s \) with curved patches of \( P_t \) and flat patches of \( P_s \) with flat patches of \( P_t \). It also enables us to reduce the total
Chapter 4. Feature Extraction on Segmented Patches and Subgraph Matching

Feature matching time. The algorithm for computing surface curvature is very simple, it starts by computing the centroid of a segmented patch, once it has the index of a point in the center, it calculates normal difference with all other point normals of this segmented patch. If the normal difference with most of its points is within the prescribed threshold range i.e. 20°, we mark this surface as flat otherwise as curved.

Since our algorithm is very basic, there is one limitation that needs to be considered. After a segmentation, there are some points on the edges of segmented patches as shown in figure 4.2. These points have higher difference of normal with the centroid point normal. Due to these points the algorithm sometimes classify flat surface as curved. To avoid this problem, we set another threshold that if maximum 10% of the points on a flat surface greater than 20°, we still classify that surface as flat otherwise as curved.

**Figure 4.2:** A point normal at the edge with higher normal difference with a centroid point normal.

**Figure 4.3:** **Left:** The algorithm has successfully classified as a curved surface. **Right:** The algorithm has successfully classified as a flat surface.

4.2 Feature Matching

Feature matching is considered to be one of the computationally expensive task in every computer vision application. In our work, we have minimized the feature matching time by calculating surface curvature that allows us to match flat with flat and curved with curved surfaces. It also allows us to minimize the chances of matching wrong correspondences. In this step, we shall match the SHOT feature of all flat segmented patches of \( P_t \) with flat segmented patches of \( P_s \), and we will do the same for curved surfaces. Now, the fundamental question is: How can we efficiently match surfaces ?.
Naturally, the idea that comes to our mind is to match the exact feature vector that we have calculated on one segmented surface to all other feature vectors of other segmented surface. However this method is not feasible because it is linear and computationally very expensive. The other method that we have used is called the nearest neighbor search which is computationally efficient and also implemented in PCL \cite{34}.

In PCL \cite{34} they have used \textit{kd}-tree for the nearest neighbor search. A \textit{kd}-tree is a binary search tree where \textit{k} stands for a number of dimensions. Each node in a \textit{kd}-tree represents an axis-aligned rectangle. Every non-leaf node divides the hyperplane into two halves, known as half spaces. Points that lie on the right side are represented by a right sub-tree and points that lie on the left side are represented by a left sub-tree. There are many ways to construct and split a \textit{kd}-tree. However in PCL they have used a median point along the longest dimension of the node’s volume. These properties of a \textit{kd}-tree helps the algorithm to minimize its search space \cite{11}.

After constructing a \textit{kd}-tree, the next step is to find a query point. The nearest neighbor search is a recursive function, which starts by taking a single feature vector as a query point and returns the index of the closest point. Overall, the algorithm does three things: First, it looks into a subtree containing the target. Second, it checks whether a current node is closer to the target, if so, it updates the current best, else, it will not update anything and proceed. Third, it checks if there is any closer point in other subtree. If so, it will check the other side, if not, then the algorithm is finished. The current best is the closest point \cite{23} \cite{9}.
Since we are dealing with textured less artificial point clouds, which means in some way or the other all the points in a point cloud are some how the same. In other words, one feature vector of one segmented patch might corresponds to one or more feature vector of another segmented patch which leads to wrong correspondence matching. To avoid this problem, we have used a correspondence rejector method called the Random Sample Consensus (RSC) which is implemented in PCL. The RSC identifies inliers and rejects outliers. For some of our segmented patches, even after applying RSC the problem still exists. In our limited knowledge the only solution to minimize this problem is the use of textured point clouds or point clouds with dissimilar points. We will discuss more about this in results section 5.1.1.

<table>
<thead>
<tr>
<th>Segmented Patches of $P_s$</th>
<th>Segmented Patches of $P_t$</th>
<th>Correspondences between $P_s$ and $P_t$ in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_s 0$</td>
<td>$P_t 0$</td>
<td>58.1</td>
</tr>
<tr>
<td>$P_s 0$</td>
<td>$P_t 1$</td>
<td>52.5</td>
</tr>
<tr>
<td>$P_s 0$</td>
<td>$P_t 2$</td>
<td>0.0233</td>
</tr>
<tr>
<td>$P_s 1$</td>
<td>$P_t 4$</td>
<td>4.72</td>
</tr>
<tr>
<td>$P_s 1$</td>
<td>$P_t 6$</td>
<td>4.17</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 0$</td>
<td>47.9</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 1$</td>
<td>45.1</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 2$</td>
<td>42.3</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 3$</td>
<td>23.4</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 4$</td>
<td>29.7</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 5$</td>
<td>2.59</td>
</tr>
<tr>
<td>$P_s 2$</td>
<td>$P_t 6$</td>
<td>26.4</td>
</tr>
<tr>
<td>$P_s 3$</td>
<td>$P_t 0$</td>
<td>58.1</td>
</tr>
<tr>
<td>$P_s 3$</td>
<td>$P_t 1$</td>
<td>52.4</td>
</tr>
<tr>
<td>$P_s 3$</td>
<td>$P_t 2$</td>
<td>0.0233</td>
</tr>
</tbody>
</table>

Table 4.3: Correspondences in % between segmented patches of $P_s$ and $P_t$. Numbers representing segmented patch IDs.

Additionally, this method also allows us to match features of both point clouds parallely. In other words, one flat segmented patch of $P_t$ can be matched with one or more flat segmented patch of $P_s$ at the same time. This parallel implementation reduces the total computation time drastically as shown in table 4.4.
<table>
<thead>
<tr>
<th>Point Clouds</th>
<th>Feature Matching Time Sequentially in Seconds</th>
<th>Feature Matching Time Parallely in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point Cloud # 1</td>
<td>162</td>
<td>99</td>
</tr>
<tr>
<td>Point Cloud # 2</td>
<td>195</td>
<td>117</td>
</tr>
<tr>
<td>Point Cloud # 3</td>
<td>166</td>
<td>58</td>
</tr>
<tr>
<td>Point Cloud # 4</td>
<td>74</td>
<td>64</td>
</tr>
<tr>
<td>Point Cloud # 5</td>
<td>210</td>
<td>86</td>
</tr>
</tbody>
</table>

**Table 4.4:** Comparison between feature matching time taken sequentially and parallely. The time difference of all point clouds is visible except a point cloud # 4. The reason is that point cloud # 4 has fewer surfaces after segmentation and parallelizing them does not make much difference.

### 4.3 Subgraph Matching

After matching all segmented patches of both point clouds with each other, the next step is to generate all possible combinations of $P_s$ graphs and find all valid structures of $P_s$ graph in $P_t$ graph. To explain this, we have used the correspondences that we have calculated in the previous section 4.3 and the connections between all segmented patches of both point clouds. In tables 4.5 and 4.6 we have shown connections between segmented patches of both point clouds depicted in figure 4.6.

![Figure 4.6](image_url)

**Figure 4.6:** **Left:** A point cloud after hidden points are removed. **Top right:** A complete point cloud. **Bottom right:** A point cloud after hidden points are removed from another viewpoint.

Before we delve more into the algorithm, we have to store some feature at the edges of both graphs. This way we trim wrong subgraphs and have to apply GICP (see section 5.1) only to fewer number of subgraphs. The edges between any two nodes can be viewed as concave or convex. In mathematical set theory concavity and convexity can be defined as:
Convex Set:

**Definition 2.** For every pair of points within the set, every point on the straight line segment that joins them is also inside the set, that is to say it only traverses regions of infinite point density \[39\].

Concave Set:

**Definition 3.** There exists a pair of points within the set, for which parts of the connecting line segment are outside the set, that is to say it traverses “empty” regions (point density zero) \[39\].

![Figure 4.7: Left: Convex Set. Right: Concave Set. [39]](image)

In terms of a point cloud, an edge between any two segmented patches can be classified as concave or convex using surface normals. Convex edges are those, whose surface normals are pointing outwards as shown in figure 4.8. If the connection is convex, we can only see the outside of an object which is in consideration. In concave edges, surface normals are inside of an object and can be joined together as shown in figure 4.8 on the right.

![Figure 4.8: Left: Convex connection between two surfaces. Right: Concave connection between two surfaces. [39]](image)

If the edges between two nodes of a $P_t$ graph is convex, its equivalent in $P_s$ graph should also be convexed. After storing convex / concave feature between all edges of both graphs, the next step is to find all possible valid structures of $P_s$ graph in $P_t$ graph. Here, the validity means, the algorithm should take into account all features that we have calculated including surface curvature and the information at the edges \[4.9\].
Chapter 4. Feature Extraction on Segmented Patches and Subgraph Matching

Figure 4.9: **Left:** A $P_t$ graph depicting the connections between segmented patches and their properties. **Right:** A $P_s$ graph depicting the connections between segmented patches and their properties.

<table>
<thead>
<tr>
<th>$P_s$</th>
<th>$P_s$</th>
<th>$P_s$</th>
<th>$P_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_s0$</td>
<td>$P_s1$, $P_s3$, $P_s4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_s1$</td>
<td>$P_s0$, $P_s5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_s2$</td>
<td>$P_s3$, $P_s6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_s3$</td>
<td>$P_s0$, $P_s2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_s4$</td>
<td>$P_s0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_s5$</td>
<td>$P_s1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_s6$</td>
<td>$P_s2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Segmented patches of $P_s$ and their connection with other segmented patches.

Overall the subgraph generating and validating algorithm has two steps:

1. The algorithm generates all possible $P_s$ graphs using the correspondences that we have calculated earlier. We assume that one segmented patch of $P_s$ only corresponds to one segmented patch of $P_t$.

2. Iteratively it validates all $P_s$ graphs in the $P_t$ graph as:

   (a) Starting with the first node of the $P_s$ graph in consideration, the algorithm checks the number of connected nodes and the feature at the edges.

   (b) The algorithm goes to the same node of the $P_t$ graph using correspondences. Here, the algorithm again checks the number of connected nodes and the feature at the edges. If everything is same as step (a), then it proceeds to the second node of the $P_s$ graph (a), otherwise this $P_s$ graph would be trimmed.

   (c) When all nodes of the $P_s$ graph have successfully gone through step (a) and (b) then the algorithm is finished and we call this subgraph: Verified.
Three possible combinations from correspondence table 4.3 and their subgraphs are shown in figure 4.10. The subgraph generating and validating algorithm successfully validates $P_s$ graph no. 2 and invalidates graph no. 1 and 3.

**Figure 4.10:** Left: Table showing three possible combinations of correspondences. Right: Possible graphs of $P_s$ using correspondence table. The algorithm has successfully validated the $P_s \neq 2$ in a full graph depicted in figure 4.9 on the left.

So far, we have seen the case where the $P_s$ graph is fully connected. But there are times, when $P_s$ graphs are not fully connected and contain one or more disjoint surfaces as depicted in figure 4.11. This is due to the hidden point removal algorithm, which removes the point in such a way that from one viewpoint it seems that a point cloud is fully connected, but if we rotate it and check from another side we will see that it is actually divided into two sub-point clouds 2.8. The subgraph matching algorithm for these types of graphs would also remain the same.
4.4 Discussion

In this chapter we computed SHOT and surface curvature on all the segmented patches of both point clouds i.e. $P_s$ and $P_t$. Next, we matched same curvature segmented patches of $P_s$ with $P_t$. In the end, we generated all $P_s$ graphs using the correspondences that we have found between both point clouds and validated $P_s$ graphs in $P_t$ graph. In the next chapter, we will apply our variant of iterative closest point algorithm to the remaining subgraphs and will choose one subgraph which has the minimum mean squared error.
Chapter 5

3D Point Cloud Alignment for Pose Estimation

5.1 Graph based Iterative Closest Point Algorithm for Point Cloud Alignment

In this chapter we apply our variant of the iterative closest point algorithm (ICP) to the remaining subgraphs that we have validated in the previous section 4.3. ICP is a point cloud alignment algorithm which was first presented by Besl and McKay [3] in the year 1992. Since then the algorithm has been modified in several ways (see section 1.2 for more details). However, different point cloud registration algorithms are also available such as the thin plate spline robust point matching algorithm, kernel correlation etc, but in our work we have used ICP because it is robust, fast and also suitable for parallel architecture.

We have seen some pose estimation algorithms in section 1.2, all those methods consider complete objects for feature extraction and matching at the same time. Our pose estimation algorithm divides both point clouds into small patches and match similarities only between relevant segmented patches as depicted in figure 5.1. To align both point clouds i.e. $P_s$ and $P_t$ we have modified ICP and we call it the graph based iterative closest point algorithm (GICP). Our variant of ICP is robust, less error prone and fast as compared to the standard ICP (SICP) [3] algorithm. Using GICP, our pose estimation algorithm can compete with other state of the art algorithms as we have shown in the results section 5.1.1. The GICP is divided into six steps:
1. Finding and Matching Features:

In this step, we pair correspondences that we have calculated earlier in the section 4.3. Instead of calculating features on both point clouds at the same time as shown in figure 5.2, we have computed features only between relevant segmented patches as shown in figure 5.1 and afterwards created correspondence pairs.

\[(p_1, q_1), (p_2, q_2), (p_3, q_3), \ldots, (p_n, q_n)\]  \hspace{1cm} (5.1)

Where \(p\) denotes the point in \(P_t\), \(q\) denotes the point in \(P_s\), \((p_1, q_1)\ldots(p_n, q_n)\) are the correspondence pairs from all segmented patches which ranges from 1 to \(N\). \(N\) is the total number of correspondences that we have found between two point clouds. This has been achieved by computing correspondences between individual patches and afterwards concatenating them.

![Feature matching between segmented patches of \(P_s\) and \(P_t\).](image)

**Figure 5.1:** Feature matching between segmented patches of \(P_s\) and \(P_t\).

2. Calculating Centroids and De-meaning both Point Clouds:

In this step, we compute the centroids of both point clouds. A centroid represents the center of mass which will be used later to move both point clouds to the origin of the coordinate system.
Chapter 5. 3D Point Cloud Alignment for Pose Estimation

Figure 5.2: Feature matching between $P_s$ and $P_t$ at the same time.

$$\tilde{p} = \frac{1}{N} \sum_{i=1}^{N} p_i$$  \hspace{1cm} (5.2)

$$\tilde{q} = \frac{1}{N} \sum_{i=1}^{N} q_i$$  \hspace{1cm} (5.3)

Where $\tilde{p}$ is the centroid of $P_t$ and $\tilde{q}$ is the centroid of $P_s$.

$$\tilde{P} = \sum_{i=1}^{N} p_i - \tilde{p}$$  \hspace{1cm} (5.4)

$$\tilde{Q} = \sum_{i=1}^{N} q_i - \tilde{q}$$  \hspace{1cm} (5.5)

where $\tilde{P}$ and $\tilde{Q}$ are the resulting point sets after moving both point clouds to the origin of the coordinate system.

3. Calculating the Correlation Matrix:

The correlation matrix tells us how two point clouds associate with each other.

$$H = \tilde{Q} \ast \tilde{P}^T$$
4. Calculating the Transformation:

Since, $H$ is a square matrix of size $3 \times 3$, we can decompose it using a singular value decomposition $[2, 18]$.

$$H = U\Sigma V$$  \hspace{1cm} (5.6)

where $U$ and $V$ considers to be a rotation matrix, while $\Sigma$ is said to be a scaling factor. Thus, the expression $H\Sigma V$ can be viewed as a three geometrical transformation: Scaling, rotation and translation. Rotation $R$ and translation $t$ can be computed as:

$$R = V * U^T$$  \hspace{1cm} (5.7)

$$t = \tilde{p} - R * \tilde{q}$$  \hspace{1cm} (5.8)

After computing $R$ and $t$, the final transformation matrix $T$ can be computed as:

$$T = [Rt]$$  \hspace{1cm} (5.9)

5. Transforming $P_s$ and calculating the Mean Squared Error:

Once we have assumed $T$, we can transform $P_s$ as:

$$P_{s+1} = T * P_s$$  \hspace{1cm} (5.10)

where $P_{s+1}$ is a new transformation of $P_s$. Mean squared error of $P_{s+1}$ can be calculated as:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (P_{s+1_i} - p_{t_i})^2$$  \hspace{1cm} (5.11)

where $P_{s+1_i}$ is the $i^{th}$ point of $P_{s+1}$ and $p_{t_i}$ is the $i^{th}$ point of $p_t$.

There are three different criteria that can be used to terminate the GICP:

(a) The total number of iterations has reached the user specified number of iterations.

(b) The difference between two consecutive transformation is less than the user imposed value.
(c) The mean squared error is less than a user defined threshold.

![Figure 5.3: The pose estimation algorithm successfully estimated the pose and aligned both aligned point clouds. Yellow points are the points of $P_s$, while black ones are the points of $P_t$.](image)

We apply the above mentioned procedure to all subgraphs that we have validated in the previous chapter 4.3 and choose the best subgraph which has the least mean squared error. A parallel implementation of GICP is also possible. For instance, if we have four subgraphs, we can apply GICP on each of them and run on different cores. A comparison between parallel and sequential implementation of five different point clouds is shown in table 5.1.

<table>
<thead>
<tr>
<th>Point Clouds</th>
<th># of Graphs</th>
<th>Parallel implementation</th>
<th>Sequential implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point Cloud # 1</td>
<td>2</td>
<td>66</td>
<td>76</td>
</tr>
<tr>
<td>Point Cloud # 2</td>
<td>4</td>
<td>114</td>
<td>198</td>
</tr>
<tr>
<td>Point Cloud # 3</td>
<td>6</td>
<td>122</td>
<td>226</td>
</tr>
<tr>
<td>Point Cloud # 4</td>
<td>16</td>
<td>478</td>
<td>1203</td>
</tr>
<tr>
<td>Point Cloud # 5</td>
<td>40</td>
<td>274</td>
<td>908</td>
</tr>
</tbody>
</table>

**Table 5.1:** A comparison between sequential and parallel implementation: time taken in seconds. The difference between sequential and parallel implementation is visible for point cloud # 2 and 3. Point cloud # 1 has only 2 graphs, that is why the difference is only 10 seconds. Point cloud # 4 has only 16 graphs but it took more time than point cloud # 5 because the density is much higher.

### 5.1.1 Results and Comparison with the State of the Art Algorithms

In this section, we present and discuss the results of different pose estimation algorithms. First, we compare the results of our pose estimation algorithm which uses graph based
Next, we present and compare the results of our method with two state of the art algorithms which are also implemented in PCL as correspondence grouping (CG) [43] and alignment prerejective (AP) [6]. Throughout this section we refer our pose estimation algorithm as a graph based pose estimation algorithm (GBPE).

For our results, we have tested all algorithms on 47 objects ($P_t$) and for every object we have generated 4 different random transformed objects ($P_s$) using a hidden point removal algorithm as discussed above in section 2.1. The total size of our data set is 188 point clouds.

<table>
<thead>
<tr>
<th>No. of Point Clouds</th>
<th>Standard ICP</th>
<th>GBPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure</td>
<td>152</td>
<td>87</td>
</tr>
<tr>
<td>Success</td>
<td>36</td>
<td>101</td>
</tr>
<tr>
<td>Failure%</td>
<td>80.85%</td>
<td>46.27%</td>
</tr>
<tr>
<td>Success%</td>
<td>19.14%</td>
<td>53.72%</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison between SICP and GBPE in terms of accuracy.

In table 5.2 we have compared the results of SICP with GBPE: Out of 188 point clouds, SICP successfully estimated the poses of 36 point clouds, while GBPE was able to estimate the poses of 101 point clouds. There are several reasons of failure that we have identified:

1. In this thesis we are dealing with textured less point clouds as shown in figure 5.5. In textured less point clouds it is difficult to compute distinguishable features which leads to wrong correspondences. To minimize wrong correspondences, we have applied correspondence rejector method called the Random Sample Consensus (RSC). The RSC is implemented in the PCL which identifies inliers and rejects outliers. Even after applying the RSC, for some point clouds the algorithm was
not able to distinguish between inliers and outliers and failed to align both point clouds.

2. Another reason of failure is related to the symmetry of point clouds. In symmetric point clouds it is difficult to extract unique features. In our dataset, most point clouds are highly symmetric as depicted in figure 5.6 which also leads to the problem of wrong correspondences.

![Figure 5.5: Textured less point cloud.](image)

3. One more cause of failure which is only related to GBPE is choosing the right graph. The GBPE chooses one graph out of all the graphs which has the least mean squared error. Algorithm generates too many graphs when point clouds are symmetric as shown in figure 5.7. The reason for generating these many graphs is that all surfaces are flat after segmentation and surface curvature does not play any role. The GBPE, based only on the mean squared error, chose wrong graph 7 times out of 188 point clouds.

![Figure 5.6: Highly symmetric textured less point clouds.](image)
4. The GBPE is a graph based pose estimation algorithm and the goal is to estimate the poses by finding exact structure of $P_s$ graph in $P_t$ graph. Here, the quality of segmentation plays very important role. If algorithm segments $P_s$ and $P_t$ differently than finding the same structure in $P_t$ graph is impossible.

The reason why GBPE works better than SICP is that we have minimized the feature matching space by segmenting both point clouds into small patches. In GBPE the probability of having wrong correspondences is only between two segmented patches of $P_s$ and $P_t$, but in SICP we match both point clouds, i.e. $P_s$ and $P_t$ entirely at the same time.

![Figure 5.7: Left: A highly symmetric textured less complete point cloud. Right: A highly symmetric textured less point cloud with hidden points removed. Bottom Right: A highly symmetric textured less point cloud with hidden points removed depiction from another side.](image)

In table 5.3 we have compared the results of GBPE with two state of the art algorithms in terms of accuracy. The Alignment Prerejective (AP) is only able to align 12 point clouds out of 188 with an accuracy of 8.51%. We spoke to the authors of AP to enquire the reason for such a high failure. They answered that our dataset is highly symmetric and point clouds are textured less which is why the algorithm has failed to estimate the poses. On the other hand, the Correspondence Grouping (CG) performed better than SICP and AP. It successfully estimated the poses of 88 point clouds with an accuracy of 46.80%. In terms of accuracy, GBPE performed better than CG and AP.
In figures 5.8, 5.9, 5.10, 5.11 we have shown the average time of set of all transformed point clouds in bins. By average set we mean all four transformation of $P_t$. We calculated the average set as: For every $P_t$ we have four different transformed point clouds call $P_s$, we have averaged the time of four $P_s$ and plot it on the graph in bins. Using all the methods, dense point clouds such as 3.11 took more time to compute the right poses. The point cloud depicted in 3.11 took around 55 minutes using GBPE to find the right pose because it is symmetric and has 1,891,725 points in total. For the same point cloud, SICP took 32 minutes (but it failed to estimate the pose), AP took 15 minutes (but it also failed to estimate the pose), CG took around 11 minutes. In terms of time, CG 5.11 is the fastest algorithm and computed poses of 44 point clouds by taking only between 1 to 3 minutes. We have noticed, GBPE is a very time consuming algorithm when it comes to symmetric points clouds, 30 point clouds took around 9 minutes, 6 point clouds took around 10 to 18 minutes, 4 point clouds took between 20 to 30 minutes to compute poses. Using SICP, 45 point clouds took between 1 to 9 minutes. AP is also a time consuming algorithm, the first two bins of figure 5.10 contain around 30 point clouds.
Chapter 5. 3D Point Cloud Alignment for Pose Estimation

clouds and it took between 1 to 5 minutes. On an average GBPE took 10.5 minutes, AP took 5.5 minutes, SICP took 3.2 minutes while CG outperformed the other algorithms by taking just one minute on an average to compute poses.

Figure 5.9: Averaged time taken by set of point clouds to calculate poses using GBPE.

Figure 5.10: Averaged time taken by set of point clouds to calculate poses using Alignment Prejective (AP).
All measurements were recorded on an Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz processor.

5.2 Discussion

In this chapter we proposed the graph based iterative closest point algorithm (GICP) for point cloud alignment. The assumption that we have taken earlier i.e. every $P_s$ is a subset of $P_t$ is hold true. We have shown that in terms of accuracy, the graph based pose estimation algorithm (GBPE) performed better than standard iterative closest point (SICP). The GBPE also performed better than correspondence grouping (CG) and alignment prerejective (AP) which are state of the art methods for pose estimation.

As far as limitations are concerned, the GBPE does not perform well when we have symmetric point clouds or point clouds where all the surfaces are flat or curved. Another limitation that we have identified is that the GBPE only works well when both $P_s$ and $P_t$ segmented are properly because it looks for the same structure within the $P_t$ graph.

In the next chapter, we will discuss about how we can improve our pose estimation algorithm, what are some practical implementation of it, how we can modify our algorithm to improve the accuracy and reduce the total computational time.
Chapter 6

Conclusion and Future Work

Throughout this work, we had one goal in mind to develop the pose estimation algorithm for 3D rigid point clouds that should be robust, computationally efficient and parallelizable. We used surface graphs to estimate the poses. The primary goal of using surface graphs was to find similarities only between relevant areas of point clouds. We called our algorithm the graph based pose estimation algorithm (GBPE). Indeed, there were still many aspects of pose estimation that were untouched and our future research will continue in those directions.

6.1 Summary of Contribution

We began in Chapter 2 and generated the dataset. We applied hidden point removal algorithm to 47 different objects and made our pose estimation scenario acted like a real world.

In chapter 3 we presented two segmentation algorithms. Both algorithms used VCCS [26] as a preprocessing step before applying actual segmentation algorithm. In particular, we showed how VCCS can be used to minimize the total segmentation time by preserving the actual accuracy of the algorithm.

Next, in chapter 4 we extracted features on every segmented patch: SHOT and surface curvature. We then generated surface graphs and showed how graphs can be used to minimize the chances of matching wrong correspondences. We also showed surface
curvatures can be used to minimize the total features matching time of the algorithm. Moreover, we implemented the subgraph matching algorithm parallely.

Finally, in chapter 5 we applied the graph based iterative closest point (GICP) algorithm to the remaining subgraphs and chose only one subgraph based on mean squared error. We compared the result of our pose estimation algorithm (GBPE) with two state of the art algorithms. GBPE worked better in terms of accuracy and also performed remarkably well when point clouds were textured less.

6.2 Limitations and Direction of Future Work

The main limitation of our work is that it generates too many $P_s$ graphs and applying GICP on each and every one of them is a time consuming activity. Surface curvatures has been used to minimize the total number of graphs, similarly object contours can also be used to reduce the total number of graphs. The segmented patches of $P_s$ and $P_t$ that are depicted in figure 6.1 have the same shape and are flat. Due to surface curvature, algorithm finds similarity only between same curvature surfaces. Introducing object contours will allow algorithm to match segmented patches where the similarity in terms of shape is high. Another way to make correspondences strong is to use geometrically aligned feature matching technique as discussed in [10].

![Figure 6.1: Left: Segmented patches of $P_s$ and $P_t$ without contours. Right: Segmented patches of $P_s$ and $P_t$ with contours.](image)

Signature of Histogram and OrienTation (SHOT) which is a local feature with an ability to capture information around point $p$ using $k$ nearest neighbor has been used in our work. Local features only considers $k$ number of points that are within the small radius around point $p$. These types of features are not very helpful when similarity between point clouds are high. One good way to avoid this problem and to capture more information is to increase the radius size. Another way to make correspondences strong without increasing the size of the radius is to use all the points that are within one supervoxel cluster and average them as depicted in figure 6.2. We can call this approach Averaged Voxelized SHOT Feature. There are two main advantages of this
approach: First, it makes local feature a good representation by combining all the information that is within one supervoxel clusters. Second, instead of matching all the points, now algorithm has fewer points to match which is equal to the total number of supervoxel clusters that are there on every segmented patch. This should reduce the total computation time drastically.

![Figure 6.2: Depiction of averaged voxelized SHOT feature matching [25].](image)

GICP can also be extended efficiently to determine hidden points of $P_s$. After estimating the pose, algorithm just have to apply hidden point determination algorithm similar to 2.1 only on those nodes of the graph that are matched. Remaining surfaces of the full graph will be considered as it is because they are not part of $P_s$. As shown figure 6.3 nodes that are marked in red are visible areas of $P_s$, while some parts of the green nodes are hidden so an efficient way is to apply hidden point determination algorithm only on green nodes.

![Figure 6.3: Graph of $P_t$, green nodes showing the matched surfaces while red ones are unmatched.](image)
Future work can also be focused on fitting two 3D point clouds with each other. Poses that we have calculated and graph based approach can be combined to fit two 3D models. For instance, box shape point cloud can be used as a starting point of this work. One box can be placed inside another box if the dimensions are satisfied. Object contours, surface curvatures, size of the segmented patches are some features that might play an important role in fitting two point clouds. This type of algorithm has application which ranges from packaging to automation industry. For example in packing industry role of a human is to put items in packages, if we are able to automate this, then robots would be performing these types of tasks for us in future.
Bibliography


53


Chapter 6. Conclusion and Future Work


