Incremental Place Recognition in 3D Point Clouds

Master Thesis

Author(s):
Gollub, Mattia

Publication date:
2017

Permanent link:
https://doi.org/10.3929/ethz-b-000202826

Rights / license:
In Copyright - Non-Commercial Use Permitted
Master Thesis

Incremental Place Recognition in 3D Point Clouds

Spring Term 2017

Supervised by:
Renaud Dubé
Hannes Sommer
Igor Gilitschenski

Author:
Mattia G. Gollub
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Incremental Place Recognition in 3D Point Clouds

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Gollub

First name(s):

Mattia

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 01.09.2017

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.
# Contents

**Preface**  
v  
**Abstract**  
vii  

1 Introduction  
1.1 Place Recognition  
1.2 Segment–Based Place Recognition  
1.2.1 Local Cloud Update  
1.2.2 Segmentation  
1.2.3 Description  
1.2.4 Matching  
1.2.5 Recognition  
1.3 Incremental Place Recognition  

2 Related Work  
2.1 Place Recognition  
2.2 Downsampling  
2.3 Normal Estimation  
2.4 Clustering  
2.5 Recognition  

3 Incremental Segment Matching  
3.1 Dynamic Voxel Grid  
3.1.1 Storage  
3.1.2 Voxel Indexing  
3.1.3 Insertion and Removal  
3.1.4 Affine Transformation  
3.1.5 Volume Coverage and Numerical Precision  
3.2 Incremental Normals Estimation  
3.2.1 Storage of the Normals  
3.2.2 Incremental Updates  
3.2.3 Affine Transformation  
3.3 Incremental Region Growing Segmentation  
3.3.1 Partial Clusters  
3.3.2 Overview  
3.3.3 Region Growing  
3.3.4 Linking Partial Clusters  
3.3.5 Growing Policies  
3.3.6 Segment Tracking  
3.4 Graph–Based Incremental Recognition  
3.4.1 Partitioned Consistency Graph Construction  
3.4.2 Incremental Consistency Graph Construction
Preface

Six months ago I started this project armed with lot of curiosity and very little understanding of robotics. After all, why should one feel the need to already know something, if there is the opportunity to learn it?

The project grew impressively, and so did my knowledge in the field. I am amazed by how much it is possible to learn and create in so little time. Of course this would have never been possible without the help of my supervisors. Hannes, Igor and in particular Renaud: for the passion you put in this project, thank you! I am also grateful to Prof. Siegwart for the opportunity of writing my thesis at the Autonomous Systems Lab (ASL).

I am very thankful to my parents, for believing in me during all this years of studies and to Ash for the constant support and for introducing me to proper data visualization.
Abstract

Light Detection And Ranging (LiDAR) sensors are often employed in robotics, as they can produce dense reconstructions of the environment without being dependent on the lighting conditions and the presence of visual features. However, performing place recognition in the generated 3D point clouds is computationally very expensive.

This thesis proposes a novel incremental approach to segment based place recognition. Input points are accumulated, and the point normals are selectively updated only when affected by the insertion. We introduce a generic incremental segmentation strategy based on region growing, which tracks the evolution of the single segments, enabling a new method for fast recognition based on partitioning and caching of geometric consistencies.

We show that the speedup obtained by the incremental method enables global place recognition at 10Hz in urban driving scenarios and lower dead reckoning times. The higher number of place recognitions leads to more accurate trajectory estimation and to more loop closures, compensating drift during mapping. The reduced computational cost of the method presented makes it suitable for applications where real-time localization is required and enables its usage on cheaper, low-energy systems.
Chapter 1

Introduction

This thesis presents a novel incremental approach to segment based place recognition in 3D point clouds. This chapter introduces the problem of place recognition and an approach based on segment extraction and matching which is used in our incremental method. Chapter 2 presents an overview of related work in the field of place recognition. We present our incremental method in chapter 3 and evaluate its performance and quality in chapter 4. In conclusion, chapter 5 briefly summarizes contributions, results and future work.

1.1 Place Recognition

Perception capabilities are a key requirement for robots in order to perform high-level tasks like navigation and interaction. For this reason, it is often important for a robot to be able to maintain and interpret a digital map of the surrounding environment.

In the case of Simultaneous Localization And Mapping (SLAM) a robot explores a scenario, localizing in already visited regions and mapping unknown environments at the same time. Although odometry sensors and local registration techniques can provide an estimate of the trajectory, the accumulation of small local errors leads to considerable drift over time. This effect becomes increasingly problematic in large-scale maps. Moreover, multi-robot applications like robot assisted disaster response require multiple robots navigating independently to estimate their relative poses without a unified target representation.

This motivates the need for loop-closure detection capabilities in robust SLAM systems: Detecting loops allows to place additional constraints on the trajectory, compensating the drift. In order to obtain reliable position information, localization must not be the result of a chain of local estimations, rather a global recognition. Given the lower descriptive power and the impossibility of estimating Six Degrees of Freedom (6DoF) poses in 2D maps, 3D point clouds are amongst the most suitable representations for this kind of tasks, as they provide a complete, dense representation of the environment. However, 3D representations and global place recognitions currently have high computational demands. In order to enable usage in large-scale scenarios and on robots with constrained resources, new solutions with lower computational complexity are required.
1.2 Segment–Based Place Recognition

Segment matching [1] is a new method for performing global place recognition and loop closure detection in 3D point clouds. It has proven to be working efficiently both in urban driving and indoor environments and is the starting point of the work done in this thesis. An overview of the pipeline is shown in fig. 1.1 while fig. 1.2 shows a visualization of the main steps. 3D points extracted from the LiDAR data are accumulated in a local cloud, from which segments are extracted. Compact descriptors are then computed for the extracted segments and matched with the descriptors of the segments of the target map. The correspondences found are used for recognizing the local map in the target map. The main stages of the pipeline are explained in details in the following paragraphs.

1.2.1 Local Cloud Update

An external SLAM application transforms the input LiDAR scans in 3D point clouds aligned with the current coordinate system of the local map and provides an estimate of the pose of the vehicle. First the local cloud is updated removing points that are too far away from the robot (e.g. 60 m), then the new points are added to the point cloud. In order to reduce noise and the computational load on the successive stages, the local cloud is downsampled to a desired resolution (e.g. 0.1 m) using voxel filtering (see section 2.2), ignoring voxels that do not contain enough points.

1.2.2 Segmentation

Contrastingly to other recognition techniques, matching is performed on segments instead of keypoints. The downsampled local cloud is subdivided in clusters, which
1.2. Segment–Based Place Recognition

Figure 1.2: Visualization of the main steps of segment–based place recognition. (a) A LiDAR sensor provides 3D laser scans of the environment. (b) The scans are filtered and accumulated in a downsampled local cloud. (c) Segments are extracted from the local cloud. (d) Using the segment descriptors, correspondences (green lines) are found between the local and target maps, possibly leading to a localization.

are then filtered according to their size. Clusters with too few points are ignored since they would not exhibit representative features. Clusters with too many points cannot be observed entirely and are discarded as well. The remaining clusters are marked as segments.

1.2.3 Description

For each extracted segment $S_i$, a descriptor is computed. In general a descriptor is a vector $f_i$ of values that encodes features of a segment. Examples of descriptors are eigenvalues–based measures [2] like planarity and scattering, but also more complex features like ensemble of shapes histograms [3]. In mapping applications, the target map is updated adding the descriptors and centroids of the newly observed segments. It is important to note that, with the technique presented in section 1.2.2, it is not possible to robustly associate segments representing the same object in multiple scans. Thus, an extra step is required to remove old segments whose centroids are too close to the centroids of the new segments.

1.2.4 Matching

Using the computed features, one can find matches between segments in the local and target maps. This is done using Nearest Neighbors (NN) searches. A $k$-d tree is built at initialization time or after each update of the target map, containing the descriptors $f_j$ of all segments in the target map. For each segment $S_i$ in the local map with descriptor $f_i$, the candidate matching segments are all segments $S_j$ such that their descriptors $\|f_i - f_j\| \leq \theta_f$, where $\theta_f$ is the threshold for the
maximum Euclidean distance such that two descriptors are considered equivalent. This is easily achieved by performing radius searches in the $k$-d tree using $\theta_f$ as the maximum radius.

### 1.2.5 Recognition

The returned matches generally contain a high fraction of false positives, which makes it impossible to directly estimate the transform between local and target maps. In order to identify the true matches it is necessary to find a set of geometrically consistent matches, i.e. a set of candidate matches whose distances between segments in the local and target maps are consistent. If the set contains a sufficient number of matches, the recognition is considered successful and the rigid transformation matrix is estimated by least-squares minimization and returned to the SLAM system. The application is then able to update the trajectory of the robot and, in case of loop closure applications, add a loop constraint to the pose graph and optimize the position of the segments in the target map.

### 1.3 Incremental Place Recognition

Although place recognition through segment matching is shown to be faster than keypoint approaches [1], it still has a high computational cost. This is caused mainly by the high number of points in the local cloud and by the potentially high number of candidate correspondences between the local and target maps. Since real-time performance increases the probability of detecting loop closures, the current performance limitations prevent the system from running optimally on hardware with constrained resources and in larger scenarios.

Accumulating data in the local cloud increases matching performance and adds robustness to the point of view. However we observe that, when the local cloud is updated (by inserting new points or by updating the position of the robot), the changes in the segmented local map and in the correspondences are minimal. Thus, the place recognition method presented in section 1.2 can be optimized so that information extracted in previous localizations can be reused, reducing the processing time.

In this work we present a place recognition method based on incremental segment matching. We propose novel incremental algorithms for the local cloud update, normals estimation, segmentation and recognition steps, significantly reducing the runtime of the entire pipeline. We evaluate the performance of the proposed approach on emergency response and urban driving scenarios and show remarkable improvements in the localization rate. The results obtained make the method suitable for applications where real-time localization is required and enable its usage on cheaper, low-energy systems.
Chapter 2

Related Work

This chapter presents an overview of the state of the art techniques related to the proposed approach and place recognition in general. Additionally, we introduce several alternative approaches that have been considered in order to evaluate which methods can benefit the most from an incremental approach.

2.1 Place Recognition

Place recognition in 3D point clouds is still an open problem in the field of robotics and several types of approaches have been proposed. Bosse and Zlot [4] propose to extract and describe keypoints from the input point clouds. Using the computed features, each keypoint votes for its most likely matching keypoints. Place recognitions are then found by applying a minimum threshold on the number of votes.

Instead of identifying local features, place recognition can be performed using global descriptors like, for example, histograms of point heights of the local clouds [5]. In this case place recognition is performed by comparing the histograms using the Wasserstein metric.

Another promising strategy consists of extracting and matching base elements from the local and target clouds. Such elements can be, for example, planes or objects. In this thesis we choose to use generic segments [1] as presented in section 1.2, since they do not rely on any assumption about the type of elements observable in the environment.

2.2 Downsampling

In the context of 3D point clouds, downsampling is the process of reducing the number of points in a dataset. Voxel filtering can be used for downsampling a point cloud to a desired resolution. However, explicit storage of the voxels requires an enormous amount of memory. The VoxelGrid filter implemented in the Point Cloud Library [PC] [6] overcomes this problem for sparse datasets by storing only occupied voxels. First, the voxel index of each voxel is computed. Points are then sorted by voxel index so that they can be grouped by voxel. Finally the centroid of each group of points belonging to the same voxel is computed and added to
the downsampled cloud. Because of the sorting step, construction and update of a
downsampling point cloud has an asymptotic runtime of $O(n \log (n))$.

An alternative data structure based on voxel hashing is presented by Nießner et al. [7]. Voxels are stored in small regular grids (e.g. $8 \times 8 \times 8$ voxels) called voxel blocks, which are only allocated for regions which actually contain voxels. A hash table is used to store the mapping from voxel index to the voxel blocks containing the voxel, allowing insertion, deletion and access by index in constant time.

Octomap [8] uses an octree as data structure for the voxels, which are stored in the leafs. Instead of their centroids, voxels contain occupancy information. Occupancy is determined probabilistically and is updated incorporating the probability contributed by new sensor measurements with the prior probability stored in the voxel.

Other downsampling approaches that do not rely on a regular grid include the supervoxels method [9]. This enforces color and euclidean distance constraints in the resulting regions and is proposed as an over–segmentation method to be used for reducing the computational load on higher level segmentation algorithms. Since our system is designed to use geometric information only, the resulting regions would be identical to the euclidean clusters described in section 2.4.

2.3 Normal Estimation

Least squares is a common method for estimating point normals in 3D point clouds and is solved by eigenvalue analysis [10]. The first step for estimating the normal of a point $p$ is the identification of its neighborhood $N(p)$ (which includes $p$ itself). This can be either the result of a $k$–NN search or of a radius search. For normal estimation applications, the choice of the search strategy depends mainly on the distribution of the points in the point clouds. Once the neighborhood is found, its covariance matrix

$$M = E[(P - E[P]) (P - E[P])^\top]$$  \hspace{1cm} (2.1)

is computed, where $P$ is the random vector containing the neighborhood $N(p)$. The estimate of the normal is equal to the normalized eigenvector of $M$ corresponding to the smallest eigenvalue. Intuitively, this is the vector perpendicular to the plane with highest variance of the points. In order to get consistent normals, the result may need to be flipped towards the viewing point. Additionally, this method allows to compute the curvature $\sigma$ of $p$ as:

$$\sigma = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}$$  \hspace{1cm} (2.2)

where $\lambda_0 < \lambda_1 < \lambda_2$ are the eigenvalues of $M$. For segmentation applications, Rabbani et al. [11], show that using the plane fitting residuals as indicator for curvature produces better segmentations in pathological cases like in presence of toroids.

The major problem of the least square approach is that, because of the uniform region of neighbors considered, sharp edges are missed, producing smooth normals. The Robust Randomized Hough Transform (RRHT) method [12] solves this problem by sampling multiple possible planes from $N(p)$ and choosing the one with the highest probability to be the correct one. Although this method is very robust, it is also an order of magnitude slower than the least squares approach, making it unsuitable for real–time applications.
When estimating normals from 2D images, box-filtering based variants of the least square approach \cite{13} achieve significant speedup factors over the original method. Since these methods are purely based on 2D data, it is not possible to integrate data from multiple scans.

Poppinga et al. \cite{14} show an incremental approach for estimating plane normals that updates the components of the covariance matrix $M$ as new points are added to the plane. In this work we present a more efficient formulation and we use it for estimating the point normals incrementally. Additionally we show a method for applying affine transformations to $M$ without the need to store and transform the original points.

2.4 Clustering

Clustering is a common problem in data processing with a wide range of applications. For this reason, a great number of methods have been developed to address the challenges posed by different kinds of datasets. Selecting the best method for a specific application is usually not trivial and depends on factors like dimensionality of the data, performance requirements and the desired properties of a cluster. This section presents a short review of relevant clustering techniques and related work.

Euclidean clustering \cite{15} is one of the simplest, but still effective, clustering strategies that has been employed for the segmentation of 3D point clouds. The method is based on region growing, in order to perform optimally, it requires environments where objects are clearly separated by empty space. Sequentially, each point that is not assigned to a cluster yet is used as seed for a new region. The region then expands to include points whose euclidean distance from the seed is less than a given threshold. The reached points are then used as seeds, and the growing process is iterated until no more points can be added to the region. New regions are grown until all points are assigned to a cluster.

Euclidean clustering requires the ground (if any) to be removed from the scene. If this is not done, all non–floating objects would be connected to the ground and thus to each other, resulting in a single cluster including all the points of the scene. If the ground can be assumed to be flat and the position of the sensor is known, then points belonging to the ground can be removed by applying a threshold on the axis perpendicular to the ground plane. More advanced methods achieve better reliability on more complex terrains as well. Bogoslavskyi and Stachniss \cite{16} perform ground removal by extracting the angle of the surface at each point from the 2.5D range image. Points whose angles are similar within a threshold are then assigned to the ground.

The effectiveness of euclidean clustering decreases in environments where objects are not clearly separated by empty spaces, like for example indoor settings or environments with few objects. For such cases, smoothness constraints \cite{11} can be added on top of the region growing process performed by euclidean clustering so that under–segmentation is reduced. With the proposed additional constraints, a region can grow to include a new point only if the angle between the normals of the new point and of the seed falls below a given threshold. Points are used as seeds only if an additional threshold test on the point curvature is satisfied. Depending on the thresholds, the resulting segments will be either planes or smooth curved surfaces. Similarly, Density–Based Spatial Clustering of Applications with Noise (DBSCAN) \cite{17} and successive improvements \cite{18} add density constraints to the criteria for region growing.
Chapter 2. Related Work

Clustering methods based on nearest neighbors perform \( \text{NN} \) searches in order to find candidate points for growing. In 3D point clouds, this is known to be a computationally expensive problem. Generally, the best solution is to use \( k \)-d trees \( [19] \) which has asymptotic complexities of \( O(n \log(n)) \) for construction and \( O(\log(n)) \) for queries. Alternative structures like hierarchical voxels \( [20] \) offer faster query times, but at the cost of a significantly higher construction cost.

Speed is a primary concern in robotics applications, and it is often desired to have real-time performance and limited resource usage. For this reason, several methods for increasing the performance of clustering have been proposed. One possible improvement is to perform segmentation on the range image directly \( [16] \). This is significantly faster, because the 3D point cloud is not computed explicitly. Moreover the structure of the laser scan implicitly provides neighborhood informations. Unfortunately, this method does not offer the possibility to integrate data from multiple scans. Contrastingly, our approach is able to perform incremental segmentation on accumulated 3D point clouds. It takes advantage of the fact that only a small fraction of the local cloud changes when new points are inserted to reuse the result of previous segmentations. Similarly, Whelan et al. \( [21] \) show a method for performing segmentation based on region growing in an incremental fashion. During the segmentation process, new points are either used for creating a new plane or merged with existing planes found in previous steps. The method does not address non-planar segments and it has been tested with cameras only, while it is not clear if the robustness is maintained with sparse LiDAR data. In this thesis we present a general incremental segmentation method that does not enforce assumptions on the type of segments and performs robustly with LiDAR data.

2.5 Recognition

In general, recognition is the process of identifying a model in a scene and determining its pose. This is usually done by extracting basis elements from the maps and then finding correspondences between the object and the scene. Examples of such elements are keypoints \( [4] \), objects \( [22] \), shapes \( [23] \) or segments \( [1] \) (see section 1.2). For the scope of this work, both the object and the scene are described as sets of segments, which contain a feature descriptor and a centroid.

Depending on the scene and on the matching method used, the set of correspondences generally contains a high fraction of false positives. For this reason, registration approaches that are sensitive to outliers like Iterative Closest Point (ICP) \( [24] \) and energy minimization methods \( [25] \) cannot be used.

Geometric consistency grouping \( [26] \) is a technique that proved to be robust also in noisy conditions. Recognition is performed by clustering the correspondences in geometrically consistent sets. Two correspondences \( c_i \) and \( c_j \) are called pairwise geometrically consistent if the difference of the Euclidean distances between the keypoints in the local map and in the target map is less than a threshold \( \epsilon \), i.e. if

\[
|d_l(c_i, c_j) - d_t(c_i, c_j)| \leq \epsilon \tag{2.3}
\]

where \( d_l(c_i, c_j) \) and \( d_t(c_i, c_j) \) are the distances between centroids in the local map and in the target map respectively.

This method is implemented in the recognition module of the \( \text{PCL} \) \( [6] \), where geometric consistencies are determined using a brute-force approach in which all possible correspondence pairs are checked for consistencies. This has an asymptotic complexity of \( O(n^3) \), where \( n \) is the number of correspondences.
This approach is well suited for scenarios with a low amount of candidates and was employed in the original segment matching pipeline [1]. However, it does not scale well to cases with a large number of candidates and outliers, eg. when doing real-time place recognition in a large target map. Moreover, as shown in section 4.3.3, the implemented method only computes an approximation of the optimal recognition.

Strategies for efficiently reducing the number of correspondence pairs have been proposed for stereo images and image retrieval. Ayache and Faverjon [27] describe a partitioning scheme for efficiently finding neighbor segments in stereo images. The SCRAMSAC method [28] performs RANSAC only on spatially consistent correspondences, i.e. correspondences that have a minimum fraction of matching neighbor features in both images. Both methods rely on assumptions about the disparity between images, thus their accuracy is influenced by the presence of high disparity and strong variation in viewing angles.

In this work we present a method for performing place recognition efficiently through partitioning and caching, reducing the asymptotic complexity of the original geometric consistency grouping method [26].
Chapter 3

Incremental Segment Matching

This chapter presents the architecture and the components of our incremental segment matching system. Pursuing the goal of reducing the computational requirements of place recognition, the idea of the method is to reuse information that overlap over multiple localization steps. Since the number of points added by each scan is only a small fraction of the total number of points in the point cloud, we expect the incremental system to significantly accelerate the place recognition process.

After an analysis of the runtime of the original SegMatch library [1], we identify four main components that have a significant influence on the runtime of the application and would benefit of an incremental approach: downsampling, normal estimation, segmentation and recognition. Performing the segmentation incrementally also enables segment tracking, which ensures that multiple views of the same segment maintain the same segment ID.

Figure 3.1 shows the high level architecture of the incremental segment matching pipeline. Part of it is identical to the original SegMatch design presented in section 1.2, but there are two important differences. First, the above mentioned components are modified to use incremental algorithms. Moreover the incremental downsampling step also provides the indices of the new points created by voxels downsampling, so that the normals estimator can operate incrementally as well by only recomputing normals affected by the new points.

The following sections will present in detail the individual methods used for achieving incremental segment matching.

3.1 Dynamic Voxel Grid

As introduced in section 2.2, the common way to perform downsampling in 3D point clouds is by voxel filtering. In this work, instead of performing a batch voxel filtering of the entire local cloud at each step, we aim at adding and updating only voxels that are affected by the new scanned points. The data structure used must support dynamic insertion and removal of points and must be able to provide information about which voxels have been affected by an update. This is not possible with the PCL VoxelGrid implementation [6], while alternative data structures like voxel
octrees are designed for traversal in ray tracing applications and are not suitable for our case. For our application we implement a Dynamic Voxel Grid (DVG), an efficient data structure supporting the functionalities mentioned above.

### 3.1.1 Storage

The data structure uses plain vectors as storage. Despite the usage of vector-based voxel grids requires a sorting step (see section 2.2) during construction, preliminary benchmarks showed that for our use cases this is faster than asymptotically more efficient hashmap-based structures. This is justified by the constant overhead for hashmap accesses which is higher than for vectors. Moreover, as shown later, in an incremental approach the impact of the sorting operation can be reduced significantly.

Voxels are stored in a vector in increasing voxel index order. For each voxel the DVG stores its centroid, the number of points used for the downsampling and the index of the voxel.

In order to reduce noise, a voxel is considered active if it contains at least a desired amount of points. For the successive stages of the segment matching pipeline, only active voxels are used. In order to improve cache performance of the normals estimator and of the segmenter, the centroids of the active voxels are stored in a separate active voxels vector.

### 3.1.2 Voxel Indexing

Virtually, the voxel grid is a regular grid of size $l \times w \times h$ voxels, where each voxel has a unique index in the interval $[0, l \cdot w \cdot h - 1]$. The grid has a center $o$, a fixed resolution $r$ and a transformation $T_b$ which is initially set to the identity matrix.
Indexes are stored in in $b$-bits unsigned integers, for computational efficiency we require the sizes of the grid to be powers of 2:

$$l = 2^{l_{\text{bits}}}, \quad w = 2^{w_{\text{bits}}}, \quad h = 2^{h_{\text{bits}}}, \quad l_{\text{bits}} + w_{\text{bits}} + h_{\text{bits}} \leq b \quad (3.1)$$

The voxel index $I(p)$ of a point $p$ is computed as:

$$I(p) = \left\lfloor \frac{c.x + \frac{l}{2}}{2} \right\rfloor + \left\lfloor \frac{c.y + \frac{w}{2}}{2} \right\rfloor \ll (l_{\text{bits}} + w_{\text{bits}})$$

where $\ll$ is the bitwise left shift operator and $c$ corresponds to the point $p$ transformed according to the parameters of the grid:

$$c = \frac{T_g^{-1}p - o}{r} \quad (3.3)$$

### 3.1.3 Insertion and Removal

When new points are inserted, the DVG first computes their voxel indices. Then, only the new points are sorted in increasing voxel index order, since the existing voxels are already sorted. This is an important optimization, since sorting has asymptotic complexity $O(n \log(n))$, and a batch voxelization needs to sort all the points at every insertion. The remaining operations can be performed in $O(n)$.

Once the new points are sorted, they can be added to the existing voxels with a merge operation. When $m$ points $p_i$ are inserted in a voxel with centroid $c$ downsampled from $n$ points, its properties are updated as follows:

$$c \leftarrow \left( n \cdot c + \sum_{i=1}^{m} p_i \right) \cdot \frac{1}{n + m}, \quad n \leftarrow n + m \quad (3.4)$$

In addition, the indices (relative to the active voxels vector) of the voxels that turned active after the insertion are collected, so that the normal estimator and the segmenter can operate incrementally on the new voxels only.

When the robot moves, the local cloud is updated by removing points that fall outside the radius of the local map. Only removal of entire voxels is allowed. Voxels are removed by deleting them from the containing vector. Vectors are contiguous containers, thus in order to minimize the number of copies triggered by the removal, voxels are removed in a single linear batch operation.

### 3.1.4 Affine Transformation

When a loop closure is detected, the SLAM application re-evaluates the trajectory of the robot and provides a new estimation of its pose. Consequently, the local cloud must be updated with an affine transform $T_i$.

The transformation $T_i$ is applied to the centroid of each voxel. Moreover, in order for new points to be assigned to the correct voxel, the transformation of the DVG is updated as $T_g \leftarrow T_i \cdot T_g$. 

3.1.5 Volume Coverage and Numerical Precision

Assuming a resolution $r$ measured in meters and a given number of bits of the index dedicated to each dimension, the volume covered by the DVG has size (in meters):

$$r \cdot 2^{bits} \times r \cdot 2^{bits} \times r \cdot 2^{bits}$$ (3.5)

Using 64–bits integers and 21 bits per dimension at a resolution of 0.1 m the DVG can cover a cube with edges of size 209.7 km, which is currently sufficient for most realistic SLAM use cases. In case of a hypothetic world–scale map or higher resolutions, 128–bits indexes can be used, which can be emulated on most processors with minimal loss of performance. Using 42 bits per dimension it is virtually possible to voxelize the solar system at 1 m resolution.

It is important that the mantissa of the floating point data type used has at least as many bits as the maximum number of bits used for a single dimension, otherwise indexing operations will yield incorrect results because of rounding errors. If a single dimension uses more than 23 bits of the index, double–precision floating points must be used.

3.2 Incremental Normals Estimation

Normals are estimated using the least squares method described in section 2.3, to which two major optimizations are applied to make the process incremental: only normals affected by new scanned points are updated, and the covariance matrix $M$ is computed incrementally.

3.2.1 Storage of the Normals

By expanding the factors in eq. (2.1) we get an alternative formulation of the covariance matrix of a point $p_i$:


$$= \frac{1}{|\mathcal{N}(p_i)|} \cdot \sum_{p \in \mathcal{N}(p_i)} pp^T - \frac{1}{|\mathcal{N}(p_i)|^2} \cdot \left( \sum_{p \in \mathcal{N}(p_i)} p \right) \left( \sum_{p \in \mathcal{N}(p_i)} p \right)^T$$ (3.6)

The advantage of the above formulation over eq. (2.1) is that it can be computed incrementally, without the need of keeping track of the neighborhoods $\mathcal{N}(p_i)$ of each point.

For each point $p_i$ in the filtered local cloud, five values are stored:

- The normal of each point $N_i$.
- The curvature of each point $c_i$.
- A $3 \times 3$ matrix $A_i$, the accumulator for the term $E[PP^T]$.
- A $3 \times 1$ vector $b_i$, the accumulator for the mean $E[P]$.
- The number of accumulated points $n_i$.

Initially, the normals and the curvatures are set to NaN, while the other three values are set to zero. Using the accumulators, eq. (3.6) can be rewritten as:

$$M_i = \frac{1}{n_i} \cdot A_i - \frac{1}{n_i^2} \cdot b_i b_i^T$$ (3.7)
3.2.2 Incremental Updates

Let \( \{ p_i \} \) be the filtered point cloud and \( U \) be the set of indices of the new points. The update of the normals is divided in two steps: contributions scattering and gathering and normals computation.

In the first step, the accumulators are updated with the contributions of the new points. This step requires neighborhood to be a symmetric relation of two points, i.e.

\[
is\text{Neighbor}(p_1, p_2) \iff is\text{Neighbor}(p_2, p_1), \quad \forall p_1, p_2 (3.8)
\]

This is not true in case of \( k\text{-NN} \) so it is necessary to sacrifice the adaptability of this kind of searches and use fixed radius searches.

Once the neighborhoods of the new points are identified, the contributions of the new points are scattered to their neighbors. At the same time, contributions of old points to the normals of the new points are gathered. For each \( i \in U \) and for each \( p_j \in N(p_i) \):

- Contributions scattering:
  \[
  A_j \leftarrow A_j + p_ip_i^T, \quad b_j \leftarrow b_j + p_i, \quad n_j \leftarrow n_j + 1 \quad (3.9)
  \]

- Contributions gathering. If \( j \) is an old point, i.e. if \( j \notin U \):
  \[
  A_i \leftarrow A_i + p_jp_j^T, \quad b_i \leftarrow b_i + p_j, \quad n_i \leftarrow n_i + 1 \quad (3.10)
  \]

Finally, the normals of the points whose accumulators have been updated are recomputed. The covariance matrices are computed as shown in eq. (3.7), while the normals and curvatures are computed from the covariance matrices as described in section 2.3.

3.2.3 Affine Transformation

As seen in section 3.1.4, an affine transformation is applied to the local cloud when the trajectory of the robot is re-estimated. One could argue that, for the scope of segmentation, it is not necessary to transform the normals, since only relative angle between normals are considered. In an incremental setting however, omitting to update the accumulators would cause inconsistencies in the accumulated contributions, as the covariance matrix would be computed from points belonging to different reference frames.

We consider a rigid transformation \( T \) expressed as a rotation matrix \( R \) and a translation vector \( t \). The transformation is applied to all points \( p_i \) belonging to the local cloud as \( p_i \leftarrow Rp_i + t \). Let \( Q = RP + t \) be the random vector of the transformed neighborhood of a point. Then the expected values in eq. (3.6) can be expressed as:

\[
\mathbb{E}[QQ^T] = \mathbb{E}[(RP + t)(RP + t)^T]
= \mathbb{E}[RP^TR^T + RP^T + tP^TR^T + tt^T]
= R \mathbb{E}[P^TR^T + R \mathbb{E}[P]t^T + t \mathbb{E}[P]^TR^T + tt^T]
\]

\[
\mathbb{E}[Q] = \mathbb{E}[RP + t]
= t + R \mathbb{E}[P] \quad (3.11)
\]
Thus, considering the normalization factors, the accumulators are updated as:

\[
A_i \leftarrow RA_i R^T + Rb_i t^T + tb_i^T R_t + n_i \cdot tt^T
\]

\[
b_i \leftarrow n_i \cdot t + Rb_i
\]

while the normals are updated as:

\[
N_i \leftarrow RN_i + t
\]

The point curvatures are not affected by the transformation.

### 3.3 Incremental Region Growing Segmentation

Similarly to the previous sections we argue that, given the low fraction of points added at each step, the segmentation stage can be accelerated by using an incremental approach. As shown in section 2.4, many segmentation algorithms (e.g., euclidean clustering, region growing segmentation with smoothness constraints and DBSCAN) rely on a common region growing logic. The difference between the methods lies mainly in the growing policies. In this section we treat these methods under a common framework and present a general algorithm for incremental segmentation of 3D point clouds based on region growing policies.

The batch versions of the segmentation algorithms considered are presented in section 2.4. Our algorithm follows a similar structure, with the addition of incremental logic, generalized to arbitrary growing policies. Here we call the regions resulting from the growing process clusters, where each cluster is associated with a unique cluster ID. Clusters, whose number of points is included in a desired bound, are called segments and obtain a unique segment ID.

Incremental segmentation is achieved by using only new points as seeds for growing regions. New regions can possibly be merged with existing clusters. We consider a 3D point cloud \( P = \{p_i\} \), where each point contains its \( x, y, z \) coordinates and its cluster ID \( c > 0 \). Points that have not been segmented yet because the local map has just been updated have \( c = 0 \). In addition we maintain a mapping from cluster to segment IDs \( M^{ID} \). It is important to note that each point is clustered only once. This does not replicate the behavior of the batch algorithms faithfully. In particular, if the centroid of a voxel changes (i.e. a point is added to the voxel), we do not check if this may change the resulting clustering. This is an approximation that has negligible impact on the quality of the resulting segmentation but improves performance significantly.

#### 3.3.1 Partial Clusters

While the batch algorithms only need to cover the cluster growing case, the incremental version must also handle the cluster merging case. This can happen when, while growing a cluster, a point that has been previously assigned to another cluster is reached. In that case the two clusters have to be merged. Examples of the grow and merge cases are shown in fig. 3.2a and fig. 3.2b respectively. More complex scenarios (fig. 3.2c) can be decomposed to a sequence of merge steps.

In order to handle the merge case efficiently we add the concepts of partial cluster and partial cluster set. When segmentation starts, existing clusters are treated as partial clusters and new seeds originate new partial clusters. Initially, each partial cluster belongs to its own partial cluster set. When partial clusters need to be
3.3. Incremental Region Growing Segmentation

Figure 3.2: Different scenarios that the incremental region growing algorithm can encounter during segmentation. Here, for simplicity, only Euclidean distance policies are used. (a) In the grow case an unclustered point is reached and added to the current partial cluster. (b) In the merge case an existing partial cluster (blue) is reached while growing a region (orange). The two partial clusters are joined in the same set. (c) Other scenarios can be handled as a combination of the base cases. Here the current partial cluster (orange) reaches two other partial clusters (blue and green). Sequentially, both of them are merged into the same set.

merged, their partial clusters sets are merged into one. Once all points have been clustered, all partial clusters belonging to the same set obtain a single cluster ID and are actually merged into the same cluster in one single operation.

3.3.2 Overview

The high level structure of the incremental segmentation algorithm is composed of four main steps:

1. Partial clusters construction: using the cluster IDs stored in the local cloud, partial clusters are constructed for already clustered points.

2. Seeds preparation: indices of the new points are collected and put in a list of seeds. Depending on the segmentation policy, this can include further preprocessing as shown in section 3.3.5.

3. Region growing: regions are grown starting from the selected seeds. This process is explained in detail in the next sections.

4. Cluster indices write: finally, a new cluster ID is assigned to each set of partial clusters and the IDs are written in the point cloud. The $MID$ mapping is updated accordingly.

Storing the cluster IDs directly in the point cloud is preferred, since an external data structure would need to be updated every time points are added to or removed from the local cloud. For this reason, partial clusters have to be reconstructed at each segmentation, with negligible overhead.

3.3.3 Region Growing

Regions are grown using the precomputed seeds as starting points. Seeds are used only if they have not been processed yet in the current segmentation pass. When a seed is added to a partial cluster, it is marked as processed. Algorithm 1 shows the pseudocode for growing a single region. The GROWFROMSEED function takes
three arguments: the starting seed \( s \), a method for searching the nearest neighbors of a point \( NN \) and the point cloud \( P \) that can contain previously clustered points as well as new unclustered points. The \( NN \) method is, in our case, a \( k \)-d tree. Since the construction of the tree is an expensive operation, this structure is constructed once and shared with the normal estimator.

Initially, the queue of seeds contains only \( s \). The algorithm then iterates through the seeds until the container is empty. At each iteration one seed is removed from the queue and the \( NN \) method is used to find neighbor points with a maximum Euclidean distance from the seed. Neighbors are added to the partial cluster only if the \text{CanGrowToPoint} policy determines that it is possible to grow from the seed to the neighbor.

When the region grows to include a neighbor, two cases are possible:

1. If the neighbor point was not assigned to a cluster: The point is marked as processed and is added to the current partial cluster. Then the \text{CanPointBeSeed} policy is used to determine if the neighbor can be used as seed to further extend the region.

2. If the neighbor point was already assigned to a cluster: The partial clusters to which the seed and the neighbor belong are linked so that they belong to the same partial clusters set.

\begin{algorithm}
\caption{Incremental region growing starting from a seed \( s \). Growing is iterated according to the desired growing policies. The result is a partial cluster containing the indices of the points of the region, possibly linked with other existing partial clusters.}
\begin{algorithmic}[1]
\Function{GrowFromSeed}{\( s \), \( NN \), \( P \)}
\State \( \text{Seeds} \leftarrow \{ s \} \)
\State \( \text{\( C^P_s \)} \leftarrow \{ s \} \)
\While{\text{Seeds} \neq \emptyset}
\State \text{seed} \leftarrow \text{First} (\text{Seeds})
\State \text{Seeds} \leftarrow \text{Seeds} / \text{seed}
\For{\text{each neighbor} : \text{\( NN(\text{seed}) \)}}
\If{\text{CanGrowToPoint}(\text{seed}, \text{neighbor})}
\If{\( P[\text{neighbor}].c \neq 0 \)}
\State \text{LinkPartialClusters}(\text{seed}, \text{neighbor})
\ElseIf{\text{IsProcessed}[\text{neighbor}] \leftarrow \text{true}}
\State \( \text{\( C^P_s \)} \leftarrow \text{\( C^P_s \)} \cup \{ \text{neighbor} \} \)
\If{\text{CanPointBeSeed}(\text{neighbor})}
\State \text{Seeds} \leftarrow \text{Seeds} \cup \{ \text{neighbor} \}
\EndIf
\EndIf
\EndIf
\EndFor
\EndWhile
\State \Return{\text{\( C^P_s \)}}
\EndFunction
\end{algorithmic}
\end{algorithm}
3.3. Incremental Region Growing Segmentation

Figure 3.3: Evolution and tracking of segments as the robot moves. The black circle represents the boundary of the local map. In this example, the minimum size a cluster must have in order to be considered a segment is 3. (a) The robot observes a segment (blue) and a cluster (orange). (b) As the robot moves, more points are inserted in the local map. The orange cluster turns into a segment and another cluster appears. The blue segment grows but maintains the same ID. (c) From a different perspective, more points are observed, triggering the merge of the blue and orange segments. (d) Once part of a segment falls outside the local map, the segment is invalidated. More points may still be added to it and it may still be merged with other segments, invalidating them.

3.3.4 Linking Partial Clusters

When two partial clusters reach each other during segmentation they are linked by merging their containing partial clusters set. By doing that, the segment IDs of the partial clusters sets must be merged as well. The segment ID of a partial clusters set can be:

- A valid ID, in case the partial cluster set already contains a valid segment.
- A special constant $NoId$ denoting a partial cluster set that is not considered
a segment yet. This is the case when it does not have enough points or if it contains only new partial clusters and was not associated to a segment ID yet.

- A special constant $InvId$ denoting a partial cluster set that has been invalidated. This can happen if it contains too many points or if, because of the movement of the robot, some of its points have been removed from the local cloud making the segment incomplete.

Segment IDs are merged according to the following rules. Rules are applied in the given order:

1. One of the two partial clusters sets is invalidated. In this case the segment ID of the resulting set is set to $InvId$.

2. Both segment IDs are $NoId$. The resulting set will maintain $NoId$ as segment ID.

3. One of the two segment IDs is $NoId$. Then the segment ID of the resulting set will be set to the other segment ID.

4. Both partial clusters sets already have a valid segment ID. In this last case the minimum (i.e. the oldest) segment ID is used for the resulting set. Additionally, a segment renaming operation from the newest to the oldest ID is triggered.

Segment renaming is performed when two segments that were originally considered disconnected are merged as the result of new points linking them. This is done in order to reduce the number of incomplete or duplicate segments in the target map, which would unnecessarily increase the load on the matching and recognition stages. Renaming consists in transferring the views of the original segment to the destination segment. The original segment is then removed from the target map. We choose to always rename the newest to the oldest segment in order to increase the stability of the target map. Continuous changes in the ID of an old and more complete segment would reduce the effectiveness of the incremental recognition stage (see section 3.4).

3.3.5 Growing Policies

The region growing policies framework provides the freedom to perform incremental segmentation using arbitrary rules. The algorithm accepts three kind of policies. 

- **PrepareSeeds** creates an ordered list containing the indices of the points that can be used as seeds for growing new regions.
- **CanGrowToPoint** decides if, starting from the given seed, it is allowed to grow to a specific neighbor.
- **CanPointBeSeed** determines if the given point can be used as seed for starting or growing a region.

**Euclidean Distance Policies**

Since the incremental region growing algorithm already finds candidate neighbors for growing based on their euclidean distance, both **CanGrowToPoint** and **CanPointBeSeed** always return true. The preparation of the seeds is also trivial and simply collects the indices of points that are not assigned to a cluster yet. This set of policies results in a segmentation very similar to the one provided by the batch euclidean clustering algorithm [15]. The only difference lies in the approximation stated in section 3.3: once a point is clustered, its cluster assignment is not changed anymore, even if new points are added to its voxel and its position is updated.
Smoothness Constraints Policies

The policies based on smoothness constraints are derived from the work by Rabbani et al. [11]. During the preparation phase, the indices of the new points are collected and sorted in increasing curvature order. The sorting step guarantees that regions are grown starting from the flattest points, reducing the number of segments created. Moreover, the indices of points that fail the CanPointBeSeed test are removed from the list.

CanGrowToPoint returns true only if the angle between the normals of the seed and neighbor points falls below a given threshold. Another maximum threshold is applied on the point curvature in order for the CanPointBeSeed test to pass.

With smoothness constraint policies, the segmentation resulting from the incremental algorithm is not identical to the one produced by the original batch algorithm. In addition to the approximation used for the euclidean distance policy, we also ignore changes in the normal vectors. This can be avoided by re-segmenting points whose normals changed but, as the results in section 4.3.2 show, this is not necessary for the environments considered.

3.3.6 Segment Tracking

While the cluster ID is only a temporary value used for identifying points belonging to the same cluster, the segment ID is a lifetime-long identifier of a segment. This allows to track segments in the local map even after addition and removal of points. Figure 3.3 shows an example of such behavior.

Thanks to the segment tracking feature, it is possible to distinguish which segments are observed for the first time and which ones have already been found before. This has three main benefits: increased robustness of the target map, possibility to store multiple views of the same segment and matches caching. Prior to this work [1], multiple views could not be associated to the same segment and would be inserted in the target map with different IDs, causing the target map to contain segment duplicates. At each insertion, a NN search is necessary to find and remove segments that are too close too the inserted segment, and thus likely to be duplicates. With segment tracking multiple views of the same segment can be stored under the same ID, providing a complete history of the observations of the segment. This opens several possibilities: for example, the matching stage could attempt to match to old views as well, reducing the number of false negatives caused by the differences in viewing angle. Moreover, matches caching can be used for performing incremental recognition as shown in section 3.4.2. Outside the context of this thesis, segment tracking has already enabled a method [30] for learning segment descriptors that are more robust to changes in the point of view.

3.4 Graph-Based Incremental Recognition

This section presents an improved version of the geometric consistency grouping presented in section 2.5. In our approach we formulate recognition as a graph problem with the goal of identifying a maximum geometrically consistent set which

---

1. The graph-based approach presented in this section, including the partitioning scheme (section 3.4.1) and the clique detection strategy (section 3.4.3), has been published in [31]. A copy of the paper is included in appendix A.
Figure 3.4: Overview of the recognition process with minimum geometrically consistent set size $T = 3$: (a) Current approaches need to test all possible correspondence pairs for consistency. Nodes represent the correspondences, while the tested pairs are shown as edges. (b) The partitioning approach presented in section 3.4.1 allows to drastically reduce the number of consistency tests. (c) The consistency graph obtained with our method and the recognized $T$–clique (in blue) of geometrically consistent correspondences. (d) Example of a successful recognition in a urban driving scenario experiment. Correspondences that constitute the maximum consistent set are highlighted in green.

is a set of maximum size consisting of correspondences that are all pairwise geometrically consistent.

Pairwise geometrical consistency relationships are encoded in a consistency graph, an undirected graph $G = (V, E)$ where $V = \{c_i\}$ is the set of correspondences $c_i$ and $E = \{e_{ij}\}$ is the set of undirected edges $e_{ij}$ connecting all consistent pairs of correspondences $(c_i, c_j)$. Once the graph is constructed, identifying a maximum geometrically consistent set is equivalent to finding a maximum clique of $G$. Figure 3.4 shows an overview of the recognition process. In this section, two different methods for constructing the consistency graph are presented.

3.4.1 Partitioned Consistency Graph Construction

The naive approach for identifying all pairwise consistencies is testing all possible correspondence pairs. Here we take advantage of our knowledge about the sizes of the target and local maps to reduce the number of tests.

In many recognition applications, like place recognition as in our case, the local map is significantly smaller than the target map. Exploiting this information we can define a criterion for reducing the amount of consistency tests a priori. Let $\epsilon$ be the tolerance for geometric consistency and $b$ be the diameter of the bounding
sphere of the local map centroids, a pair of correspondences \( c_i \) and \( c_j \) can then be consistent only if
\[
d_t(c_i, c_j) \leq b + \epsilon \tag{3.14}
\]
Environments for autonomous vehicles usually extend along the earth’s surface only (\( x \) and \( y \) coordinates), but not vertically. Without loss of performance or correctness, we simplify the problem and consider only a bounding cylinder of diameter \( b \). The new relaxed criterion becomes
\[
d^{2D}_t(c_i, c_j) \leq b + \epsilon \tag{3.15}
\]
where \( d^{2D}_t(c_i, c_j) \) computes the distance between the target map centroids of \( c_i \) and \( c_j \) projected on the \( xy \)-plane.

We create a 2D grid partitioning of the correspondences according to the position of their centroids in the target map, where each partition \( p_{u,v} \) has a square base with a side length of \( b + \epsilon \) and infinite height. An example of 2D grid partitioning is illustrated in fig. 3.4c. Each correspondence \( c_i \) is assigned to its partition \( P(c_i) = p_{u,v} \),
\[
\begin{align*}
  u &= \left\lfloor \frac{t_c(c_i).x - o.x}{b + \epsilon} \right\rfloor, \\
  v &= \left\lfloor \frac{t_c(c_i).y - o.y}{b + \epsilon} \right\rfloor
\end{align*}
\tag{3.16}
\]
where \( t_c(c_i) \) is the centroid of \( c_i \) in the target map and \( o \) is the origin of the grid. A good choice of \( o \) is the componentwise minimum of all \( t_c(c_i) \).

With the chosen grid size \( b + \epsilon \), it is guaranteed that the bounding cylinder of the model is always contained in a squared group of four adjacent partitions. Thus geometric consistency tests are necessary only on a set of candidate pairs of correspondences that is much smaller than \( V \times V \):
\[
\{(c_i, c_j) \in V \times V \mid i < j \land \exists u, v : c_i \in p_{u,v} \land c_j \in \mathcal{N}(u, v)\}
\]
\[
\mathcal{N}(u, v) := \bigcup_{l,m \in \{-1, 0, 1\}} p_{u+l,v+m} \tag{3.17}
\]
i.e. for a given \( c_i \in V \) only correspondences in the partition of \( c_i \) and the 8–neighbor-partitions need to be tested. Since consistency is a symmetric property, each pair is tested only once, i.e. if \( i < j \). Other pairs of correspondences are ignored, since they cannot be consistent. The consistency graph is constructed as an adjacency list and contains all the geometrically consistent correspondence pairs \( (c_i, c_j) \) as edges. Fig. 3.4a shows a set of 7 correspondences where all 21 pairs are tested for consistency. Applying this partitioning strategy reduces the number of tests to 14 (Fig. 3.4b).

The same approach can be used for cases where the environment extends into the third dimension. In this case, assuming that the local map is bounded by a sphere of diameter \( b \), a 3D grid of cubes with size \( b + \epsilon \) is used and consistency tests are performed over the 26–neighborhood of each partition.

### 3.4.2 Incremental Consistency Graph Construction

We observe that, as the robot moves, part of the matches appear multiple times in successive localization steps. Thanks to the segment tracking feature described in section 3.3.6, we can uniquely identify and track correspondences using a combination of the IDs of the matched segments.

---

2The z-axis is assumed to be roughly gravity aligned.
In this section we aim at constructing the consistency graph in an incremental fashion, reusing consistency information from previous recognition steps. As segments are continuously updated, their centroids also change, making caching of pure pairwise consistency information ineffective. We rather choose to cache, for each correspondence $c_i$, an arbitrary set of correspondences that are potentially consistent with $c_i$. Such a set consists of the correspondences that are close to being consistent with $c_i$ according to the measure defined in eq. (3.18). This is done through a consistent candidates cache. Consistency tests for the construction of the consistency graph then need to be performed only between each correspondence and its cached candidates, significantly reducing the total number of tests.

**Consistent Candidates Cache**

Based on the condition for geometric consistency stated in eq. (2.3), we define a measure for how far two correspondences are from being consistent:

$$
\Delta(c_i, c_j) = |d_l(c_i, c_j) - d_t(c_i, c_j)|
$$

(3.18)

where $d_l(c_i, c_j)$ and $d_t(c_i, c_j)$ are the distances between centroids in the local map and in the target map respectively. $\Delta(c_i, c_j)$ is called the consistency distance between $c_i$ and $c_j$. The consistent candidates set $S(c_i) \subset V$ of a correspondence $c_i$ is defined as

$$
S(c_i) = \{ c_j \in V \mid j \leq i \land \Delta(c_i, c_j) \leq \theta \Delta + \epsilon \}
$$

(3.19)

i.e. the set of correspondences whose consistency distance is less or equal to an arbitrary threshold $\theta \Delta$. $\epsilon$ is the tolerance for consistency and the condition $j \leq i$ prevents duplicate entries of the same pair (caused by the symmetry of the consistency relation) from being stored. This assumes that the indices of the correspondences are assigned in increasing order. The consistent candidates cache $C$ contains, for each of the current correspondences, its consistent candidates set.

**Cache Maintenance**

When a correspondence $c_i$ is found for the first time, $S(c_i)$ is computed and stored in the cache, together with the centroid of the local map segment. This can be done efficiently using the partitioned approach described in section 3.4.1. When, in successive recognitions, the correspondence is not observed anymore, all references to it are removed from the cache.

Consistent candidates sets are generally computed only once when the correspondence first appears. However, for an exact result, it is necessary to consider the case in which one segment of the local map changes so much that its cached information is not correct anymore. When the segment centroid of a cached correspondence $c_i$ moves from its cached position by a distance $\delta > \frac{1}{2} \theta \Delta$, the cached information can become outdated and consistent pairs may be ignored. This can happen when the local map centroid of another correspondence $c_j \notin S(c_i)$ moved from its cached position by a distance greater than $\theta \Delta - \delta$. Combining the changes in the two segments, the new consistency distance between $c_i$ and $c_j$ could fall below the threshold for consistency $\epsilon$.

In order to ensure exact results we apply a conservative approach and we invalidate the cached information of a correspondence when its local map centroid moves by at least $\frac{1}{2} \theta \Delta$ from its cached position. The correspondence is then reinserted in the cache as a new correspondence, causing its consistent candidates set to be recomputed. Figure 3.5 shows an example of consistency cache maintenance.
25 3.4. Graph–Based Incremental Recognition

Figure 3.5: Visualization of an example of consistency cache maintenance. Correspondences have arbitrary positions, while the distance between correspondences represents their consistency distance according to eq. (3.18). Arrows point from correspondences to their consistent candidates. The circles around the correspondences represent the threshold for caching $\Delta \theta$. (a) The correspondences $c_1$, $c_2$ and $c_3$ are inserted in the given order, and $c_1$ is found to be a candidate for consistency with $c_2$. (b) When $c_4$ is inserted, $c_1$ and $c_2$ are cached as its candidates for consistency. (c) The centroid of the local map segment of $c_4$ changes by a distance smaller than $\frac{1}{2} \theta \Delta$, thus its cached information is still valid. $c_2$ changes by a distance greater than $\frac{1}{2} \theta \Delta$, thus its consistent candidates are recomputed.

The method above assumes that the segment centroids in the target map do not change. This is true except when a loop closure is found. In that case, based on the changes in the target segments, some cached correspondences may need to be invalidated and recomputed.

The ideal threshold $\theta \Delta$ depends on the expected size of the segments in the considered scenario. In general, high values of $\theta \Delta$ result in bigger consistent candidates sets, while small values increase the number of invalidated cache entries. In our experiments we found $\theta \Delta = 3 \text{ m}$ to give the best performance.

Consistency Tests

Consistencies involving new correspondences are identified while searching for their consistent candidates sets. For all other correspondences $c_i$, consistency tests are performed only with respect to the cached candidates, i.e. on the set of correspondences pairs $\{c_i\} \times S (c_i)$.

3.4.3 Maximum Consistent Set Identification

We consider a recognition to be successful in case the size of the detected maximum geometrically consistent set is greater than or equal to a threshold parameter $T$. Thus we need to identify a maximum $k$–clique with $k \geq T$.

Enforcing the partitioning constraints in situations where the local map is significantly smaller than the target map guarantees that the consistency graph is sparse. With this knowledge we can rely on a class of $k$–clique detection algorithms that visit the graph in degeneracy order to find the maximum geometrically consistent set. A graph $G$ can be characterized by its degeneracy (or $k$–core number) $d$, which is the smallest number so that every subgraph of $G$ contains a vertex with degree
Each graph has a degeneracy ordering, an order in which each vertex has at most \( d \) vertices that come later in the ordering.

We use a generalization of the maximal clique listing approach described in \cite{32,33} to find a maximum clique of \( G \). An outline of the algorithm is presented in Algorithm 2. First, we use the bucket sort algorithm for sorting the vertices in increasing degree order. This is described in detail in \cite{34} and can be performed in \( O(|V|) \). Then we iterate on the vertices according to the found order. At each step, the function \( \text{Clique} \) is called using the current vertex of minimum degree as input. If the function returns a clique that is bigger than the current maximum clique \( \text{max}\_\text{clique} \), the new maximum clique is stored. The vertex is then removed from the graph together with its incident edges and the vertex ordering is updated in \( O(d) \). The resulting degeneracy ordering in the visit of the vertices guarantees that at each step \( v \) has at most \( d \) neighbors, bounding the computational load on \( \text{Clique} \).

\( \text{Clique} \) can be any function that returns the biggest clique \( C \subseteq G \) such that \(|C| \geq T \) and \( v \in C \) (or the empty set, if such a clique does not exist). Instances of this function can be any clique detection approach like the branch-and-bound method presented in \cite{35} or successive refinements like \cite{36}. For simplicity, in this work we use a method based on heavy pruning strategies \cite{37}. This algorithm builds a subgraph containing \( v \) and its neighbors and recursively visits its subgraphs to determine if a clique of the required size can be built. Vertex degrees are constantly checked to discard fruitless candidates as soon as possible in the recursion.

\begin{algorithm}
\caption{Outline of the algorithm for maximum clique detection. Iterating over the vertices in degeneracy order guarantees that \( v \) always has at most \( d \) neighbors.}
\begin{algorithmic}[1]
\Function{MaximumClique}{\( G, T \)}
\State \( \text{max}\_\text{clique} \leftarrow \emptyset \)
\State \( \text{sorted}\_\text{vertices} \leftarrow \text{SortByDegree}(G) \)
\While{\( |\text{sorted}\_\text{vertices}| \neq 0 \)}
\State \( v \leftarrow \text{sorted}\_\text{vertices}[0] \)
\State \( C = \text{Clique}(v, G, T) \)
\If{\( |C| \neq 0 \)}
\State \( \text{max}\_\text{clique} \leftarrow C \)
\State \( T \leftarrow |C| + 1 \)
\EndIf
\State \( \text{RemoveVertex}(G, v) \)
\State \( \text{UpdateOrder}(G, \text{sorted}\_\text{vertices}) \)
\EndWhile
\State \Return{\( \text{max}\_\text{clique} \)}
\EndFunction
\end{algorithmic}
\end{algorithm}

Transformation Estimation

The condition on the size of the maximum geometrically consistent set can be removed in case it is safe to assume that a true recognition always exists. This is however not always possible, e.g. in loop-closure detection applications. Once an acceptable maximum clique is identified, the relative transformation between the local and target maps can be found using any rigid transformation estimation method. Here we use the least-squares approach described in \cite{38}. 

3.4.4 Complexity Analysis

For this analysis we consider partition–based construction only. The introduction of caching approximately improves performance by a constant factor, but does not change the asymptotic scaling.

In the consistency graph $G = (V, E)$, let the parameters $n = |V|$ be the number of correspondences and $d$ be the degeneracy of the graph. Assuming that outliers are uniformly distributed in the target map and are present in a high ratio over inliers, we can say that $n$ is proportional to the size of the target map. The average number of correspondences per partition $n_p$ and $d$ are considered constant as they depend on fixed parameters like the density of objects in the environment, the size of the local map and the correspondence matching strategy.

Partitioning the correspondences is trivial and can be done in $O(n)$. All the necessary consistency tests can be performed in $O(n_p n)$ since each of the $\frac{n}{n_p}$ partitions requires $O\left(n_p^2\right)$ tests. The bucket sort is done in $O(n)$. Since every vertex visited in the loop has at most $d$ neighbors, each of the $n$ calls to CLIQUE has a worst case complexity of $O(d!)$, thus the complexity of the code in Algorithm 2 is $O(d! n)$. Although the $d!$ term may appear like an important bottleneck, any reasonable implementation for CLIQUE usually performs better. As shown in Table 4.2, maximum clique detection brings a minimal contribution to the total runtime. The estimation of the 3D transformation scales linearly with the number of elements in the geometrically consistent set, thus it is bound by $O(d)$.

The total complexity of our method is $O(n (d! + n_p))$. Under the homogeneity assumption, since $d$ and $n_p$ are parameters that do not depend on the target map’s size, we can conclude that the performance of our algorithm scales linearly with the size of the target map.
Chapter 4

Results

This chapter describes the experiments performed in order to evaluate the developed methods. The results first focus on performance in section 4.2 and will expand on quality and functionalities in section 4.3.

4.1 Experimental Setup

The system has been tested in four different conditions:

- **KITTI Localization**: The vehicles drives through a known urban scenario continuously trying to localize. The dataset is part of the KITTI Vision Benchmark Suite [39].

- **KITTI Loop Closure**: The vehicle explores an unknown urban scenario, continuously maintaining a dynamic target map and trying to detect loop-closures.

- **Powerplant Localization**: A rescue ground vehicle drives in a known indoor scenario continuously trying to localize. The dataset has been recorded at the Knepper power plant in Dortmund in the context of the Long-Term Human–Robot Teaming for Disaster Response (TRADR) project [40].

- **Powerplant Multi-Robot**: Three robots drive in an unknown scenario without knowledge of their relative poses, continuously updating a dynamic target map and trying to detect loop-closures. This experiment has only been used to verify the correctness of the incremental normals estimator in section 4.3.1.

The baseline used for the comparison is the original SegMatch approach [11]. The incremental version uses the DVG, the incremental normals estimator (powerplant examples only), the incremental region growing segmenter and the incremental graph-based recognizer. Simple Euclidean distance policies have been used for the KITTI examples, while the powerplant scenario was tested with smoothness constraints policies. Our implementation also includes some minor performance optimizations.

Detailed statistics about the experiments are summarized in table 4.1. Values are computed as mean and standard deviation of measurements performed at each place recognition step of the experiment. Standard deviation is computed within a single experiment, capturing the variance caused by the navigation through different regions.
Chapter 4. Results

Figure 4.1: Speedup of the incremental segment based place recognition over the baseline version. The loop closure example has a smaller speedup and higher standard deviation.

Hardware

All experiments have been performed on a relatively dated machine (year 2011), which is comparable to current medium–low cost hardware. The system has 8GB or RAM and an Intel i7–2600K processor with 8MB cache and 4 cores running at 3.4GHz. The segment matching pipeline runs in single–threaded mode and, with the datasets used, the amount of RAM is not a limitation.

Assuming that the local cloud never contains more than 200000 points, cache thrashing is not an issue: Each point contains $x$, $y$, $z$ coordinates, cluster IDs and padding fields for a total of 32 bytes. The total size occupied by the local cloud does not exceed 6.4MB.

If normals are required, the normals estimator has higher memory requirements for storing the components of the covariance matrices of each point. However, since normals are estimated incrementally, only a fraction of the data is actually accessed, reducing cache misses.

4.2 Performance

The system has been benchmarked using the KITTI (localization and loop closure) and powerplant (localization) scenarios. All scenarios have been tested using both the baseline and the incremental version of the place recognition method.

Figure 4.1 shows the result of the benchmarks. The speedup achieved by the incremental approach is 4.84x, 2.34x, and 7.41x respectively for each scenario. No justified observation can be made on the Coefficient of Variation (CV) $c_v = \frac{\sigma}{\mu}$, meaning that the incremental approach increased the speed, but did not affect the relative oscillations caused by differences in the environment.

In both localization examples the system can process the inputs faster than the update rate of the sensor. In the loop closure setting, the pipeline is slower than the sensor, causing new scans to queue while waiting to be processed. This situation is not optimal for the incremental method as its speed is based on the assumption that, at each update, only a small number of new points must be processed, with minimal changes on the stored state. If points start queuing, the system suffers of
**Table 4.1:** Statistics and runtimes (mean and standard deviation) characterizing the different experiments. Values refer to observations made in one place recognition step (e.g., points per step). It is interesting to note some strong differences between experiments, e.g., number of partitions and cache invalidations, caused by the changes in scenario and configuration. These are discussed in detail in the following sections.

A feedback loop behavior: At each step the system must process a higher number of points, becoming slower and causing more scans to be placed in the queue. Despite the sub-optimality of such cases, the incremental system can still run faster than the original version.

The second factor contributing to the reduced speedup in the loop closure example can be observed in fig. 4.2. The plots show the fraction of the runtime dedicated to each step of the segment matching pipeline. Clearly the runtime of the system in loop closure settings is dominated by external operations. These include processing of the loop closures and update of the target map. In the latter, the k-d tree used in the matching step must be rebuild to include the features of the newly observed segments. Observing the segmentation matching stages we observe that a high fraction of the runtime is still required by the segmenter. This includes the construction of a k-d tree containing all the points of the local map.

The following sections will analyze the runtimes of the single stages in detail.
Figure 4.2: Fraction of the runtime required by each of the segment matching stages. The runtime of the loop closure example is strongly affected by external operations like loop closure processing and target map update which have not been addressed in this thesis as they are not directly in the scope of place recognition.

Figure 4.3: Speedup of the DVG over the batch voxel filtering provided by the PCL (pcl::VoxelGrid). The measurements include one pose update (removal of voxels outside the radius of the local cloud) and one insertion of the queued scans.

4.2.1 Dynamic Voxel Grid

As described in section 3.1, voxel filtering requires a sorting step to group points belonging to the same voxel. This $O(n \log n)$ step has a significant impact if the local cloud contains a lot of points (see table 4.1). With the incremental approach, this operation is reduced to sorting the new points only and then merging them with the stored sorted points in linear time. This optimization enabled speedups of 8.08x, 11.18x, and 12.92x respectively in the benchmarked scenarios. As shown in the statistics, the number of new voxels is only a small fraction of the total, justifying the observed speedup.

Additionally, the DVG correctly provides information about which voxels are created by accumulating a scan in the local cloud, allowing the incremental normals estimator to only recompute the normals of the points affected by the insertion.
4.2. Performance

Figure 4.4: Speedup of the incremental normals estimator over the batch normals estimator provided by the PCL (pcl::NormalEstimation). The third bar shows the runtime of the estimator reusing the $k$–d tree built by the segmenter.

Figure 4.5: Speedup of the incremental region growing segmentation over the batch methods provided by the PCL (pcl::EuclideanClusterExtraction and pcl::RegionGrowing).

4.2.2 Incremental Normals Estimation

Figure 4.4 shows the runtimes of the original and incremental normals estimators. The estimators have been tested in the powerplant scenario only, as for the KITTI examples simple Euclidean distance policies are used, which do not require point normals. In an equitable comparison, where both estimators need to build a $k$–d tree of the local cloud, the incremental approach is 7.65 times faster in average. Recycling the same tree built by the segmenter leads to a 17.17x speedup. It is important to note that this last value denotes an optimization in the code implementation only, while the algorithmic improvement can be observed only when both estimators build their own $k$–d tree. The speedup of the incremental approach is explained by the smaller number of $\text{NN}$ searches required and by the caching of the covariance matrix, that does not need to be recomputed at each step.

4.2.3 Incremental Region Growing Segmentation

The performance improvement obtained by performing segmentation incrementally is shown in fig. 4.5. Similarly to the normals estimator, the most important im-
Figure 4.6: Speedup of incremental recognition over the simple geometric consistency approach. Since the powerplant environment contains a small number of segments, the recognizers can test for consistencies very quickly.

Improvement is the reduction of the number of \( \text{NN} \) searches performed at every step. This is achieved by reusing stored information about the clusters in the cloud and only starting region growing from new unsegmented points.

Incremental segmentation achieves a speedup over the batch methods of 5.82x, 5.15x and 11.52x. The two KITTI examples use similar environments, thus they produce similar speedups. Contrastingly, the powerplant scenario benefits of much higher improvement. This is explained by code performance differences in the two implementations. In our code, the angle between normals is never computed explicitly. Instead, only the dot product between the normals is computed and compared to the precomputed inverse cosine of the angle threshold. The PCL implementation is penalized by not precomputing this value. Since trigonometric functions are relatively slow, this difference can have a significant impact.

4.2.4 Graph–Based Recognition

Recognition is a key component of the localization pipeline, and the most critical in terms of performance. While the runtime of most of the stages depends only on the properties of the sensor and of the environment (e.g. acquisition rate, local cloud radius and density), matching and recognition depend on the size of the target map as well. Even without optimization, matching is not a bottleneck: its runtime scales \( O(n \log n) \) in the number of target segments and can easily be accelerated using an incremental approach as suggested in chapter 5. Contrastingly, the original recognition approach scales cubically with the number of correspondences (and thus, assuming a homogeneous map, with the size of the target map). This makes the application of the method to city–scale or bigger environments computationally infeasible.

As shown in section 3.4.4 our partitioned approach achieves linear scaling, enabling usage in larger scenarios. Moreover, the incremental caching of candidate consistencies further accelerates the method. The performances of the original and graph–based methods are compared in fig. 4.6. The achieved speedups are 15.58x, 15.68x and 6.16x respectively. Since the powerplant scenario is contained in a single partition, the speedup is caused by the incremental caching only. As shown in table 4.2 exploiting the low degeneracy of the consistency allows fast detection of maximum cliques.
4.3. Quality

Figure 4.7: Mean runtime of the recognition stage with different caching radii. None indicates the pure partitioned approach without caching. In our dataset the ideal compromise between caching and invalidation is about 3 m.

<table>
<thead>
<tr>
<th>Step</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency tests</td>
<td>7.77 ± 4.43</td>
</tr>
<tr>
<td>New matches</td>
<td>4.86 ± 3.43</td>
</tr>
<tr>
<td>Cached matches</td>
<td>2.30 ± 1.21</td>
</tr>
<tr>
<td>Partitioning</td>
<td>0.07 ± 0.02</td>
</tr>
<tr>
<td>Transform estimation</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>Max clique detection</td>
<td>0.25 ± 0.12</td>
</tr>
<tr>
<td>Total</td>
<td>8.43 ± 4.75</td>
</tr>
</tbody>
</table>

Table 4.2: Runtimes of the steps of the graph–based incremental partitioning in the KITTI localization experiment. Cached correspondences represent > 90% of the total (Table 4.1), but are tested for consistency faster than the remaining new correspondences.

Figure 4.8: Dead reckoning plot indicating the probability to travel a specific distance without a successful localization. The data are recorded during 5 runs of the KITTI localization example.

Figure 4.7 compares the effect of different thresholds $\theta_\Delta$ on the consistency distance for caching. If the radius is too big, lot of candidates are stored, requiring a high number of consistencies tests. Decreasing the radius too much can cause more frequent invalidation of cached correspondences, due to segments being updated. The best setting for the radius is a trade–off between number of candidates and invalidation frequency.

In the KITTI localization experiment, the incremental recognizer performed on average 313497 and 30839 consistency tests on new and cached correspondences respectively. This is only 7.2% of the about 4.7 million tests performed by the baseline implementation at each recognition step.
Figure 4.9: Example of multi-robot SLAM. During the exploration process the poses of the robots are updated. The components of the covariance matrices used by the incremental normals estimator are transformed as well. (a) Three robots drive in an unknown environment without knowledge of their relative poses. (b) One robot drives through a region already visited by another robot. A loop closure is found and the target maps are merged accordingly. (c) A loop closure involving the third robot is found. All target maps are now aligned. (d) The Knepper power plant, where the laser data have been recorded.

4.3 Quality

In principle the incremental approach is a performance optimization and the qualitative results closely resemble the ones of the original method [1]. However some assumptions and improvements made in this work affected the final result qualitatively.

Figure 4.8 shows the probability of traveling a specific distance without successful localization. Mathematically, this is equivalent to the inverse of the Empirical Cumulative Distribution Function (ECDF) of the time between two consecutive localizations. Since the incremental approach is significantly faster, it can also localize within shorter distances. Localization happens within 1.5m more than 90% of the times, while the original approach can localize within the same distance less than 10% of the times, occasionally traveling more than 10m without localization.

The higher number of localizations is an advantage also in mapping applications, since a higher number of successful loop closures adds more constraints to the factor graph of the trajectory estimator, compensating for drift and increasing the quality of the generated maps.
Figure 4.10: Comparison of the extracted segments between batch and incremental region growing segmentation with smoothness constraints. (a) View of the power-plant environment with segments extracted in one batch processing. (b) The same view segmented incrementally while the robot is driving through it.

Figure 4.11: Evolution of three segments (tree, car and building) growing as more points are accumulated in the local cloud. (Image credits: Renaud Dubé)

4.3.1 Incremental Normals Estimator

Section 3.2 introduced an incremental approach to normal estimation, based on incremental update of the components of the covariance matrix (eq. (3.6)). The updating of the accumulators after transformation of the local cloud is not trivial and a solution has been presented in section 3.2.3. Figure 4.9 shows a sequence of loop closures detected in the powerplant multi-robot example. After each transformation of the local cloud, segments continue growing, indicating that the covariance matrix components are updated correctly and normals remain consistent after a transformation.

4.3.2 Incremental Segmentation

As explained in section 3.3, the incremental region growing segmentation algorithm relies on a key approximation: Once a point is segmented, its assignment is not modified even if its position changes (because of more points being added to its voxel) or if its normal is updated (because of more point being added to its neighborhood).

In fig. 4.10 we compare the result of batch segmentation on a section of the powerplant dataset with an incremental segmentation of the same scenario. We observe
that after incremental processing more segments are created than in the batch case. Moreover segments appear to be more complete. However we can also observe a case of under–segmentation (bottom right orange segment, 4.10b). This is a consequence of the fact that a point can be erroneously added to a segment when its position and normal are inaccurate, but it is never removed after refinement.

Although this behavior does not reproduce the results of the original batch algorithms (section 2.4) faithfully, we do not observe any degradation in the quality of matching and recognition. Contrastingly, fig. 4.15 shows that the reduced processing time significantly improves the localization rate.

Figure 4.11 shows the evolution of three different segments as points are accumulated in the local cloud. Segment tracking allows to associate all views of the same segment, so that segments can be updated in the target map without creating duplicates.

4.3.3 Recognition

We performed additional tests to compare the quality of recognitions produced by the naive and graph–based methods. As metric for the quality we use the size of the identified maximum consistent set. The graph–based method uses an exact algorithm, and is always able to find a consistent set of maximum size. On the other side, the PCL implementation uses a greedy approach and occasionally fails to detect the optimal solution. Figure 4.12 shows a tested pathological case: depending on the order of the input correspondences the greedy approach can detect a maximum (Figure 4.12a in blue) or a maximal (Figure 4.12b in red) consistent set. Our method detects the maximum clique in both cases. Note that the PCL implementation does not explicitly build a graph, but detects the same pairwise consistency, thus we can use the same visualization.
4.4 Implementation

The presented methods have been integrated in the SegMatch project and will be open sourced on https://github.com/ethz-asl/segmatch after the publication of this work in a paper.

The contribution of this thesis consists in almost 15000 lines of code developed in C++11. The code follows the coding guidelines of the project and each change has been reviewed by at least one supervisor. Correctness of the implemented algorithms is ensured by a suite of unit tests developed during the thesis. Additionally, a benchmarking system was written for measuring the runtimes of the different components of the system and to collect statistics about the performed experiments. All public classes, methods and fields are documented with Doxygen syntax, so that HTML or LaTeX documentation can be generated.
Chapter 5

Conclusions and Future Work

This thesis presented a fast incremental approach to segment based place recognition in 3D point clouds. This is a key ability for autonomous driving tasks, since odometry–based trajectory estimation methods accumulate drift over time, which can be corrected with global localizations. After introducing place recognition and the challenges involved (chapter 1) we presented an overview of related work in the field (chapter 2) evaluating possible alternatives. Chapter 3 explained our incremental algorithms in detail, whose performances were evaluated extensively in chapter 4.

Contributions

Although there is prior literature in the area of incremental segmentation (see 2.5), to the best of our knowledge this is the first work to propose a fully incremental place recognition pipeline of this type. The main contributions of this thesis are:

- A novel method for incrementally estimating the normals of a 3D point cloud from a stream of points supporting affine transformations.
- A novel incremental and generic region growing segmentation algorithm supporting segment tracking, customizable growing policies and arbitrary cloud update patterns.
- A novel high performance graph–based recognition approach for detecting geometric consistencies in a set of correspondences through space partitioning and caching.
- Development for the open–source code base of the SegMatch project hosted by the ASL\textsuperscript{1}, including implementations of the new algorithms, design improvements, evaluation tools, documentation and unit tests.
- Qualitative and performance evaluation of the proposed incremental approach, comparing with the original batch methods.

The partitioned approach to recognition has been published in a workshop paper at the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)\textsuperscript{1}.

\textsuperscript{1}Git repository: https://github.com/ethz-asl/segmatch. The code written for this thesis will be available after the publication of the relative paper.
The main work about incremental place recognition will be submitted to the IEEE Robotics and Automation Letters (RA–L) journal.

Results

The presented approach was tested and evaluated on urban driving and rescue scenarios. The overall speed of the place recognition improved by a factor of 2.34x to 7.41x depending on the experiment, reaching a rate of 10Hz in the KITTI dataset. The performance improvements enable higher localizations rate, lower dead reckoning times and usage of the method on systems with limited computational resources such as Unmanned Aerial Vehicles (UAVs).

Such improvement has been made possible by the optimization and tight synergy of the different stages of the pipeline. The DVG provides information about the new points created in the accumulated local cloud, enabling the incremental normals estimator to update only the normals affected by the insertion. The incremental segmenter can track segments as they grow, enabling incremental recognition, more robust maintenance of the target map and a new method for learning more robust segment descriptors.

The incremental algorithms presented are significantly faster than their batch baseline versions, with speedups reaching 15.58x in case of incremental recognition. As shown in section 3.4.4, the performance of the new recognizer scales linearly with the size of the target map, while the original algorithm has cubic scaling. Moreover the presented method always detects the best recognition, avoiding suboptimal solutions that would be detected by the baseline method. This enables localization in larger scenarios, which were computationally intractable with the previous approach.

Future work

Although the work in this thesis greatly improved runtime and asymptotic scaling, some parts of the segment based place recognition method can be further optimized. The most important improvement we foresee is the usage of an NN search method based on a data structure that supports fast insertion and removal. Currently a k–d tree of the local cloud must be constructed after each update. Moreover, in loop closure settings, the k–d tree containing the descriptors is rebuild every time new segments are added to the target map. Possible solutions include updating and re–balancing the k–d tree, using more dynamic structures like octrees or exploit the locality of updates by partitioning the space and building multiple trees.

Segment tracking can be used for detecting segments bigger than the local cloud. Instead of invalidating them when they partially cross the boundary of the local cloud, segments should be maintained until they completely exit the local map. Moreover, the performance and design improvements contributed in this thesis now allow the usage of multiple segmenters and descriptors, that could facilitate place recognition in situations that are too challenging for the original system. Multiple segmenters could detect segments of different types, at multiple scales and resolution, providing a better description of the scene. As suggested in the results, the asymptotic scaling of the contributed methods theoretically enables place recognition in much bigger scenarios. This should be verified in practice using or acquiring bigger datasets.
In terms of implementation, improvements can be made in the strategy used for managing the local cloud. Currently, the points are compacted to a contiguous array after insertion and removal. Although this improves cache performance, it also causes a significant index management overhead when remapping cached information. With the incremental approaches, the bottleneck shifted from computational (mathematical operations on points and vectors) to control logic (indexing, caches maintenance) tasks. We encourage future work to rely on a sparser cloud where points maintain the same position in memory during their lifetime in the cloud.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>6DoF</td>
<td>Six Degrees of Freedom</td>
</tr>
<tr>
<td>ASL</td>
<td>Autonomous Systems Lab</td>
</tr>
<tr>
<td>CV</td>
<td>Coefficient of Variation</td>
</tr>
<tr>
<td>DBSCAN</td>
<td>Density-Based Spatial Clustering of Applications with Noise</td>
</tr>
<tr>
<td>DVG</td>
<td>Dynamic Voxel Grid</td>
</tr>
<tr>
<td>ECDF</td>
<td>Empirical Cumulative Distribution Function</td>
</tr>
<tr>
<td>ICP</td>
<td>Iterative Closest Point</td>
</tr>
<tr>
<td>IROS</td>
<td>IEEE/RSJ International Conference on Intelligent Robots and Systems</td>
</tr>
<tr>
<td>LiDAR</td>
<td>Light Detection And Ranging</td>
</tr>
<tr>
<td>NN</td>
<td>Nearest Neighbors</td>
</tr>
<tr>
<td>PCL</td>
<td>Point Cloud Library</td>
</tr>
<tr>
<td>RA–L</td>
<td>IEEE Robotics and Automation Letters</td>
</tr>
<tr>
<td>RRHT</td>
<td>Robust Randomized Hough Transform</td>
</tr>
<tr>
<td>SLAM</td>
<td>Simultaneous Localization And Mapping</td>
</tr>
<tr>
<td>TRADR</td>
<td>Long-Term Human–Robot Teaming for Disaster Response</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
</tbody>
</table>
Bibliography


Appendix A

Publications
A Partitioned Approach for Efficient Graph–Based Place Recognition

Mattia G. Gollub  Renaud Dubé  Hannes Sommer  Igor Gilitschenski  Roland Siegwart

Abstract—Place recognition is a crucial capability of autonomous vehicles that is commonly approached by identifying keypoint correspondences which are geometrically consistent. This geometric verification process can be computationally expensive when working with 3D data and with increasing number of candidates and outliers. In this work, we propose a technique for performing 3D geometric verification efficiently by taking advantage of the sparsity of the problem. Exploiting the relatively small size of the area around the vehicle, the reference map is first subdivided in partitions, and geometric verifications are only performed across relevant partitions, guaranteeing the sparseness of the resulting consistency graph. A maximum clique detection algorithm is finally applied for finding the inliers and the associated 3D transformation, taking advantage of the low degeneracy of the graph. Through experiments in urban driving scenarios, we show that our method outperforms a state of the art method both asymptotically and in practice.

1. INTRODUCTION

Efficient and reliable place recognition is one important challenge for enabling fully autonomous driving. With considerable changes in illumination occurring in driving scenarios, it is interesting to consider geometric information for performing place recognition. Autonomous vehicles are therefore often equipped with 3D time of flight sensors which permit a precise estimation of the road environment through the generation of point cloud maps.

One standard approach for recognizing places in 3D point cloud data is to compare a local map characterizing the vicinity of the vehicle, to a target map representing the full environment. This comparison can be done by extracting different basis elements such as keypoints [1], objects [2], shapes [3] or segments [4]. Place recognition is then performed by identifying correspondences between these basis elements and by verifying these correspondences for geometric consistency. This final 3D geometric verification step can be computationally expensive when working with large maps and with increasing number of correspondence candidates and outliers.

In this work, we formulate the problem of geometric verification as identifying a maximum clique in a consistency graph where edges connect correspondences that are geometrically consistent. A simplified example of a consistency graph is illustrated in Fig. 1c. We propose to perform the geometric verification by exploiting two important characteristics of the problem. First, we take advantage of the significant difference in size between the local and target maps by subdividing the latter into partitions and by performing geometric verifications only across relevant partitions. This approach not only effectively reduces the number of consistency tests but also guarantees the sparseness of the resulting consistency graph. In a second step, we exploit this sparsity by leveraging an efficient algorithm for detecting maximum cliques in sparse graphs.

The proposed approach is compared to a state of the art baseline method in urban driving scenario experiments which demonstrate that our approach is more efficient. We show through a simple example that the baseline method can return sub-optimal solutions whereas our method always identifies a maximum clique. A derivation of the complexity of our method is presented, demonstrating that it scales linearly with the size of the target map.

To summarize, this paper presents the following contributions:

- A novel partition-based algorithm for efficiently identifying maximum geometrically consistent sets of correspondences.
- An asymptotic complexity analysis of the proposed method.

This work was supported by the European Union’s Seventh Framework Programme for research, technological development and demonstration under the TRADR project No. FP7-ICT-609763.
• An evaluation of the proposed method based on experiments in urban driving scenarios.

The remainder of the paper is structured as follows: Section II provides an overview of the related work in the field of geometric verification for place recognition. Section III describes our partition-based approach to place recognition. The approach is evaluated in Section IV, and Section V finally concludes with a short discussion.

II. RELATED WORK

Numerous methods have been developed for performing global registration of partially overlapping 3D objects [5–7]. Unfortunately such methods cannot always be applied for performing place recognition in real-time, as the target map can be many orders of magnitude bigger than the local map and as the matching process can produce a high fraction of false correspondences.

Chen and Bhanu [8] propose to filter these outliers by clustering correspondences into geometrically consistent sets. Two correspondences \( c_i \) and \( c_j \) are called pairwise geometrically consistent if the difference of the Euclidean distance between the keypoints in the local map and in the target map is below a threshold \( \epsilon \), i.e. if

\[
|d_l(c_i, c_j) - d_t(c_i, c_j)| \leq \epsilon
\]  

(1)

where \( d_l(c_i, c_j) \) and \( d_t(c_i, c_j) \) are the keypoint distances in the local map and in the target map respectively.

This idea is employed in the recognition module of the Point Cloud Library (PCL) [9] where geometric consistencies are determined using a brute-force approach in which all possible correspondence pairs are checked for consistencies. This has an asymptotic complexity of \( O(n^3) \), where \( n \) is the number of correspondences. This approach is well suited for scenarios with a low amount of candidates and was employed in our previous work on segment-based place recognition [4]. However, it does not scale well to cases with a large number of candidates and outliers, e.g., when doing real-time place recognition in a large target map. We consider this method as our baseline in the experiments of Section IV.

Strategies for efficiently reducing the number of correspondence pairs have been proposed for stereo images and image retrieval. Ayache and Faverjon [10] describe a partitioning scheme for efficiently finding neighbor segments in stereo images. The SCRAMSAC method [11] performs RANSAC only on spatially consistent correspondences, i.e. correspondences that have a minimum fraction of matching neighbor features in both images. Both methods rely on assumptions about the disparity between images, thus their accuracy is influenced by the presence of high disparity and strong variation in viewing angles.

Graphical models have successfully been employed in the analogous task of recognizing places based on camera images [12, 13]. In the context of geometry-based place recognition, Fernandez-Moral et al. [3] propose to leverage graphical models for executing the geometric verification. An interpretation tree is used to match the local and target graphs where vertices represent planes and edges represent geometric relationships between these planes. Finman et al. [14] also propose a similar graph-matching strategy where vertices represent objects instead of planes. Contrastingly in our approach, vertices and edges respectively represent correspondences and geometric consistencies. Place recognition is then executed by extracting a maximum clique.

III. METHOD

This section presents our approach for performing geometric verification of correspondences in order to recognize places in 3D maps. We treat this task as a graph problem with the goal of identifying a maximum geometrically consistent set which is a set of maximum size consisting of correspondences that are all pairwise geometrically consistent. Pairwise geometrical consistency relationships are encoded in an undirected graph \( G = (V, E) \) where \( V = \{c_i\} \) is the set of correspondences \( c_i \) and \( E = \{e_{ij}\} \) is the set of undirected edges \( e_{ij} \) connecting all consistent pairs of correspondences \( (c_i, c_j) \). Once the graph is constructed, identifying a maximum geometrically consistent set is equivalent to finding a maximum clique of \( G \). An example of a consistency graph with its maximum consistent set is given in Fig. 1c.

A. Partition-based consistency graph construction

The naive approach for identifying all pairwise consistencies is testing all possible correspondence pairs. Here we take advantage of our knowledge about the sizes of the target and local maps to reduce the number of tests.

In many recognition applications, like place recognition as in our case, the local map is significantly smaller than the target map. Taking advantage of this information we can define a criterion for reducing the amount of consistency tests a priori. Let \( \epsilon \) be the tolerance for geometric consistency and \( b \) be the diameter of the bounding sphere of the local map keypoints, a pair of correspondences \( c_i \) and \( c_j \) can then only be consistent if

\[
d_i(c_i, c_j) \leq b + \epsilon \tag{2}\]

Urban driving scenarios usually extend along the earth’s surface only \((x, y)\) coordinates, but not vertically. Without loss of performance or correctness, we simplify the problem and consider only a bounding cylinder of diameter \( b \). The new relaxed criterion becomes

\[
d_{2D}^{1D}(c_i, c_j) \leq b + \epsilon \tag{3}\]

where \( d_{2D}^{1D}(c_i, c_j) \) computes the distance between the target map keypoints of \( c_i \) and \( c_j \) projected on the \( xy\)-plane.

We create a 2D grid partitioning of the correspondences according to the position of their keypoints in the target map, where each partition \( p_{u,v} \) has a square base with a side length of \( b + \epsilon \) and infinite height. An example of 2D grid partitioning is illustrated in Fig. 1b. Each correspondence \( c_i \) is assigned to its partition \( P(c_i) = p_{u,v} \),

\[
u = \left[ \frac{k_i(c_i) \cdot x - a \cdot x}{b + \epsilon} \right], v = \left[ \frac{k_i(c_i) \cdot y - a \cdot y}{b + \epsilon} \right] \tag{4}\]

1The z-axis is assumed to be roughly gravity aligned.
where $k_t(c_i)$ is the keypoint of $c_i$ in the target map and $o$ is the origin of the grid. A good choice of $o$ is the componentwise minimum of all $k_t(c_i)$.

With the chosen grid size $b + \epsilon$, it is guaranteed that the bounding cylinder of the model is always contained in a squared group of four adjacent partitions. Thus geometric consistency tests are necessary only on a set of candidate pairs of correspondences that is much smaller than $V × V$:

$$\{(c_i, c_j) \in V × V \mid i < j \land \exists u, v : c_i ∈ p_u, v \land c_j \in N(u, v)\}$$

(5)

where $N(u, v) := \bigcup_{l, m \in \{-1, 0, 1\}} p_{u + l, v + m}$. i.e. for a given $c_i \in V$ only correspondences in the partition of $c_i$ and the 8-neighbor-partitions need to be tested. Since consistency is a symmetric property, each pair is tested only once, i.e. if $i < j$. Other pairs of correspondences are ignored, since they cannot be consistent. The consistency graph is constructed as an adjacency list and contains all the geometrically consistent correspondence pairs $(c_i, c_j)$ as edges. Fig. 1a shows a set of 7 correspondences where all 21 pairs are tested for consistency. Applying this partitioning strategy reduces the number of tests to 14 (Fig. 1b).

The same approach can be used for cases where the environment extends into the third dimension. In this case, assuming that the local map is bounded by a sphere of diameter $b$, a 3D grid of cubes with size $b + \epsilon$ is used and consistency tests are performed over the 26-neighborhood of each partition.

B. Maximum clique detection

We consider a recognition to be successful in case the size of the detected maximum geometrically consistent set is greater than or equal to a threshold parameter $T$. Thus we need to identify a maximum $k$-clique with $k ≥ T$.

A second advantage of enforcing the partitioning constraints in situations where the local map is significantly smaller than the target map is that the sparseness of the consistency graph is guaranteed. With this knowledge we can rely on a class of $k$-clique detection algorithms that visit the graph in degeneracy order to find the maximum geometrically consistent set. A graph $G$ can be characterized by its degeneracy (or $k$-core number) $d$, which is the smallest number so that every subgraph of $G$ contains a vertex with degree $\leq d$. Each graph has a degeneracy ordering, an order in which each vertex has at most $d$ vertices that come later in the ordering.

We use a generalization of the maximal clique listing approach described in [15, 16] to find a maximum clique of $G$. An outline of the algorithm is presented in Algorithm 1. First, we use the bucket sort algorithm for sorting the vertices in increasing degree order. This is described in detail in [17] and can be performed in $O(|V|)$. Then we iterate on the vertices according to the found order. At each step, the function CLIQUE is called using the current vertex of minimum degree as input. If the function returns a clique that is bigger than the current maximum clique max_clique, the new maximum clique is stored. The vertex is then removed from the graph together with its incident edges and the vertex ordering is updated in $O(d)$. The resulting degeneracy ordering in the visit of the vertices guarantees that at each step $v$ has at most $d$ neighbors, bounding the computational load on CLIQUE.

CLIQUE can be any function that returns the biggest clique $C \subseteq G$ such that $|C| ≥ T$ and $v ∈ C$ (or the empty set, if such a clique does not exist). Instances of this function can be any clique detection approach like the branch-and-bound method presented in [18] or successive refinements like [19]. For simplicity, in this work we use a method based on heavy pruning strategies [20]. This algorithm builds a subgraph containing $v$ and its neighbors and recursively visits its subgraphs to determine if a clique of the required size can be built. Vertex degrees are constantly checked to discard fruitless candidates as soon as possible in the recursion.

Algorithm 1 Outline of the algorithm for maximum clique detection. Iterating over the vertices in degeneracy order guarantees that $v$ always has at most $d$ neighbors.

1: procedure MAXIMUM CLIQUE(G, T)
2: max_clique ← 0
3: sorted_vertices ← SORT BY DEGREE(G)
4: while |sorted_vertices| ≠ 0 do
5: $v ←$ sorted_vertices[0]
6: $C = \text{CLIQUE}(v, G, T)$
7: if $|C| ≠ 0$ then
8: max_clique ← $C$
9: $T ← |C| + 1$
10: end if
11: REMOVE VERTEX(G, v)
12: UPDATE ORDER(G, sorted_vertices)
13: end while
14: return max_clique
15: end procedure

C. Recognition

The condition on the size of the maximum geometrically consistent set can be removed in case it is safe to assume that a true recognition always exists. This is however not always possible, e.g. in loop-closure detection applications. Once an acceptable maximum clique is identified, the relative transformation between the local and target maps can be found using any rigid transformation estimation method. Here we use the least-squares approach described in [21].

D. Asymptotic complexity and scaling

In the consistency graph $G = (V, E)$, let the parameters $n = |V|$ be the number of correspondences and $d$ be the degeneracy of the graph. Assuming that outliers are uniformly distributed in the target map and are present in a high ratio over inliers, we can say that $n$ is proportional to the size of the target map. The average number of correspondences per partition $n_p$ and $d$ are considered constant as they depend on fixed parameters like the density.
of objects in the environment, the size of the local map and the correspondence matching strategy.

Partitioning the correspondences is trivial and can be done in \(O(n)\). All the necessary consistency tests can be performed in \(O(n_p n)\) since each of the \(n_p\) partitions requires \(O(n^2)\) tests. The bucket sort is done in \(O(n)\). Since every vertex visited in the loop has at most \(d\) neighbors, each of the \(n\) calls to CLIQUE has a worst case complexity of \(O(d!)\), thus the complexity of the code in Algorithm 1 is \(O(d! n)\). Although the \(d!\) term may appear like an important bottleneck, any reasonable implementation for CLIQUE usually performs better. As shown in Table 1, maximum clique detection brings a minimal contribution to the total runtime. The estimation of the 3D transformation scales linearly with the number of elements in the geometrically consistent set, thus it is bound by \(O(d)\).

The total complexity of our method is \(O(n (d! + n_p))\). Under the homogeneity assumption, since \(d\) and \(n_p\) are parameters that do not depend on the target map’s size, we can conclude that the performance of our algorithm scales linearly with the size of the target map.

### IV. EXPERIMENTS

We tested our method in the place recognition module of our online LiDAR-based localization and mapping system described in [22]. We compare our new method with our current baseline approach implemented in the PCL (pcl::GeometricConsistencyGrouping). Both implementations are benchmarked in two different configurations:

- **Localization**: An autonomous vehicle drives in a known urban scenario, continuously trying to localize inside a static target map.
- **Loop-closure**: The vehicle explores an unknown urban scenario, continuously updating a dynamic target map and trying to detect loop-closures.

Each configuration uses a different dataset of the KITTI collection [23], both datasets have similar sizes. At each call we measure the runtime of the recognition. We use \(\varepsilon = 0.4\)m as consistency threshold and \(T = 6\) as minimum consistent set size.

Figure 2 shows a runtime comparison between our method and the PCL implementation. In the localization test, our new method is 6.57 times faster than the reference method. This improvement comes from two main factors. First, only 32.6% of the consistency tests are performed. In addition, our partitioned approach enables better cache locality when accessing correspondence keypoints, increasing the performance of the consistency test functions.

In the loop-closure setting the speedup decreases to 2.05x. This is due to the fact that during the first part of the experiment the target map is relatively small and prevents partitioning from being effective. In general we observe that partitioning successfully accelerates the recognition process by reducing the number of tested correspondence pairs and our method scales better as the size of the map increases. In future work the method should be tested with bigger maps in order to prove the theorized linear scaling of the method (see Section III-D).

<table>
<thead>
<tr>
<th>Step</th>
<th>Mean runtime (Localization)</th>
<th>Mean runtime (Loop-Closure)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>0.06 ± 0.01ms</td>
<td>0.03 ± 0.02ms</td>
</tr>
<tr>
<td>Graph construction</td>
<td>13.86 ± 7.95ms</td>
<td>8.09 ± 9.70ms</td>
</tr>
<tr>
<td>Max clique detection</td>
<td>0.13 ± 0.08ms</td>
<td>0.17 ± 0.23ms</td>
</tr>
<tr>
<td>Transformation estimation</td>
<td>&lt; 0.01ms</td>
<td>&lt; 0.01ms</td>
</tr>
<tr>
<td>Total (Our method)</td>
<td>14.31 ± 8.16ms</td>
<td>8.52 ± 9.98ms</td>
</tr>
<tr>
<td>Total (PCL)</td>
<td>94.23 ± 56.28ms</td>
<td>17.49 ± 32.10ms</td>
</tr>
<tr>
<td>Speedup</td>
<td>6.57x</td>
<td>2.05x</td>
</tr>
</tbody>
</table>

TABLE 1: Runtimes and speedups of the different steps of our algorithm. Clearly the most important factor is the construction of the consistency graph.

Table 1 shows the time needed by each step of our method. While partitioning quickly and effectively reduces the number of consistency tests required, the construction of the consistency graph is still the most expensive operation. Detecting a maximum clique is a relatively cheap task thanks to the sparseness of the graph and to the traversal strategy that bounds the complexity of the search.

We performed additional tests to determine the quality of the resulting recognition in the two methods. As metric for the quality we use the size of the identified maximum consistent set. Our method uses an exact algorithm, and is always able to find a consistent set of maximum size. On the other side, the PCL implementation uses a greedy approach and occasionally fails to detect the optimal solution. Fig. 3 shows a tested pathological case: depending on the order of the input correspondences the greedy approach can detect a maximum (3a, in blue) or a maximal (3b, in red) consistent set. Our method detects the maximum clique in both cases. Note that the PCL implementation does not explicitly build a graph, but detects the same pairwise consistency, thus we can use the same visualization.
Fig. 3: Detection of consistent sets with $T = 3$ in two graphs based on the same correspondences stored in different orders, indicated by the numbers. The PCL implementation identifies the blue biggest cluster in (a), but can only find a suboptimal solution (in red) if the correspondences are stored differently (b). Our implementation uses exact $k$-clique detection and finds a maximum consistent set in all cases.

V. CONCLUSION

In this work, we presented a new graph-based method for place recognition in 3D point clouds that improves on our reference baseline in terms of performance. We stated the recognition task as a graph problem and used a novel partitioning approach to significantly reduce the number of consistency tests required. Taking advantage of the sparseness of the consistency graph, we use a clique detection algorithm to identify the biggest set of geometrically consistent correspondences quickly and exactly.

Experiments in urban driving scenarios show that our method performs better than the greedy approach of our selected baseline. We theoretically demonstrate that the runtime of this algorithm scales linearly with the size of the map, enabling fast localization in large environments. Benchmark results show that the present bottleneck of the method is the construction of the consistency graph. In future work we will explore other approaches to further accelerate the construction task. Moreover we would like to evaluate our method on bigger scenes to prove the theorized linear scaling behavior.

REFERENCES