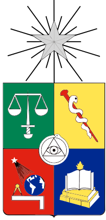


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

AN IMPROVED FEATURE EXTRACTOR FOR THE LIDAR ODOMETRY AND MAPPING ALGORITHM

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA MENCIÓN ELÉCTRICA

CLAYDER ALEJANDRO GONZALEZ CADENILLAS

PROFESOR GUÍA:
MARTIN ADAMS

MIEMBROS DE LA COMISIÓN:
JAVIER RUIZ DEL SOLAR SAN MARTÍN
MIGUEL TORRES TORRITI

SANTIAGO DE CHILE
2019

RESUMEN DE LA TESIS PARA OPTAR
AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA MENCIÓN ELÉCTRICA
POR: CLAYDER ALEJANDRO GONZALEZ CADENILLAS
FECHA: 2019
PROF. GUÍA: SR. MARTIN ADAMS

AN IMPROVED FEATURE EXTRACTOR FOR THE LIDAR ODOMETRY AND MAPPING ALGORITHM

Feature extraction is a critical task in feature-based Simultaneous Localization and Mapping (SLAM), which is one of the most important problems in the robotics community. An algorithm that solves SLAM using LiDAR based features is the LiDAR Odometry and Mapping (LOAM) algorithm. This algorithm is currently regarded as the best performing SLAM algorithm according to the KITTI Vision Dataset Benchmark.

The LOAM algorithm solves the SLAM problem through a feature matching approach and its feature extraction algorithm detects features classifying points of a point cloud as flat or sharp. This classification results from an equation that defines the level of smoothness for each point. However, this equation does not consider the range noise of the sensor. Therefore, if the LiDAR's range noise is high, LOAM's feature extractor could confuse flat and sharp points, causing the feature matching task to fail.

This thesis proposes the replacement of the original LOAM's feature extraction algorithm with the Curvature Scale Space (CSS) algorithm. The choice of this algorithm was made after studying various feature extractors in the literature. The CSS algorithm can potentially improve the feature extraction task in noisy environments because of its various levels of Gaussian smoothing. The replacement of the original LOAM's feature extractor with the CSS algorithm was achieved by adapting the CSS algorithm to the Velodyne VLP-16 3D LiDAR.

The LOAM's feature extractor and CSS feature extractor were tested and compared with simulated and real data including the KITTI Vision Benchmark Dataset using the Optimal Sub-Pattern Assignment (OSPA) and Absolute Trajectory Error (ATE) metrics. For all these datasets the CSS feature extraction performance was better than that of the LOAM algorithm in terms of the OSPA and ATE metric values.

RESUMEN DE LA TESIS PARA OPTAR
AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA MENCIÓN ELÉCTRICA
POR: CLAYDER ALEJANDRO GONZALEZ CADENILLAS
FECHA: 2019
PROF. GUÍA: SR. MARTIN ADAMS

AN IMPROVED FEATURE EXTRACTOR FOR THE LIDAR ODOMETRY AND MAPPING ALGORITHM

La extracción de características es una tarea crítica en la localización y mapeo simultáneo o Simultaneous Localization and Mapping (SLAM) basado en características, que es uno de los problemas más importantes de la comunidad robótica. Un algoritmo que resuelve SLAM utilizando características basadas en LiDAR es el algoritmo LiDAR Odometry and Mapping (LOAM). Este algoritmo se considera actualmente como el mejor algoritmo SLAM según el Benchmark KITTI.

El algoritmo LOAM resuelve el problema de SLAM a través de un enfoque de emparejamiento de características y su algoritmo de extracción de características detecta las características clasifican los puntos de una nube de puntos como planos o agudos. Esta clasificación resulta de una ecuación que define el nivel de suavidad para cada punto. Sin embargo, esta ecuación no considera el ruido de rango del sensor. Por lo tanto, si el ruido de rango del LiDAR es alto, el extractor de características de LOAM podría confundir los puntos planos y agudos, lo que provocaría que la tarea de emparejamiento de características falle.

Esta tesis propone el reemplazo del algoritmo de extracción de características del LOAM original por el algoritmo Curvature Scale Space (CSS). La elección de este algoritmo se realizó después de estudiar varios extractores de características en la literatura. El algoritmo CSS puede mejorar potencialmente la tarea de extracción de características en entornos ruidosos debido a sus diversos niveles de suavizado Gaussiano. La sustitución del extractor de características original de LOAM por el algoritmo CSS se logró mediante la adaptación del algoritmo CSS al Velodyne VLP-16 3D LiDAR.

El extractor de características de LOAM y el extractor de características de CSS se probaron y compararon con datos reales y simulados, incluido el dataset KITTI utilizando las métricas Optimal Sub-Pattern Assignment (OSPA) y Absolute Trajectory Error (ATE). Para todos estos datasets, el rendimiento de extracción de características de CSS fue mejor que el del algoritmo LOAM en términos de métricas OSPA y ATE.

A Dios
A Claudia, mi esposa
A mi familia
A los que me dieron ánimos en este largo trayecto

Acknowledgments

I would like to thank professor Martin Adams for giving me the opportunity to learn about robotics. His kind and strict manner helped me to mature as a professional. This thesis has helped me to understand more deeply how scientific research works in real life.

Contents

List of Tables xiii

List of Figures xiv

1 Introduction 1
- 1.1 Hypothesis 4
- 1.2 Objectives 5
  - 1.2.1 General Objective 5
  - 1.2.2 Specific Objectives 5
- 1.3 Contributions 5
- 1.4 Document Structure 5

2 Literature Review 6
- 2.1 Feature Extraction 6
- 2.2 General Feature Extraction in Point Clouds 7
  - 2.2.1 Harris Corner Detector 7
    - 2.2.1.1 Harris Corner Basics 8
    - 2.2.1.2 Harris Corners in 3D Point Clouds 9
  - 2.2.2 Scale Invariant Features Transform (SIFT) 10
    - 2.2.2.1 SIFT Basics 10
    - 2.2.2.2 SIFT in 3D Point Clouds 12
  - 2.2.3 Intrinsic Shape Signatures (ISS) 14
  - 2.2.4 Why a general feature extractor is not suitable for LOAM? 15
- 2.3 Feature Extraction algorithms commonly used in SLAM 17
  - 2.3.1 RANSAC 17
  - 2.3.2 Hough Transform 17
  - 2.3.3 Split-and-Merge 19
  - 2.3.4 Adaptive Curvature Estimation 20
  - 2.3.5 Curvature Scale Space 22
  - 2.3.6 Why not apply other feature extractors in LOAM? 25
- 2.4 Adaptive Breakpoint Detector 27
- 2.5 LiDAR Odometry and Mapping (LOAM) 30
  - 2.5.1 LiDAR Odometry Estimation 31
    - 2.5.1.1 LOAM's Feature Extraction algorithm 31
    - 2.5.1.2 Feature Point Correspondence 31
    - 2.5.1.3 Motion Estimation 33

2.5.1.4 LiDAR Odometry Algorithm . . . 33
2.5.2 Why the LOAM's Feature Extractor can be improved? . . . 34
2.6 Summary . . . 39

3 Methodology and Implementation **40**
3.1 Curvature Scale Space with Sparse Point Cloud Data: The New Feature Extractor of the LOAM Algorithm . . . 40
3.2 Implementation of the Curvature Scale Space algorithm with 3D Sparse Point Cloud Data . . . 42
3.3 Summary . . . 52

4 Results **53**
4.1 The Optimal Sub-Pattern Assignment (OSPA) . . . 53
4.2 The Absolute Trajectory Error (ATE) . . . 54
4.3 Simulated Data . . . 54
4.3.1 Scenario with the LiDAR stationary . . . 55
4.3.2 Scenario with robot in motion . . . 60
4.4 Real Data . . . 69
4.5 KITTI Dataset . . . 73

5 Conclusions **88**

6 Bibliography **90**

A KITTI Dataset Experiments **94**

List of Tables

2.1 The computational complexities of well known point cloud feature extractors. 16
2.2 P. Nuñez et al. [26] Feature extraction algorithms comparison. 25
2.3 Elements of point cloud $\mathcal{P}_1$. 35
2.4 Elements of point cloud $\mathcal{P}_2$. 36
2.5 Elements of point cloud $\mathcal{P}_3$. 36
2.6 Elements of point cloud $\mathcal{P}_4$. 37
2.7 Elements of point cloud $\mathcal{P}_5$. 38

3.1 Laser IDs and their angles. 43

List of Figures

1.1 Part of the sequence number 05 of the KITTI Dataset. 3

2.1 Computation of Difference of Gaussian images in the scale space. 12
2.2 Processing time required by various feature extractors as a function of feature radius. 16
2.3 RANSAC in an indoor environment [31]. 18
2.4 Polar representation of line $\mathcal{L}$. 18
2.5 Split-and-Merge graphical representation [7]. 20
2.6 Adaptive Curvature Estimation graphical representation [26]. 21
2.7 Example of a real point cloud segment obtained with a VLP-16 LiDAR. . . . 27
2.8 Adaptive breakpoint detector (adapted from [6]). 28
2.9 Block diagram of the LOAM software system [42]. 30
2.10 Sharp feature correspondence. 32
2.11 Flat feature correspondence. 32
2.12 Point Cloud $\mathcal{P}_1$ and its smoothness c. 35
2.13 Point Cloud $\mathcal{P}_2$ and its smoothness c. 36
2.14 Point Cloud $\mathcal{P}_3$ and its smoothness c. 37
2.15 Point Cloud $\mathcal{P}_4$ and its smoothness c. 38
2.16 Point Cloud $\mathcal{P}_5$ and its smoothness c. 38

3.1 VLP-16 hardware configuration. 41
3.2 Diagram of the methodology used to compare the original LOAM algorithm and the LOAM algorithm with the CSS feature extractor. 42
3.3 Complete point cloud taken with a Velodyne VLP-16 in an indoor environment (each scan is shown with a different color). 44
3.4 Closer view of part of the point cloud of figure 3.3. 45
3.5 Segmentation of one scan using the Adaptive Breakpoint Detector. 46
3.6 2D projection of the point cloud segment $\mathcal{S}_1$. 47
3.7 2D projection of the point cloud segment $\mathcal{S}_1$ (Figure 3.6) at various levels of Gaussian smoothing. 49
3.8 Curvature functions κ at various levels of the smoothing for segment $\mathcal{S}_1$. . . 50
3.9 Curvature scale space created with the functions κ shown in Figure 3.8. . . . 51

4.1 Scenario with the LiDAR stationary. 55
4.2 True positives vs false positives (ROC curve) for three σ_r evaluated with the LOAM's feature extractor and the CSS feature extractor. 57
4.3 Standard deviation in x of estimated features in 50 sweeps. 58

4.4 Standard deviation in y of estimated features in 50 sweeps. 58
4.5 Estimated features per sweep extracted with the LOAM's feature extractor and the CSS feature extractor in 50 sweeps. 59
4.6 False positives features per sweep extracted with the LOAM's feature extractor and the CSS feature extractor in 50 sweeps. 59
4.7 Scenario with the robot in motion. 60
4.8 Feature estimation error using the OSPA metric ($\sigma_r = 0.01$ m). 61
4.9 Feature estimation error using the OSPA metric ($\sigma_r = 0.02$ m). 61
4.10 Feature estimation error using the OSPA metric ($\sigma_r = 0.03$ m). 62
4.11 Estimated trajectories compared to ground-truth using simulated data ($\sigma_r = 0.01$ m). 63
4.12 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using simulated data ($\sigma_r = 0.01$ m). 64
4.13 Estimated trajectories compared to ground-truth using simulated data ($\sigma_r = 0.02$ m). 65
4.14 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using simulated data ($\sigma_r = 0.02$ m). 66
4.15 Estimated trajectories compared to ground-truth using simulated data ($\sigma_r = 0.03$ m). 67
4.16 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using simulated data ($\sigma = 0.03$ m). 68
4.17 Error percentages per each range noise standard deviation value. 69
4.18 Satellite image of Universidad de Chile - campus Beauchef. 70
4.19 Estimated trajectories compared to ground-truth using real data. 71
4.20 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using real data. 72
4.21 Estimated trajectories compared to ground-truth in the sequence 00. 75
4.22 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 00. 76
4.23 Number of features in the sequence 00. 77
4.24 Estimated trajectories compared to ground-truth in the sequence 05. 78
4.25 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 05. 79
4.26 Number of features in the sequence 05. 80
4.27 Estimated trajectories compared to ground-truth in the sequence 06. 81
4.28 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 06. 82
4.29 Number of features in the sequence 06. 83
4.30 Estimated trajectories compared to ground-truth in the sequence 09. 84
4.31 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 09. 85
4.32 Number of features in the sequence 09. 86
4.33 Quantity of dropped point clouds, $\mathcal{P}_k$, per sequence. 87

A.1 Estimated trajectories compared to ground-truth in the sequence 00. 95
A.2 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 00. 96

A.3 Number of features in the sequence 00. 97
A.4 Estimated trajectories compared to ground-truth in the sequence 01. 98
A.5 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 01. 99
A.6 Number of features in the sequence 01. 100
A.7 Estimated trajectories compared to ground-truth in the sequence 02. 101
A.8 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 02. 102
A.9 Number of features in the sequence 02. 103
A.10 Estimated trajectories compared to ground-truth in the sequence 03. 104
A.11 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 03. 105
A.12 Number of features in the sequence 03. 106
A.13 Estimated trajectories compared to ground-truth in the sequence 04. 107
A.14 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 04. 108
A.15 Number of features in the sequence 04. 109
A.16 Estimated trajectories compared to ground-truth in the sequence 05. 110
A.17 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 05. 111
A.18 Number of features in the sequence 05. 112
A.19 Estimated trajectories compared to ground-truth in the sequence 06. 113
A.20 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 06. 114
A.21 Number of features in the sequence 06. 115
A.22 Estimated trajectories compared to ground-truth in the sequence 07. 116
A.23 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 07. 117
A.24 Number of features in the sequence 07. 118
A.25 Estimated trajectories compared to ground-truth in the sequence 08. 119
A.26 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 08. 120
A.27 Number of features in the sequence 08. 121
A.28 Estimated trajectories compared to ground-truth in the sequence 09. 122
A.29 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 09. 123
A.30 Number of features in the sequence 09. 124
A.31 Estimated trajectories compared to ground-truth in the sequence 10. 125
A.32 Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 10. 126
A.33 Number of features in the sequence 10. 127

Chapter 1

Introduction

Feature extraction is an important task in many areas such as machine learning, pattern recognition, image processing and robotics. It is necessary in some solutions for Simultaneous Localization and Mapping (SLAM), which is one of the most important problems in the robotics community [4] [3] [39]. In feature-based SLAM solutions, feature extraction involves extracting reliable features, also known as landmarks. These features can be extracted with a variety of sensors. A frequently used sensor for this task is LiDAR (Light Detection and Ranging), the data from which can be processed to attain a set of points called point clouds. We can divide point cloud feature extractors into two groups: General point cloud feature extractors and feature extractors commonly used in SLAM.

General point cloud feature extractors are algorithms that process any kind of point cloud and are not only used in SLAM. Among them, there are algorithms such as the Harris corner detector, the Scale Invariant Feature Transform (SIFT) keypoint detector and the Intrinsic Shape Signatures (ISS) detector. The Harris corner detector [12] is an algorithm which has its basis in the image processing community, however, it has also been adapted to work with point clouds. In the point cloud version, the algorithm processes surface normals rather than image derivatives. The SIFT key-point detector [20] also has its origin in the image processing community. The main difference between the Harris and SIFT detectors is that instead of processing surface normals, the SIFT algorithm processes the intensity values of the LiDAR. This is done by convolving the intensity values of the LiDAR with a Gaussian kernel and then obtaining the "difference of Gaussians", which can be represented as the difference of various point clouds convolved with scaled Gaussian kernels. Also, the ISS algorithm [40] is based on the Eigen value decomposition of a covariance matrix of a certain spherical neighborhood of a point cloud. According to F. Silvio [8], among these algorithms, the one that has the best performance in terms of repeatability of detected features is the ISS algorithm. Its use in SLAM, however, is computationally expensive, making it difficult to process the data from the sensors in real SLAM scenarios.

Alternatively, there are various feature extractors which have been derived specifically for feature-based SLAM and they are commonly used in this task with a variety of these algorithms existing in the literature. Previously, the use of 2D LiDARs produced many feature extraction algorithms, examples that include [1] [38] [5], which are algorithms which

extract edges, lines and corners. Currently, 2D LiDARs are being replaced by 3D versions. 3D LiDAR sensors can be divided into two categories: those which return a dense point cloud and those which return a sparse point cloud. We call sparse point clouds those that have some points with a considerable space in some part of them. An example of these point clouds are those returned by Velodyne LiDARs, which are mainly formed of N 2D LiDARs. These 2D LiDARs are located with a certain angle of inclination between them, which make the point cloud sparse in the vertical direction. Algorithms that are specially designed for this type of point cloud are those which extract 3D lines [34], corners based on the Kanade-Tomasi algorithm [18] and vertical corners [14].

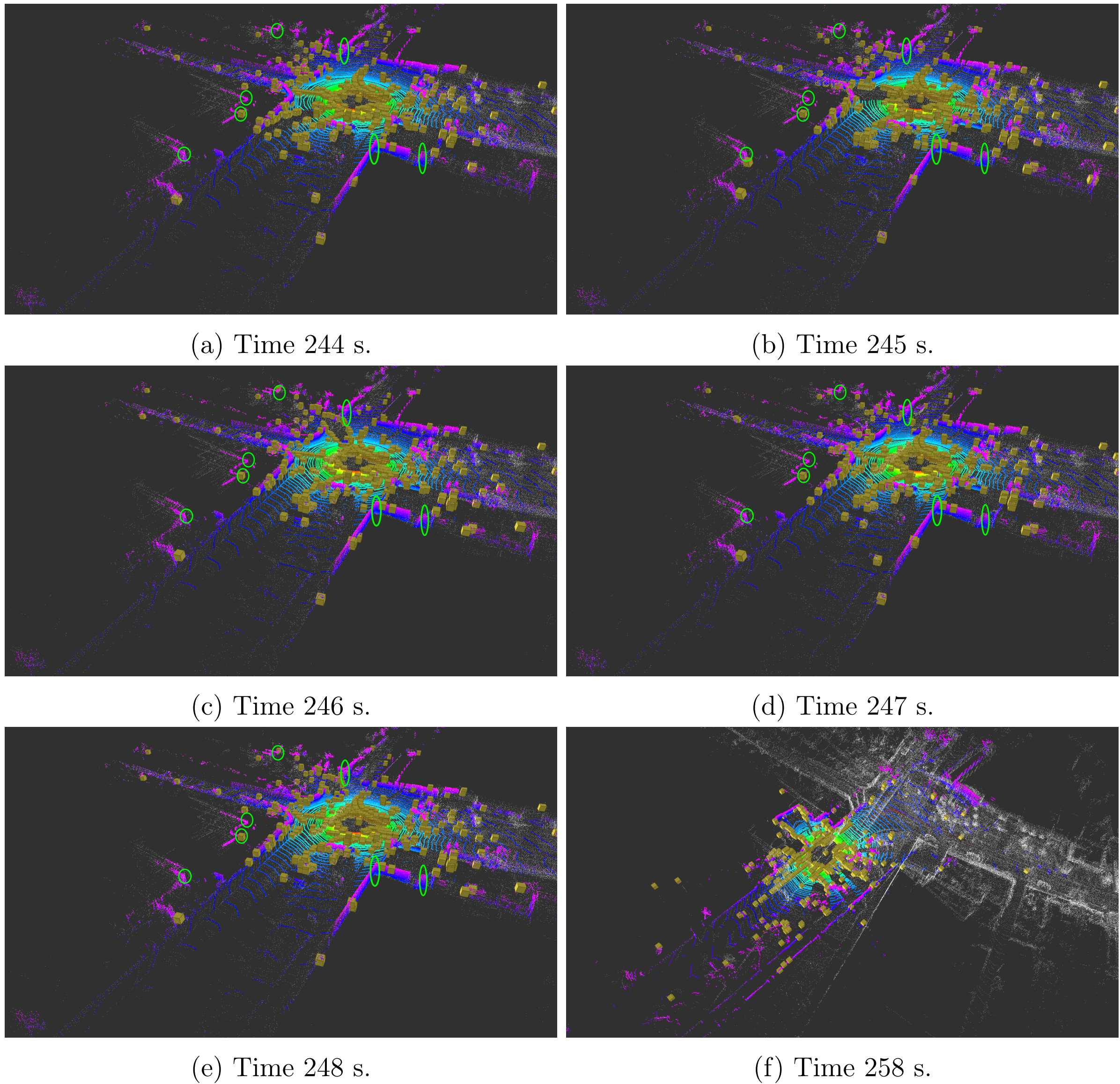
In addition to the feature extractors mentioned previously, there are other algorithms commonly used in SLAM that extract lines such as RANSAC [9], the Hough Transform [13] and the Split and Merge algorithm [27]. V. Nguyen et al. [25] made a comparison of these algorithms and concluded that the Split and Merge algorithm has the best performance in the number of lines extracted. In the feature extractor of P. Núñez et al. [26], an algorithm to extract edges, lines, corners and circular arcs is proposed based on adaptive curvature estimation. Its performance was compared with the Hough Transform, Split and Merge and Curvature Scale Space (CSS) [22] algorithms. It was concluded that their proposed algorithm had the overall best performance (recognition rate, accuracy of feature locations and processing time of the algorithm), however, the Curvature Scale Space had a better recognition rate even though it is more computationally complex.

Although there are several feature extractors, there is not a specific feature extractor that is widely used to solve the feature-based SLAM problem. Most of the time, each solution for feature-based SLAM proposes its own feature extractor. This is the case of the LiDAR Odometry and Mapping (LOAM) algorithm, which is the highest ranked algorithm in terms of SLAM performance in the KITTI Visual Odometry Benchmark. The LOAM algorithm solves the SLAM problem through a feature matching approach and its feature extraction algorithm detects features classifying points as being either smooth or non-smooth in a same scan[1] according to a threshold.

One issue that the LOAM's feature extractor has is that it does not consider the range noise of the sensor. This issue could make the feature extractor confuse smooth or non-smooth points if the noise level is high. Another issue is related to the processing time of the feature matching approach. Since this approach uses the Levenberg-Mardquat optimization algorithm to match the extracted features, the processing time for the whole LOAM algorithm also depends on the number of features extracted. Hence, a large quantity of features could make the LOAM algorithm not process some 3D scans, because it needs to finish the scan matching task before a new 3D scan arrives. Both of these issues occur in figure 1.1, which shows times 244 s, 245 s, 246 s, 247 s, 248 s and 258 s of the sequence 05 of the KITTI Dataset. In this figure, the current points of the scans of the LiDAR are represented in colors from green to purple. Yellow boxes represent non-smooth features and the estimated map is represented in gray color. From figures 1.1a to 1.1e, the robot maintains its position and some non-smooth points are miss-detected (represented with green ellipses). In Figure 1.1f, the robot moved straight forward resulting in the estimated map represented as the the gray

[1] In the case of the Velodyne VLP-16, which is the LiDAR used in this work, a scan is a subset of points returned at a single elevation angle by one of the 16 lasers.

point cloud. It should be noted that this estimated map does not correspond well with the actual 3D scan shown in colors.



(a) Time 244 s. (b) Time 245 s.

(c) Time 246 s. (d) Time 247 s.

(e) Time 248 s. (f) Time 258 s.

Figure 1.1: Part of the sequence number 05 of the KITTI Dataset.

The results in Figure 1.1 were achieved by using the original LOAM code[2] published in the ROS Indigo website. It is important to note that this code version is not the same as that which produced the results in the KITTI vision benchmark. This code version was developed for raw data and it was used because it is the most reliable version available since the original author removed his code from his repository[3]. Given that the code used was developed for raw data, the previous example was configured to work as raw data. In the KITTI vision benchmark website, the LOAM algorithm produces a low error rate in all sequences, including the sequence 05 presented in the previous example. However, the code used to produce those results may have some changes compared to the version used in this example, additionally

[2] `http://docs.ros.org/indigo/api/loam_velodyne/html/files.html`

[3] `https://github.com/jizhang-cmu/loam_velodyne`

that code does not consider the sampling frequency in which the LiDAR returns the data.

To solve the confusion of smooth and non-smooth features and the errors in the estimated map, seen in Figure 1.1, we propose to replace the original LOAM feature extraction algorithm with the Curvature Scale Space (CSS) algorithm. The choice of this algorithm was made after studying various feature extractors in the literature such as the SIFT, Harris, ISS, RANSAC, Hough transform, Adaptive Curvature Estimation and CSS algorithms. Additionally, the replacement was carried out based on the evidence provided in the articles by V. Nguyen et al. [25] and P. Nuñez et al. [26], in which the CSS algorithm demonstrated itself to perform well. Therefore, the CSS algorithm can potentially improve the feature extraction task in environments where the range noise is high because of its various levels of Gaussian smoothing. Additionally, this replacement can speed up the feature matching task because of the few, but precise features extracted by this algorithm. The replacement of the original LOAM feature extractor with the CSS algorithm was achieved by adapting the CSS algorithm to the Velodyne VLP-16 3D LiDAR.

The LOAM feature extractor and CSS feature extractor were tested and compared with both simulated and real data including the KITTI vision benchmark dataset using the Optimal Sub-Pattern Assignment (OSPA) [33] and Absolute Trajectory Error (ATE)[4] metrics. Large differences occurred when testing both of the algorithms when additional range noise was added to the simulated data. The mean OSPA error of the CSS feature extractor reached a value equal to 3.94 meters, while the mean OSPA error of the LOAM's feature extractor was 4.58 meters[5]. In the case of the ATE metric, the mean error of the CSS feature extractor reached a value equal to 0.06 meters, while the mean error of the LOAM feature extractor was 0.12 meters. Another significant difference occurred with the KITTI dataset. In this experiment, the LOAM algorithm produced high ATE errors in many sequences. This problem occurred because the number of LOAM features was very high, which made the task of feature matching computationally complex. This complexity made that some point clouds were not being processed because of the sampling frequency in which the LiDAR was returning the point clouds. This issue did not occur with the CSS feature extractor, because the number of features was lower and the feature matching task could be completed before the arrival of a new point cloud. These results allow us to conclude that the CSS feature extractor is robust and suitably resistant to range noise and can speed up the feature matching task. Therefore, it is a suitable feature extractor for the LOAM algorithm.

1.1 Hypothesis

The use of the Curvature Scale Space algorithm as an alternative to the feature extractor in the LOAM algorithm will allow the extraction of more repeatable and robust features. This will reduce both localization and feature map errors, which will be quantifiable via the use of the OSPA and ATE metrics .

[4] `http://www.rawseeds.org/`

[5] The OSPA metric cut off (c) and power (p) parameters were 5 and 2, respectively

1.2 Objectives

1.2.1 General Objective

The main objective of this thesis is to implement a feature extractor for SLAM that can improve the results obtained by the LiDAR Odometry and Mapping algorithm, which is a primary state of the art SLAM algorithm.

1.2.2 Specific Objectives

The specific objectives of this thesis are to:

1. Study the most important feature extraction algorithms in the literature and choose the one expected to yield the best results with Velodyne 3D LiDARs.
2. Adapt and implement the chosen feature extraction algorithm for Velodyne 3D LiDAR data.
3. Replace the feature extractor in the original implementation of the LOAM algorithm with the chosen feature extraction algorithm.
4. Test the implementation with simulated, real data and the KITTI dataset.

1.3 Contributions

The contributions of this thesis are:

1. An adaptation and implementation of the Curvature Scale Space algorithm for Veloyne 3D LiDAR data.
2. An improvement of the original LOAM algorithm in terms of trajectory estimation using the KITTI Dataset.

1.4 Document Structure

The remainder of this thesis is organized as follows. In Chapter 2, the basic concepts of feature extraction and the most important algorithms in this area are presented. Also, a discussion of which feature extractor could be the most suitable in the LOAM algorithm is given. In Chapter 3, the adaptation and implementation of the Curvature Scale Space (CSS) algorithm is explained in detail. In Chapter 4, the results of the implementation are shown for simulated data, real data taken from the surroundings of the University of Chile and the KITTI Dataset. Finally, in Chapter 5, the results are discussed, and possible future work is suggested.

Chapter 2

Literature Review

This chapter presents the state of the art related to this thesis. This part is important for achieving the general objective of this thesis, which is to replace the LiDAR Odometry and Mapping (LOAM)'s feature extractor in order to improve its results. We analyzed the main feature extractors for point clouds in the literature. This analysis is useful to evaluate if an existing feature extractor could fit in the LOAM algorithm or if it is necessary to implement a new feature extraction algorithm. Hence, the objective of this chapter is to introduce readers to the theory of the main feature extraction algorithms used in point clouds and to justify the final choice of feature extractor.

This chapter is structured as follows. Section 2.1 defines the feature extraction task. Section 2.2 describes general feature extraction algorithms used for point clouds. Section 2.3 describes the main feature extraction algorithms used in SLAM. Section 2.4 describes the algorithm used in this thesis to segment point clouds. Finally section 2.5 describes the LOAM algorithm emphasizing its feature extractor.

2.1 Feature Extraction

Feature extraction is a general term used in many fields such as machine learning, pattern recognition, computer vision and robotics. In robotics, features are dependent on the environment where a robot is able to maneuver. For example, when using a LiDAR in indoor environments, features could correspond to doors, walls and corners, while in outdoor environments features could correspond to tree trunks or lamp posts.

According to S. Thrun [39], we can denote a feature extractor as a function $f(z_t)$ in which z_t are range measurements obtained in time t. Commonly, feature extraction algorithms return only certain points from the whole measurement. In this thesis, a 3D LiDAR is used and the whole measurement obtained can be grouped in a special data type called a *point cloud*. A point cloud can be represented as a set of points with 3 Euclidean components (x, y, z).

From the whole point cloud, a feature extractor returns a sub-set of points called features. This sub-set, $\mathcal{F}$, contains vectors with 3 Euclidean components (x, y, z), each one corresponding to a feature. The position of the ith feature at time t, $f_{(i,t)}$ can be represented as

$$\mathcal{F} = f(z_t) = \{f_{(1,t)}, f_{(2,t)}, ..., f_{(n,t)}\} = \{\begin{pmatrix} x_{(1,t)} \\ y_{(1,t)} \\ z_{(1,t)} \end{pmatrix}, \begin{pmatrix} x_{(2,t)} \\ y_{(2,t)} \\ z_{(2,t)} \end{pmatrix}, ...\}. \tag{2.1}$$

2.2 General Feature Extraction in Point Clouds

This section presents a summary of the main general feature extraction algorithms for point clouds. The algorithms presented in this section are based on the Point Cloud Library (PCL) [32], the importance of which has been considered in the study by S. Filipe [8]. In this study some of the algorithms of the keypoints module of PCL were tested. [8] concludes that algorithms that perform well based upon their repeatability are the Scale Invariant Feature Transform (SIFT) algorithm [20] and the Intrinsic Shape Signatures (ISS) algorithm [40]. However, the Harris Corner Detector [12] is also considered in this summary since it is a widely used corner detector in the literature.

It is important to mention that all the algorithms presented in this section have in common the use of a KD-tree and the algorithm radius search. A KD-tree is a data structure, based on binary search trees, that is used to store K-dimensional points, in this case 3-dimensional points. This data structure is useful to reduce the computational complexity when searching for a point. On the other hand, the radius search is an algorithm that returns neighbor points based on a query point. Therefore, the radius search defines a sphere with a certain radius for each point in the point cloud and returns a set of neighbors.

The Harris corner detector and the SIFT algorithm have their bases in image processing. The explanation, therefore, starts with the image processing version and then moves toward its adaptation to point clouds. Hence, section 2.2.1 describes the main concepts of the Harris Corner Detector, then the SIFT algorithm is described in section 2.2.2. Then, in section 2.2.3 the ISS algorithm is described. Finally, in section 2.2.4 the reasons why general feature extractors are not suitable for the LOAM algorithm are explained in detail.

2.2.1 Harris Corner Detector

The Harris Corner Detector [12] is a feature extractor proposed by C. Harris, and M. Stephens to detect corners originally in gray scale images. In their work, they mathematically formalize the Moravec's Corner Detector [23], and through a mathematical analysis, they improve the most important flaws of this detector.

2.2.1.1 Harris Corner Basics

Harris based his detector on the formalization of the Moravec's corner dectector, which indicates when a pixel is a corner and is defined by

$$e(u,v) = \sum_x \sum_y w(x,y)[i(x+u, y+v) - i(x,y)]^2. \tag{2.2}$$

Equation 2.2 quantifies the difference in intensity, $e(u,v)$, between an image patch $i(x,y)$ and a shifted image patch $i(x+u, y+v)$ convolved with a window function $w(x,y)$. The original Moravec corner detector uses a rectangular window that is one within a specified rectangular region, and zero elsewhere. What C. Harris and M. Stephens propose in their work is to replace the original window, which is binary and rectangular, with a Gaussian window to make the image smoother. The Gaussian window is defined by

$$w(x,y) = \exp(-(x^2+y^2)/2\sigma). \tag{2.3}$$

The next step is to take into account small shifts u, v that Moravec's detector does not consider. To do this, the function $i(x+u, y+v)$ of equation 2.2 is expanded using a Taylor series which results in

$$e(u,v) \approx \sum_x \sum_y [i(x,y) + \frac{\partial i}{\partial x} + \frac{\partial i}{\partial y} - i(x,y)]^2. \tag{2.4}$$

After mathematical rearrangement, equation 2.4 can be written in a matrix form as

$$e(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} \mathbf{M} \begin{bmatrix} u \\ v \end{bmatrix}, \tag{2.5}$$

where the matrix $\mathbf{M}$ is defined in

$$\mathbf{M} = \begin{bmatrix} \sum \frac{\partial i}{\partial x}^2 & \sum \frac{\partial i}{\partial x}\frac{\partial i}{\partial y} \\ \sum \frac{\partial i}{\partial x}\frac{\partial i}{\partial y} & \sum \frac{\partial i}{\partial y}^2 \end{bmatrix}. \tag{2.6}$$

The main diagonal elements of this matrix represent the sum of the derivatives of the image patch squared, which are the vertical and horizontal edges of the image patch. The other two elements are the sum of the product of previous derivatives. The analysis of matrix $\mathbf{M}$ is carried out via Eigen value decomposition, however, since the eigenvalue decomposition would be computationally expensive, Harris proposed a formula to avoid this operation. This formula is shown in

$$r = det(\mathbf{M}) - a(trace(\mathbf{M}))^2 \tag{2.7}$$

and is used as an analogy to analyze the eigenvalues of $\mathbf{M}$ (equation 2.6). When r is large, both eigen values are large, as a consequence a corner is assumed detected. When r is small a flat region is assumed and when r is negative an edge is assumed. The value a is an empirical constant that defines the sharpness of the corners detected.

2.2.1.2 Harris Corners in 3D Point Clouds

There are different versions of the Harris Corner Detector for point clouds, such as [36] and [30], however, in this work only the PCL version [32] is evaluated. The PCL version of the Harris corner detector adapts the idea of the original paper, which uses derivatives, but in this case it uses the surface normals of each point in the point cloud. These are vectors that are orthogonal to local surfaces estimated in the point cloud.

To estimate a surface normal, $s_i = (x_i \; y_i \; z_i)^T$, it is necessary to form a covariance matrix, $\mathbf{N}$, in a certain neighborhood via equation

$$\mathbf{N} = \frac{1}{m}\sum_{j=1}^{m}(p_j - \bar{p}) \times (p_j - \bar{p})^T, \mathbf{N} \times v_k = \lambda_k \times v_k, k \in \{0, 1, 2\}, \tag{2.8}$$

where m is the number of neighbors of each point $p_j = (x_j \; y_j \; z_j)^T$ in a certain neighborhood and $\bar{p}$ is the centroid of the set of points in the same neighborhood. The surface normal computation consists of evaluating the eigenvectors v_k and eigenvalues λ_k of the covariance matrix $\mathbf{N}$. The eigenvector associated with the smallest eigenvalue is considered as a surface normal s_i of p_j.

Once the surface normal of each point is computed, the matrix $\mathbf{M}$ of the original Harris algorithm (equation 2.6) is computed for each point with its neighbors. In this case the matrix $\mathbf{M}$ is modified to use the Euclidean components of the surface normals as

$$\mathbf{M} = \begin{bmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i z_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i z_i \\ \sum x_i z_i & \sum y_i z_i & \sum z_i^2 \end{bmatrix}. \tag{2.9}$$

It is important to note that matrices $\mathbf{N}$ and $\mathbf{M}$ of equations 2.8 and 2.9, respectively are calculated using the radius search algorithm. Finally, equation 2.7, which defined the response of a corner based on matrix $\mathbf{M}$ in the image version, remains the same for the 3D point cloud.

The computational complexity of the Harris Corner algorithm is closely related to the complexity of a KD-Tree and its operations. Therefore, the following items describe the steps of the 3D Harris corner detector and their complexity:

- Create KD-Tree: $n \log n$.
- Radius search: $mn \log n$.
- Compute Surface Normals: $mn \log n + mn$.
- Compute Response Point Cloud: $mn \log n + mn$.
- Compute Non Maximum Suppression: $mn \log n + mn$,

where n represents the number of points in the point cloud and m the number of neighbors in radius search. Finally the total complexity is given by $4mn \log n + n \log n + 3mn \in O(mn \log n)$.

In the analysis of this complexity the value m is important because it can be very small if the radius is small or it can be as large as n ($m \approx n$) if the radius is large. The surface normal computation is a procedure that is computationally expensive and is highly dependent on the value m. This characteristic makes the Harris algorithm not suitable for LOAM as will be seen later in section 2.2.4

2.2.2 Scale Invariant Features Transform (SIFT)

The Scale Invariant Features Transform (SIFT) [20] is a feature extractor proposed by David G. Lowe originally used with gray scale images. His work consists of many steps varying from extracting scale invariant keypoints to extracting complete descriptors. In this thesis we describe the basis of this algorithm and then we focus on its implementation in the Point Cloud Library. The SIFT algorithm can be divided into the following steps:

1. Scale-space extrema detection.
2. Keypoint localization.
3. Orientation assignment.
4. Keypoint descriptor determination.

In this thesis we only consider the steps related to keypoint (feature) computation, which are the first two items in the above list, because what we need is a just a sub-set of points of the point cloud. The orientation assignment is only useful to compute the whole keypoint descriptor consisting of a 128 dimensional vector per point which is important only when matching two images or point clouds.

2.2.2.1 SIFT Basics

The first step in the SIFT algorithm is to generate a scale space. This task is the most important in the detection of keypoints. As Lowe states: *"the first stage of keypoint detection is to identify locations and scales that can be repeatably assigned under differing views of the same object"*. He also states in his work that: *"Detecting locations that are invariant to scale change in the image can be accomplished by searching for stable features across all possible scales, using a continuous function of scale known as scale space"*.

Hence, keypoint detection is carried out by convolving the input image with a Gaussian kernel function. This operation can be expressed as

$$l(x, y, \sigma) = g(x, y, \sigma) * i(x, y), \tag{2.10}$$

where the $*$ represents convolution, $i(x, y)$ is the function that defines the image irradiance and $g(x, y, \sigma)$ is a Gaussian function defined as

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}. \tag{2.11}$$

Equations 2.10 and 2.11 are necessary to compute the Difference of Gaussian (DoG) images. This is done by subtracting images previously convolved with a Gaussian function scaled by different factors b. This operation can be defined mathematically as

$$d(x, y, \sigma) = (g(x, y, b\sigma) - g(x, y, \sigma)) * i(x, y) = l(x, y, b\sigma) - l(x, y, \sigma). \tag{2.12}$$

Equations 2.10 to 2.12 are used to generate different octaves. An octave is a sequence of images smoothed by differently scaled Gaussian kernels. At each octave, convolved images are sub-sampled. After generating the different octaves, the Difference of Gaussians (DoG) is computed per each pair of Gaussian images. This operation can be seen in Figure 2.1, which is from the original paper [20].

After DoG images are computed, the next step is to find the local extrema. To do this, in each DoG image (except the first and the last one), each pixel is evaluated with its 8 neighbors in the same scale and 9 neighbors in the scales above and below. The pixel is chosen as a candidate keypoint (feature), if it is larger or smaller than all its neighbors.

Once all keypoint candidates have been found, the final step is to carry out a refinement of these keypoints. Usually the maximum or minimum points do not lie exactly on a pixel value. Therefore, to access the sub-pixel values, the original author used a Taylor expansion of the image around the approximate keypoint up to the quadratic terms of the scale space function d, which results in

$$d(u) = d + \frac{\partial d^T}{\partial u} + \frac{1}{2} u^T \frac{\partial^2 d}{\partial u^2} u. \tag{2.13}$$

where $u = (x, y, \sigma)^T$ and $\frac{\partial d}{\partial u} = [\frac{\partial d}{\partial x} \; \frac{\partial d}{\partial y} \; \frac{\partial d}{\partial \sigma}]$. Then, the location of the extremum, $\hat{u}$, is found by taking the derivative of equation 2.13 and equating it to zero, which gives

$$\hat{u} = -\frac{\partial^2 d^{-1}}{\partial u^2} \frac{\partial d}{\partial u}. \tag{2.14}$$

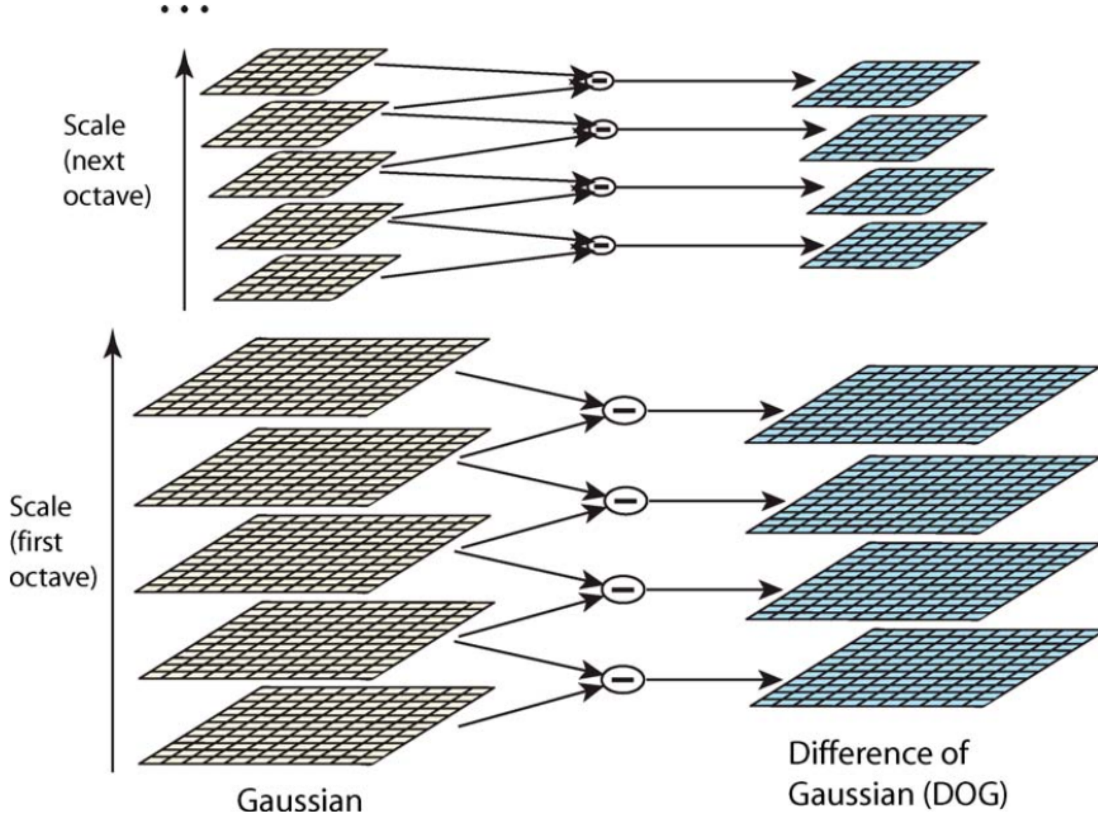


Figure 2.1: Computation of Difference of Gaussian images in the scale space.

Given that $\hat{u} = (\hat{x}, \hat{y}, \hat{\sigma})$ returns the shift for a keypoint, it would also return a sub-pixel value. To solve this issue, the author proposes that if the shift $\hat{u}$ is larger than 0.5 in any dimension, then that value lies on the next sample point. Finally, to reject unstable extrema, equation 2.14 is substituted into equation 2.13, resulting in

$$d(\hat{u}) = d + \frac{1}{2}\frac{\partial d^T}{\partial u}\hat{u}, \tag{2.15}$$

where shift $\hat{u}$ is evaluated and discarded as a keypoint, if it is less than 0.03. This value is proposed in the paper according to the experiments carried out by the author and is carried out to remove keypoints with low contrast.

2.2.2.2 SIFT in 3D Point Clouds

The SIFT keypoint detector applied to the field of 3D point clouds is a specific version developed for the Point Cloud Library and it presents many variations when compared with the image version.

As described earlier, an image can be defined as a 2D function $i(x, y)$, however, for a 3D point cloud, the PCL's particular implementation uses the spatial coordinates (x, y, z) of each

point and its corresponding intensity $p(x, y, z)$. This function represents the intensity of each point with Euclidean components x, y, z. This intensity, $p(x, y, z)$, depends on the sensor; for example it can be the quantified value of how much light power returns to the receiver of a LiDAR or simply the weighted sum of the R,G,B values taken by a RGBD camera.

A difference between the functions $i(x, y)$ and $p(x, y, z)$ is that while in the first case x, y are positive integers and adjacent numbers, in the second case x, y, z are continuous numbers and not necessarily adjacent and can be very sparse. The sparsity of these values is also dependent on the sensor. For example, in this work we used a Velodyne VLP-16 LiDAR which is sparse the in vertical direction, as will be explained in section 3.1.

The first step in the original SIFT algorithm performs the convolution of the image with a Gaussian kernel, which was described in equation 2.10. This equation presents some changes in the point cloud version as shown in following equation

$$l(x, y, z, d, \sigma) = g(d, \sigma) * p(x, y, z). \tag{2.16}$$

In equation 2.16, the function $p(x, y, z)$ is convolved with a Gaussian kernel of only 2 dimensions defined by $g(d, \sigma)$. In this equation, d represents the distances returned by a KD-tree radius search of each (x, y, z). With the distances available, the Gaussian function is defined as

$$g(d, \sigma) = \frac{1}{2\pi\sigma^2} e^{-d^2/2\sigma^2}. \tag{2.17}$$

Finally, as also occurs in the original SIFT algorithm, the difference of Gaussians is calculated as

$$d(x, y, z, d, \sigma) = (g(d, c\sigma) - g(d, \sigma)) * p(x, y, z) = l(x, y, z, d, c\sigma) - l(x, y, z, d, \sigma). \tag{2.18}$$

As occurs with the image version of the SIFT algorithm, the function $d(x, y, z, d, \sigma)$ has to be repeated in different octaves where point clouds are down-sampled. At each octave, the Gaussian kernels are computed with a different standard deviation σ and then they are convolved with their corresponding function $p(x, y, z)$. Finally, the difference of Gaussians are calculated by subtracting the convolved functions computed previously. The unique difference of this part of the SIFT algorithm, when compared with the image version, is that the keypoint removal, which depends on contrast, is performed directly without approximating the function with a Taylor expansion.

An important characteristic to consider when extracting keypoints with the SIFT algorithm using point clouds is that the algorithm needs an input rich in different intensity values. Therefore, a flat-intensity point cloud would not return any keypoints. Furthermore, there are various parameters to consider in order to extract good keypoints with SIFT, which are as follows:

- Number of octaves (o).
- Number of scales per octave (s),
- KD Tree radius search neighbors (m).
- Number of points of the point cloud (n).

These parameters are also important when computing the computational complexity of the SIFT algorithm. The top level of the iterations in the SIFT algorithm is the number of octaves, o. Per octave, there are sub-sets of point clouds with n elements. Each point cloud is convolved with a certain quantity of Gaussian kernels, s, of different scales. Gaussian kernels are formed by distances defined by a KD Tree radius search that returns m point neighbors. All these items have a total complexity of $O(o \cdot s \cdot m \cdot n \log n)$.

A common characteristic between the SIFT algorithm and the Harris corner detector is that the SIFT algorithm also depends on the KD Tree radius search to find m neighbors. As explained earlier, this characteristic, apart of the continuous calculation of Gaussian functions, make the SIFT algorithm computationally expensive and not suitable for real-time LOAM implementation.

2.2.3 Intrinsic Shape Signatures (ISS)

The Intrinsic Shape Signatures (ISS) algorithm [40] is a recent feature extractor developed by Y. Zhong for object recognition in 3D point clouds that has obtained repeatable results compared to other algorithms [8]. This algorithm is based in the eigenvalue decomposition of a covariance matrix of a certain spherical neighborhood of a point p_i. Since we are only interested in keypoints, only the eigenvalue decomposition is considered, because the rest of the steps, as with the SIFT algorithm, return a complete descriptor which is only useful when matching two point clouds.

The ISS algorithm is based on finding an intrinsic reference frame. This task is solved by constructing a covariance matrix within a spherical neighborhood of the point p_i under consideration, as shown in Algorithm 1.

Algorithm 1 Compute intrinsic reference frame

```
1: procedure ISS(P, r_density, r_frame)
2:     for all points in p_i ∈ P in a neighborhood defined by r_frame do
3:         w_i = 1 / ||p_j : |p_j − p_i| < r_density||
4:         COV(p_i) = Σ_{|p_j−p_i|<r_frame} w_j (p_j − p_i)(p_j − p_i)^T / Σ_{|p_j−p_i|<r_frame} w_j
5:         e_i = {λ_i^1, λ_i^2, λ_i^3}                          ▷ λ_i is an eigenvalue
6:     end for
7:     return E = {e_1, e_2, ..., e_n}
8: end procedure
```

In this algorithm, p_i is a point with Euclidean components x, y, z in a 3D point cloud $\mathcal{P}$, $r_{density}$ is used for the radius search and r_{frame} a supporting radius used in the computation

of the covariance matrix $COV(p_i)$. $r_{density}$ and r_{frame} are used to obtain neighbors p_j for each p_i. Therefore, the first step in this algorithm is to compute different weights w_i to compensate the uneven sampling of 3D points. After that, the covariance matrix, $COV(p_i)$ for each p_i according to the r_{frame} is computed. Finally k_i is formed by the eigenvalues $\lambda_i^1, \lambda_i^2, \lambda_i^3$ of the previous covariance matrix.

After executing the algorithm 1, we have a set of eigenvalues $\mathcal{E} = \{e_1, e_2, ..., e_n\}$ with the same number of elements as $\mathcal{P}$. To obtain the final keypoints it is necessary to find neighbors based on a radius for each point p_i. It is then necessary to evaluate if the largest eigenvalue λ_i of the corresponding point p_i is larger than the eigenvalues of its neighbors. If this condition is true, p_i is considered a keypoint.

The computational complexity of this algorithm is similar to the 3D Harris corner detector and is related to the KD-Tree complexities and how it partitions the whole point cloud according to a certain radius. However, as seen in the Algorithm 1 in this case two radius are used. Therefore, the algorithm complexities proceed as follows:

- Create KD-Tree: $n \log n$.
- Radius search 1 ($r_{density}$): $mn \log n$.
- Radius search 2 (r_{frame}): $ln \log n$.
- Find edge points: $lmn + lmn \log n$,

where n represents the number of points in the point cloud, m the number of neighbors in radius search 1 and l the number of neighbors in radius search 2. Finally the total complexity is given by $lmn \log n + ln \log n + mn \log n + n \log n + lmn \in O(lmn \log n)$.

As in the case of the complexity of the Harris corner detector and the SIFT algorithm, the value m is important because it determines the time the algorithm will take to process each radial neighborhood depending on the size of the radius.

2.2.4 Why a general feature extractor is not suitable for LOAM?

Analyzing the possible performance of any feature extractor in the LOAM algorithm is related to the number of features returned by the feature extractor and its computational complexity. In the case of the number of features, the original LOAM feature extractor returns more than $1,000$ features in less than 0.1 seconds (as will be seen later in the results section). In the case of the computational complexities of general feature extractors, it is important to note that the LOAM algorithm needs to estimate the LiDAR position and the map with the current point cloud before a new point cloud is input to the algorithm. According to the original LOAM paper [42], the feature extraction task is part of a block called "LiDAR Odometry". This block updates the LiDAR position 10 times per second. Since feature extraction is part of this block, any candidate to replace the feature extraction task should be fast enough to not affect the LiDAR odometry block.

General feature extractors for point clouds such as the Harris, SIFT and ISS algorithms can extract reliable features, however, they are not appropriate for the LOAM algorithm

because of their computational complexities. The complexities of these algorithms are shown in table 2.1.

Table 2.1: The computational complexities of well known point cloud feature extractors.

Algorithm	Complexity
Harris	$O(m \cdot n \log n)$
SIFT	$O(o \cdot s \cdot m \cdot n \log n)$
ISS	$O(l \cdot m \cdot n \log n)$

To see the equivalent values of these complexities in units of time, an experiment was carried out in which the Harris, SIFT and ISS algorithms were evaluated with a point cloud composed of $n = 21572$ points. This point cloud was recorded in an indoor environment and obtained with a Velodyne VLP-16 LiDAR. The experiment consisted of varying the radius related to the m neighbors on each algorithm and then the time each algorithm took to perform the task was evaluated and how many features it returned. For the SIFT algorithm the parameters considered were $o = 10$ and $s = 8$ and the ISS algorithm's second radius, r_{frame} (that related to the l neighbors), was set to 0.3 meters. This experiment was carried out with the PCL library and a computer Intel Core i7-7500U CPU @ 2.70GHz $\times$ 4.

Figure 2.2 shows the time (in seconds) each algorithm takes to process the point clouds and how many features are extracted by each algorithm. The time is represented on the vertical axis from 0 to 5 seconds and the number of features extracted by each algorithm for each radius is located next to the colored points. In this figure, it can be seen that the feature extractors take approximately from 0.25 to 4.3 seconds to extract features from the point cloud. Additionally, except for the SIFT algorithm, the algorithms return less than 150 features, which is a value far from the $1,000$ features extracted by the original LOAM feature extractor, mentioned before. Hence, including any of these algorithm in the LOAM algorithm would negatively affect the LiDAR odometry block.

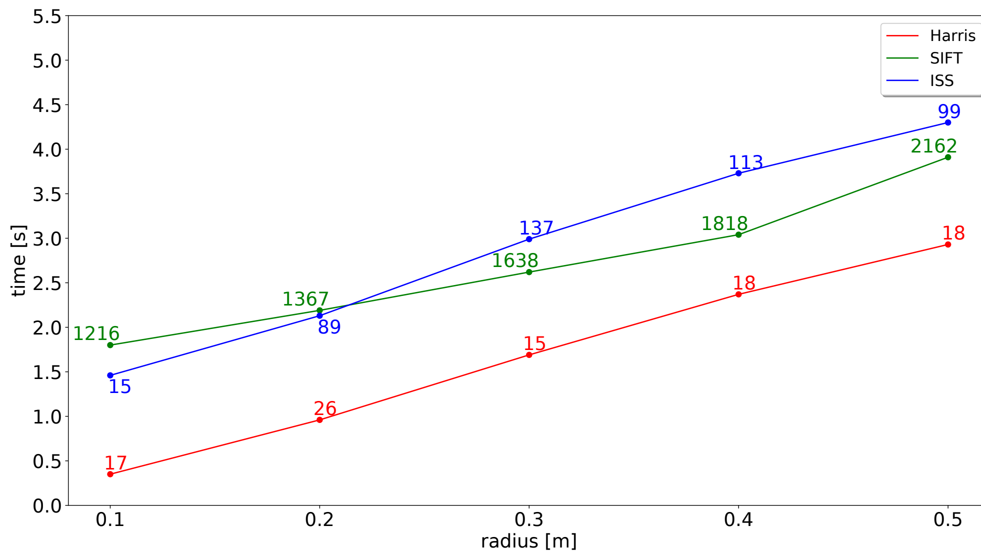


Figure 2.2: Processing time required by various feature extractors as a function of feature radius.

2.3 Feature Extraction algorithms commonly used in SLAM

There are various feature extraction algorithms that have been widely used in SLAM by the robotics community. Most of these algorithms use 2D LiDAR data to extract shapes such as lines and circles. Although not all were developed by the robotics community, they have been shown to have a good performance in robotics [25].

The most used algorithms for feature extraction with 2D LiDARs are those that extract lines and that consequently can extract corners. It is therefore important to know which of them have the best performance. The first three algorithms described in this section are part of a comparison performed by V. Nguyen et al. [25]. In that research it was concluded that the Split-and-Merge algorithm is the one that returned the best results in the number of features extracted and in terms of estimated feature position accuracy. The last two algorithms described in this section where evaluated and compared in the work done by P. Nuñez et al. [26]. P. Nuñez et al. proposed a new algorithm to extract features which was compared with various algorithms (Hough Transform and Split and Merge). Hence, the algorithms used in these articles are described below.

2.3.1 RANSAC

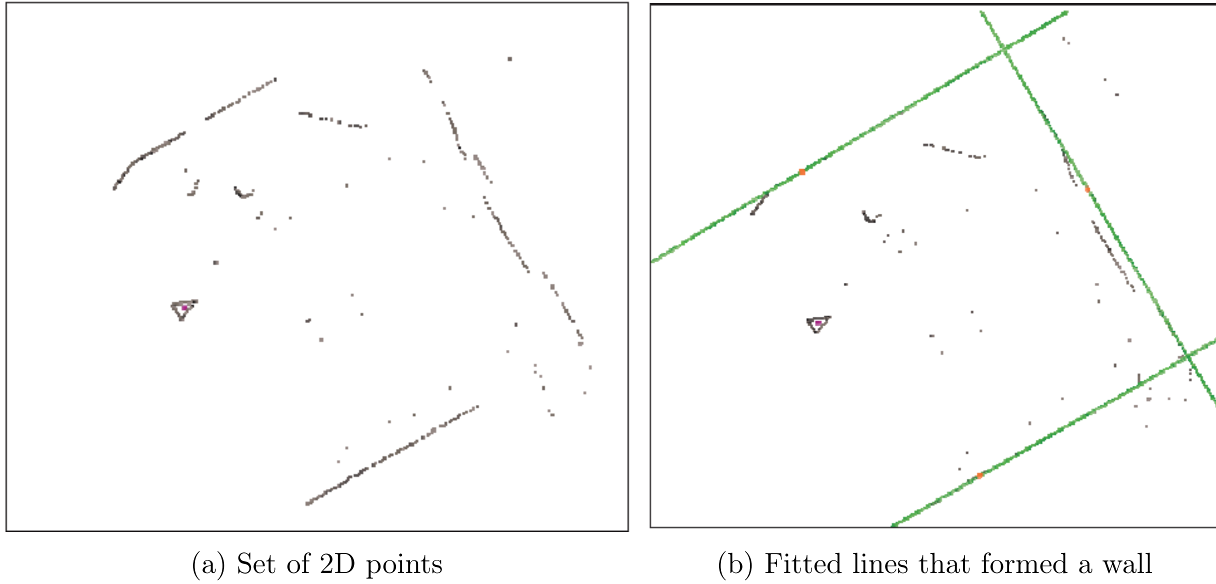
The Random Sample Consensus (RANSAC) algorithm [9] is an algorithm used to fit models in the presence of outliers. In the robotics community, this algorithm is commonly used to extract features such as lines [31] and circles [15], however, it can also be used to extract 3D features such as planes [10].

The RANSAC algorithm needs the general equation of the model (line, circle, plane, etc) and the minimum number of points needed to form that model. It iterates by fitting several parameters of the equation to finally take the one with the least error with respect to the outliers. An example of how this algorithm extracts lines as features in an indoor environment is shown in Figure 2.3a where from a data set of 2D points obtained with a 2D LiDAR, three different lines are fitted (Figure 2.3b).

The complexity of this algorithm is $O(s \cdot n \cdot nTrials)$, where s is the number of extracted model segments (lines), n the number of 2D points in the set and $nTrials$ the number of trials.

2.3.2 Hough Transform

The Hough Transform (HT) is an algorithm invented and patented by Paul Hough [13] in 1962. Like RANSAC, this algorithm needs a mathematical model, however, fitting is carried out by previously creating a Hough space in which the parameters of the model accumulate votes. The parameters with more votes are the ones that are selected to fit the model.



(a) Set of 2D points (b) Fitted lines that formed a wall

Figure 2.3: RANSAC in an indoor environment [31].

The HT algorithm is widely used in the computer vision community to find lines and circles. In the robotics community, this algorithm is not as popular as RANSAC in feature extraction, however, there are some articles which have adapted it successfully with LiDAR data [16] [28].

The input to this algorithm is n points with Euclidean components x, y that are candidates to form a line $\mathcal{L}$. When finding lines with the HT it is always the best option to work with the polar form to avoid undefined values in the slope of the line when using the slope-intercept form. The polar form of the line is described in equation 2.19 and depicted in Figure 2.4.

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{2.19}$$

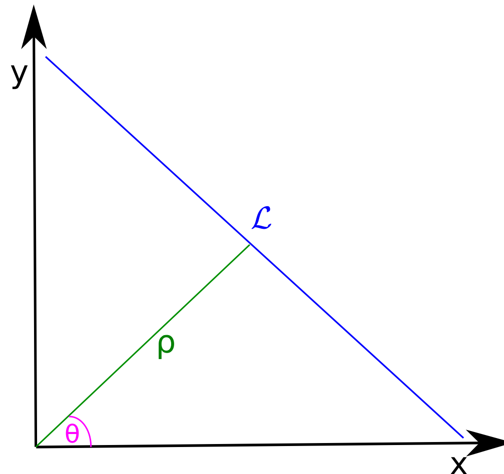


Figure 2.4: Polar representation of line $\mathcal{L}$.

Once the polar equation of a line is set, this algorithm depends on the accumulation of

votes to find the correct line or at least the one with more votes. This accumulation is also dependent on how we discretize the parameters of this algorithm. The parameters that are discretized are θ and ρ. Algorithm 2 describes the steps necessary to implement the Hough transform.

Algorithm 2 Hough Transform

```
 1: procedure HOUGHALGORITHM(x, y)
 2:     H_{ρ,θ} = {0}
 3:     for each point (x, y) do
 4:         for θ ← 0 to 180 do
 5:             ρ = x cos(θ) + y sin(θ)
 6:             H_{ρ,θ} = H_{ρ,θ} + 1
 7:         end for
 8:     end for
 9:     return values ρ, θ where H_{ρ,θ} are local maxima
10: end procedure
```

The HT algorithm needs as inputs the points x, y (in the case of working with images, an edge detector is required first). To begin, the Hough space ($H_{\rho,\theta}$), which can also be seen as a matrix H with indices ρ and θ, has to be initialized with zeros. Then, each point is used to calculate several ρ, which uses the discretized list of line angles, θ. After that, the matrix H at the indices (ρ, θ) must be incremented in one. Finally, the Hough transform returns θ and ρ values that form a local maximum in H.

The computational complexity of this algorithm is a weakness that make it difficult to fit models with more parameters. In the case of lines the complexity is $O(s \cdot n \cdot nc + s \cdot nr \cdot nc)$, where s is the number of segments (lines), n is the number of points and nc and nr the number of columns and rows of the Hough accumulator matrix H.

Although the previous description of the algorithm is general, it can also be applied to find lines in robotics, as V. Nguyen et al. [25] did in their experiments. In that article, the authors conclude that the HT is not the best option when extracting lines because of its computational complexity. Additionally, in the experiments carried out by P. Nuñez et al. [26], the HT extracted accurate lines but it involved a high computational complexity. A deeper analysis of why this feature extractor is not used in the LOAM algorithm is carried out in section 2.3.6.

2.3.3 Split-and-Merge

The Split-and-Merge algorithm was proposed by T. Pavlidis and S. Horowitz [27] to detect lines based on continuous partitions of segments. It originated in the computer vision community and its divide and conquer version has a complexity of $O(n \log n)$, where n is to total number of points in the 2D point cloud. The algorithmic description can be seen in Algorithm 3 and a graphical representation can be seen in Figure 2.5.

Algorithm 3 Split-and-Merge algorithm

1: Find the line $\mathcal{L}$ formed by the two furthest points.
2: Find the furthest point to the $\mathcal{L}$.
3: If the distance is greater than a threshold, then split and repeat the step 1 and 2 with the left and right sub-sets of points.
4: If the distance is lower than a threshold, then merge both segments.

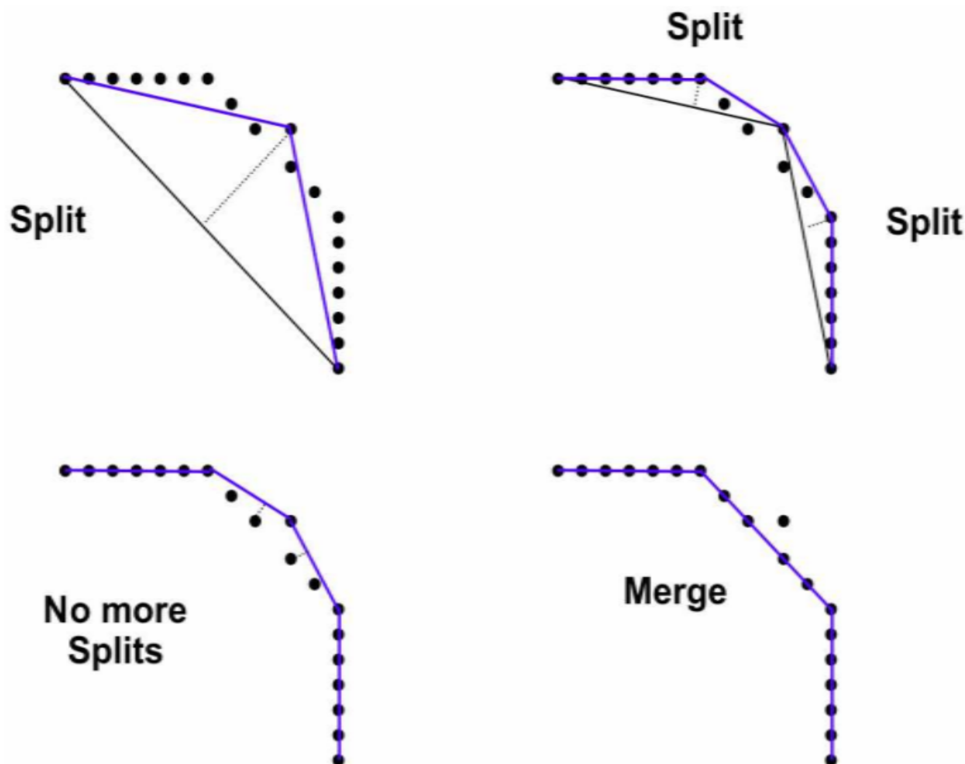


Figure 2.5: Split-and-Merge graphical representation [7].

The use of the split and merge algorithm is recommend in the study by V. Nguyen et al. [25] because it shows a high accuracy in the position of the features and its divide and conquer implementation make this feature extractor a fast algorithm. However, since this algorithm only extracts lines, its use is limited. The inclusion of the split and merge feature extractor in the LOAM algorithm is analyzed in section 2.3.6.

2.3.4 Adaptive Curvature Estimation

The adaptive curvature estimation algorithm was proposed by P. Nuñez et al. [26]. This algorithm estimates the curvature of a 2D point cloud obtained with a 2D LiDAR. The curvature obtained from this algorithm is then used to extract 4 types of features: line segments, centers of tree-like objects, corners and edges. Line segments and the centers of tree-like objects are determined using a non-linear regression method and the Levenverg-Marquardt algorithm, respectively. Corners are found with the intersection of the line segments previ-

ously extracted and edges are found using an adaptive breakpoint detector [6].

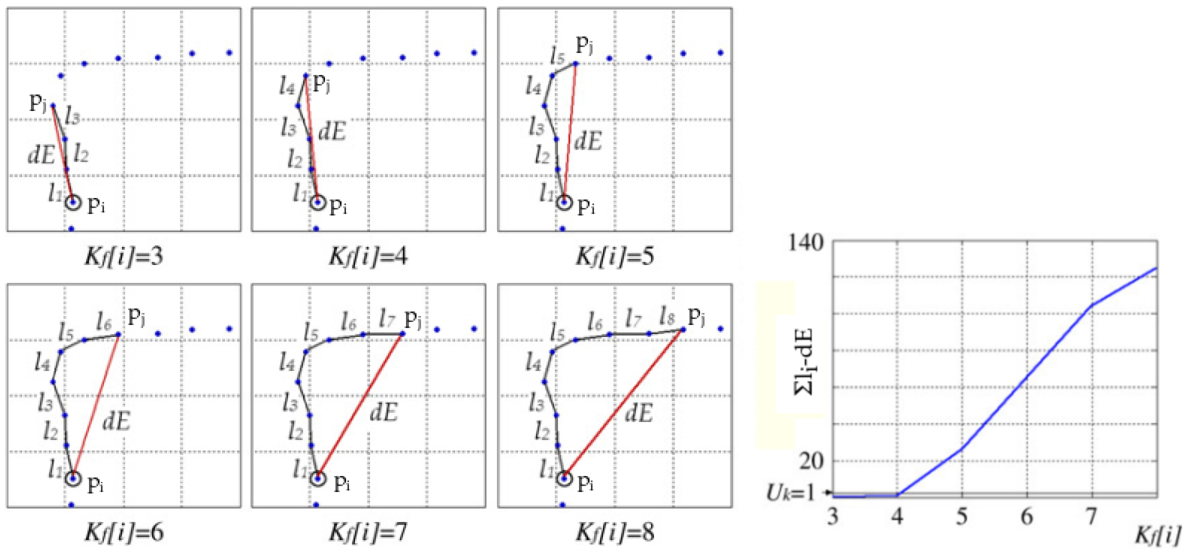
To estimate the curvature of a segment it is necessary to calculate: l_i, l and dE, defined in the following equations:

$$l_i = ||p_{i+1} - p_i|| \tag{2.20}$$

$$l(i,j) = \sum_{i=1}^{j} l_i \tag{2.21}$$

$$dE(i,j) = ||p_j - p_i||, \tag{2.22}$$

where l_i is the Euclidean distance between two consecutive points, l is the accumulated Euclidean distance from the ith point to the jth point and dE is the Euclidean distance from the ith point to the jth point. These distances are computed for the current point, p_i, in clockwise direction using the variable $K_f[i]$ and in counter-clockwise direction using the variable $K_b[i]$. An example of the computation of these distances on clockwise direction is shown in Figure 2.6a. For point sets with a high curvature, the difference of the sum of l_i and dE are larger than those that be obtained with straight lines. Figure 2.6b shows how the mentioned difference is below one for $K_f[i] = 3$ and $K_f[i] = 4$. This value is used to set the threshold U_k, which indicates that up to that point a line is determined. When $K_f[i]$ is greater than four the difference increases, which indicates that a potential high curvature of the data points have been found.



(a) Computation of l and dE (b) Difference of distances l and dE

Figure 2.6: Adaptive Curvature Estimation graphical representation [26].

The previous threshold, U_k, is dependent on functions l and dE. To define when a subset of points is a line, the following condition must be true:

$$l(i, i + K_f[i]) - dE(i, i + K_f[i]) \leq U_\kappa. \tag{2.23}$$

In the condition 2.23, U_k determines when data within a 2D point cloud forms a straight line. If U_k is very large, point clouds with a high range noise can be detected as straight lines. However, one must be careful with this constant because it can result in the confusion of straight lines with curves. The original article recommends determining U_k by scanning a flat surface like a wall and then calculating the distances l and dE to find the correct U_k using the equation 2.23.

In the original article [26], this algorithm is compared with feature extractors such as the Hough transform and the split and merge algorithm. It was concluded that their proposed algorithm had the overall best performance (recognition rate, accuracy of features location and processing time of the algorithm). However, the Curvature Scale Space algorithm had a better recognition rate even though it is more computationally complex. In this thesis, its use along with the LOAM algorithm was first considered, however, it was discarded because of the lack of robustness in the presence of range noise. This analysis is described later in section 2.3.6.

2.3.5 Curvature Scale Space

The Curvature Scale Space (CSS) algorithm was created by Mokhtarian F. and Mackworth A. [22] to determine different levels of curvature with a high precision. The algorithm was originally applied to images, but was then used for mobile robot feature extraction along with EKF-SLAM by Madhavan R. and Durrant-Whyte H. F. [21].

The original algorithm begins with the parametrization of curves within 2D point clouds in terms of functions $x(t)$ and $y(t)$, which result in a set of points described in the following equation:

$$C = \{x(t), y(t)\}. \tag{2.24}$$

In this equation, t is a linear function of the path length ranging over the closed interval $[0, 1]$. After defining the set of points, the variable that measures the curvature of C is κ. The variable κ can be defined as the instantaneous rate of change of the slope angle ψ of the tangent at point P with respect to arc length s. This expression is equal to the inverse of the radius of curvature ρ as follows:

$$\kappa = \frac{d\psi}{ds} = \frac{1}{\rho}. \tag{2.25}$$

κ can also be computed in terms of the derivatives of function y which are shown in the following equations:

$$y' = \frac{dy}{dx} \quad \text{and} \quad y'' = \frac{d^2y}{dx^2} \tag{2.26}$$

$$\kappa = \frac{y''}{(1 + (y')^2)^{3/2}}. \tag{2.27}$$

To use equation 2.27, it is important to recall that in equation 2.24 the curve C was parametrized in terms of t, which results in small variations of the derivatives of x and y, as shown in the equations:

$$\dot{x} = \frac{dx}{dt} \qquad \ddot{x} = \frac{d^2x}{dt^2} \qquad \dot{y} = \frac{dy}{dt} \qquad \ddot{y} = \frac{d^2y}{dt^2}. \tag{2.28}$$

Then

$$y' = \frac{\dot{y}}{\dot{x}} \quad \text{and} \quad y'' = \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^3}. \tag{2.29}$$

And finally, replacing 2.29 in 2.27 results in

$$\kappa = \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{(\dot{x}^2 + \dot{y}^2)^{3/2}}. \tag{2.30}$$

Now, curvature κ can be computed in terms of functions $x(t)$ and $y(t)$. Nevertheless, to find reliable features a scale space of Gaussian smoothing needs to be formed. The scale space requires these functions to be convolved with a Gaussian kernel $g(t, \sigma)$ at various levels of σ. So, the Gaussian kernel is defined as

$$g(t, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-t^2/2\sigma^2}. \tag{2.31}$$

Additionally, the convolution of $x(t)$ and $g(t, \sigma)$ is given by

$$X(t, \sigma) = x(t) * g(t, \sigma) = \int_{-\infty}^{\infty} x(u) \frac{1}{\sigma\sqrt{2\pi}} e^{-(t-u)^2/2\sigma^2} du \tag{2.32}$$

and $Y(t, \sigma)$ is defined in a similar manner.

In equation 2.30, one needs $\dot{x}$, $\ddot{x}$, $\dot{y}$, $\ddot{y}$; however, now we need those functions to be smoothed by the Gaussian kernel. Therefore, the new functions needed are $\dot{X}(t, \sigma)$, $\ddot{X}(t, \sigma)$,

$\dot{Y}(t,\sigma)$, $\ddot{Y}(t,\sigma)$. These functions can be computed from $x(t)$ and $y(t)$ as shown in equations 2.33 to 2.36

$$\dot{X}(t,\sigma) = \frac{\partial X(t,\sigma)}{\partial t} = \frac{\partial [x(t) * g(t,\sigma)]}{\partial t} = x(t) * \frac{\partial g(t,\sigma)}{\partial t} \tag{2.33}$$

$$\ddot{X}(t,\sigma) = \frac{\partial^2 X(t,\sigma)}{\partial t^2} = x(t) * \frac{\partial^2 g(t,\sigma)}{\partial t^2} \tag{2.34}$$

$$\dot{Y}(t,\sigma) = \frac{\partial Y(t,\sigma)}{\partial t} = y(t) * \frac{\partial g(t,\sigma)}{\partial t} \tag{2.35}$$

$$\ddot{Y}(t,\sigma) = \frac{\partial^2 Y(t,\sigma)}{\partial t^2} = y(t) * \frac{\partial^2 g(t,\sigma)}{\partial t^2}. \tag{2.36}$$

Finally, equation 2.30, which defines κ, results in

$$\kappa(t,\sigma) = \frac{\dot{X}(t,\sigma)\ddot{Y}(t,\sigma) - \dot{Y}(t,\sigma)\ddot{X}(t,\sigma)}{(\dot{X}(t,\sigma)^2 + \dot{Y}(t,\sigma)^2)^{3/2}}. \tag{2.37}$$

With equation 2.37, we are able to compute the curvature of each point at t for a certain smoothed curve with a scale σ. To detect the points with the highest curvature, we only need to detect those points that, in spite of smoothing, maintain a high curvature value. To extract features from a complete point cloud with the CSS algorithm we depend on the number of smoothing scales, s, the number of clusters, c (generated with the segmentation algorithm), and the number of points per cluster, n_c. So, the computational complexity of this algorithm can be expressed as $O(s \cdot c \cdot n_c)$.

The CSS algorithm was included in this literature review because it obtained good results in the comparison done by P. Nuñez [26]. Although, in that study, the CSS algorithm was slower than the adaptive curvature estimation algorithm when comparing their time complexities. The CSS algorithm is still an important algorithm that can be robust to range noise. This characteristic was important in the decision to discard the adaptive curvature estimation algorithm and choose the CSS algorithm to replace LOAM's feature extractor.

The previous description of the curvature scale space algorithm is general. Since this algorithm is the one selected to be implemented in this thesis it will be extended in the next chapter from its theory to specific implementation details corresponding to the LiDAR used in this work.

2.3.6 Why not apply other feature extractors in LOAM?

The analysis carried out for feature extractors commonly used in SLAM was based mainly in two research articles: *A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics* [25] and *Natural landmark extraction for mobile robot navigation based on an adaptive curvature estimation* [26]. The former article [25], evaluates line extraction algorithms in 2D point clouds. The algorithms studied in this research were: Split and Merge, Line Regression, the Incremental algorithm, RANSAC, Hough Transform and the Expectation Maximization (EM) algorithm. This article concludes that the best algorithm for line extraction is the Split and Merge algorithm based on its computational complexity, number of lines extracted, and position accuracy of the features extracted.

Article [26] proposed a new feature extraction algorithm based on adaptive curvature estimation, with comparisons to other algorithms. One of the algorithms included in this comparison is the Split and Merge algorithm, which was the best one according to V. Nguyen et al. The complete comparison includes the following algorithms: Split and Merge (SM), Split and Merge (SMF), Iterative End-Point-Fit (IEPF), Curvature Scale Space (CSS), Hough Transform (HT) and their proposal. A summary of the execution times of each algorithm can be seen in Table 2.2.

Table 2.2: P. Nuñez et al. [26] Feature extraction algorithms comparison.

Algorithm	SM	SMF	IEPF	HT	CSS	P. Nuñez et al. [26]
Execution Time (ms)	6.7	14.2	3.8	31.2	37.7	9.4
True Positives	0.76	0.79	0.77	0.71	0.92	0.91
False Positives	0.19	0.19	0.21	0.23	0.02	0.02

The Split and Merge (SM) algorithm extracts lines and had the best performance in V. Nguyen et al.'s comparison. This algorithm also has a variant called Split and Merge Fuzzy (SMF [6]), which uses fuzzy clustering before applying the SM algorithm. In P. Nuñez et al.'s comparison, these algorithms are used to extract lines and polygonal approximations of curves. [1] The results for Split and Merge and Split and Merge Fuzzy are good in terms of their speed but they lack detection accuracy. The lack of detection accuracy is a product of the polygonal approximation of curves.

The Iterative End-Point-Fit (IEPF) algorithm is a variant of the Split and Merge algorithm and consequently only extracts lines. This algorithm has a similar performance in terms of detection accuracy compared with the SM algorithm, however, the time it take to process the data is less. As occurs with the SM algorithm, the IEPF algorithm's correctness is low because of the polygonal approximation of curves.

The Hough Transform (HT) algorithm, in P. Nuñez et al.'s work, was only used to segment the laser scan. Its circle fitting version was not implemented because of its computational complexity. Although the HT algorithm was only used to segment the laser scan, it takes a significant amount of time to run.

[1] The polygonal approximation of curves consists on forming a curve with a sequence of estimated lines.

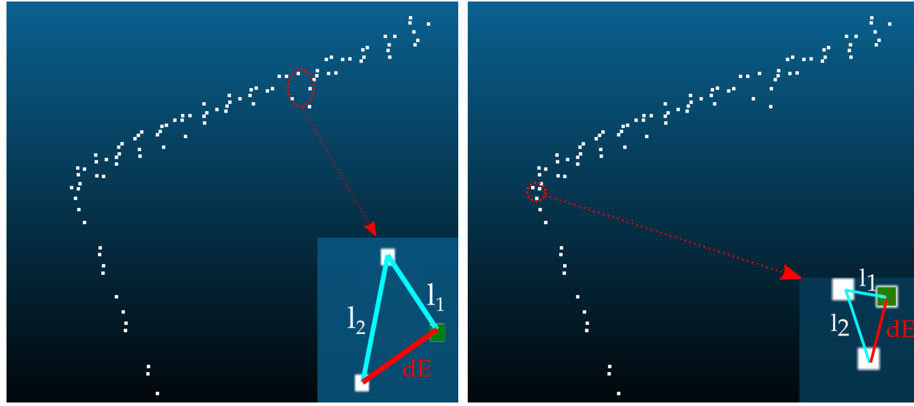
The remaining algorithms are the Curvature Scale Space (CSS) algorithm and P. Nuñez et al. proposal. In this case, both algorithms extract more than lines. Also, both algorithms base their behavior on a curvature variable κ, which indicates the points with curvature extrema. These methods differ in execution time and true positive rates (see table 2.2).

The execution time of P. Nuñez et al. proposal is less than CSS because the P. Nuñez et al. proposal uses the entire point cloud in its raw format and computes its curvature. In the case of the CSS algorithm, the point cloud data is smoothed several times with a Gaussian kernel until a scale space is created and then the curvature is computed.

The higher true positive rate of the CSS algorithm compared with P. Nuñez et al.'s proposal is due to creation of a suitable scale space. The different levels of smoothing that create the scale space in the CSS algorithm, make this algorithm more robust to noise. In the P. Nuñez et al. proposal, the range noise, which affects the curvature function κ, is controlled by the parameter U_k. U_k is dependent on functions l and dE, as described in equation 2.23 (functions l and dE are described in more detail in section 2.3.4). U_k indicates when a point of a set of points is the start of a curve. If U_k is large, the curvature of the point is high, which indicates the start of a curve; otherwise the curvature is low, which indicates the point is part of a straight line. However, this condition fails when the point cloud presents a high level of range noise. A representation of how this condition fails is shown in Figure 2.7.

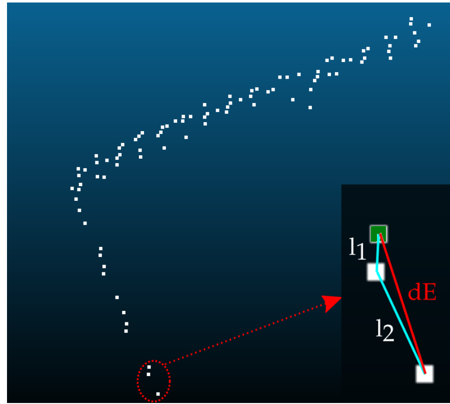
Figure 2.7 shows distances l_1, l_2 and dE evaluated in different points of a point cloud segment obtained with a VLP-16 LiDAR. This point cloud segment represents two walls that form one corner. Let us evaluate the green point on each sub-figure. In the case of Figure 2.7a $l_1 = 14.427$ mm, $l_2 = 19.118$ mm and $dE = 15.218$ mm, so $l(1,2) = 33.545$ mm and $l(1,2) - dE = \mathbf{18.327}$ mm. In the case of Figure 2.7b $l_1 = 5.150$ mm, $l_2 = 8.635$ mm and $dE = 7.644$ mm, so $l(1,2) = 13.785$ and $l(1,2) - dE = \mathbf{6.141}$ mm. Finally, in the case of Figure 2.7c $l_1 = 5.160$ mm, $l_2 = 16.192$ mm and $dE = 21.290$ mm, so $l(1,2) = 21.352$ mm and $l(1,2) - dE = \mathbf{0.062}$ mm. If we had to choose a U_k that satisfies the condition presented in Equation 2.23, we would fail, because we would have to choose a U_k between the **18.327** mm of Figure 2.7a or **0.062** mm of Figure 2.7c (both represent a wall). If we chose the first option, the corner presented in Figure 2.7b would be detected as part of a line (low curvature). On the other hand, if we chose the second option, the algorithm would detect two high curvatures (one in Figure 2.7b and one in Figure 2.7b). Therefore, the P. Nuñez et al. proposal would have problems with these level of range noise.

Although the adaptive curvature estimation algorithm performance was good with P. Nuñez et al.'s dataset, it would have problems with the data returned by the VLP-16. Therefore, the curvature scale space algorithm appears as an option to work in the LOAM algorithm since Gaussian filtering makes it robust to noise and can extract several features.



(a) Case 1: Three points that are part of a wall. $l(1,2) - dE = 18.327$ mm.

(b) Case 2: Three points that represent a corner. $l(1,2) - dE = 6.141$ mm.



(c) Case 3: Three point that are part of a wall. $l(1,2) - dE = 0.062$ mm.

Figure 2.7: Example of a real point cloud segment obtained with a VLP-16 LiDAR.

2.4 Adaptive Breakpoint Detector

In the feature extraction literature focused on SLAM, there are some feature extractors that require a previous step called segmentation. Segmentation is the task of clustering certain points of the cloud based on their distance. This task is important because it can reduce the computational complexity of the whole feature extractor.

Segmentation algorithms can also be called *breakpoint detectors* because they detect where a set of points presents a break. There are many approaches one can take to detect the breakpoints in a point cloud, from those that use algorithms such as the Kalman filter [7] [1] to those that are based on the Euclidean distance to separate the points [37]. In this thesis, we used an algorithm proposed by G. Borges and M. Aldon [6] called the *Adaptive Breakpoint Detector* because it improves the approaches based on the Euclidean distance and according to [6] its performance is similar when compared to Kalman filter breakpoint detectors.

The Adaptive Breakpoint Detector detects a breakpoint based on the comparison of the

current point p_i and the previous point p_{i-1}. Then, two distances are computed: the Euclidean distance between p_i and p_{i-1}, and a threshold called D_{max}. This threshold is a distance formed by a virtual line that crosses the previous point p_{i-1}. This line has an inclination λ with respect to the line r_{i-1} and represents an extreme case where an environmental line can be reliably detected.

In Figure 2.8, the origin of the x and y axes represents the LiDAR's location, the red lines are consecutive laser beams where each range is represented by r_i. The blue points are points p_i detected by the LiDAR, the light blue dotted line is the virtual line with inclination λ with respect to r_{i-1} and the green circle is the threshold circle with radius D_{max}. If a point p_i is outside the circle, p_i and p_{i-1} are marked as breakpoints. It is necessary to focus on the triangle formed by the line r_{i-1} and the distances r_i^h and D_{max}. Distance r_i^h represents the hypothetical range that goes from the origin of the LiDAR to its intersection with the virtual planar obstacle line.

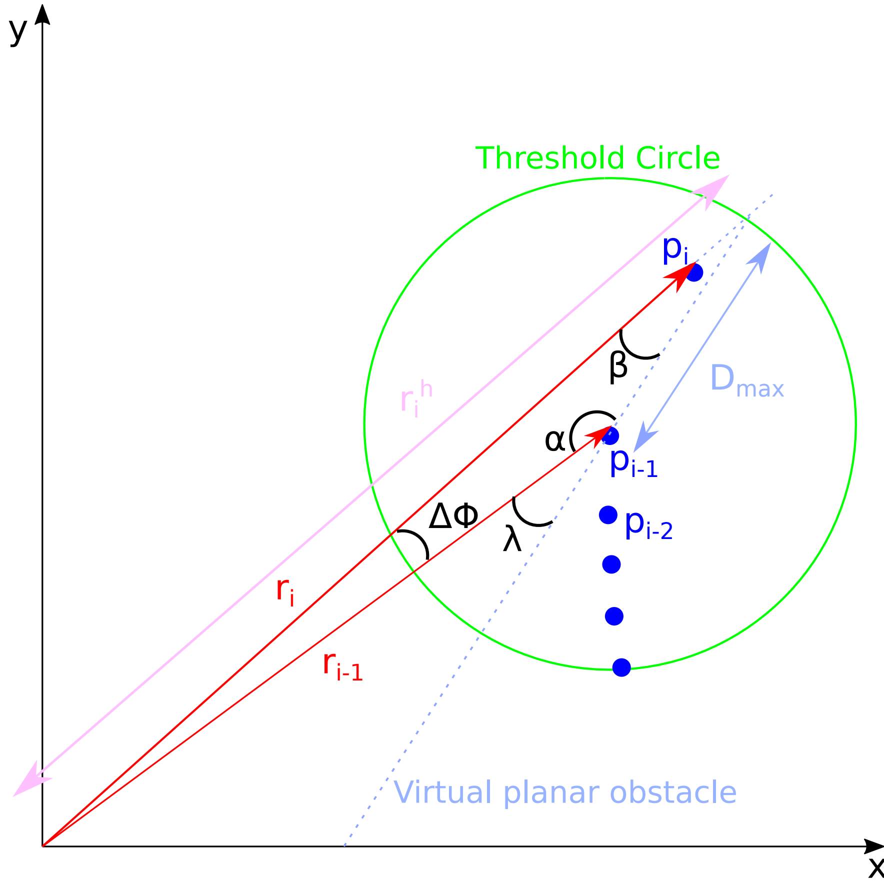


Figure 2.8: Adaptive breakpoint detector (adapted from [6]).

Based on angles $\Delta\phi$, α and β of the triangle shown in Figure 2.8, we can define the following equations:

$$\alpha = 180^\circ - \lambda \tag{2.38}$$

$$\Delta\phi + \alpha + \beta = 180°. \tag{2.39}$$

Replacing 2.39 in 2.38, we obtain

$$\beta = \lambda - \Delta\phi. \tag{2.40}$$

Then, the hypothetical range r_i^h for the ith range is related to r_{i-1} by the following equation:

$$r_{i-1} \cdot sin(\alpha) = r_i^h \cdot sin(\lambda - \Delta\phi). \tag{2.41}$$

Based on the previous equations, we can now define an equation for D_{max} as follows

$$D_{max} = r_{i-1} \frac{\sin(\Delta\phi)}{\sin(\lambda - \Delta\phi)}. \tag{2.42}$$

D_{max} can be used as a threshold to detect breakpoints. However, this value does not consider the noise associated with r_{i-1}. To tackle this issue, the author proposes to add 3 times the standard deviation of the sensor range noise to D_{max}. However, in this thesis we computed the range accuracy of D_{max} ($\Delta_{D_{max}}$) associated to the whole equation 2.42 by taking the derivative of D_{max} with respect to r_{i-1} and multiplying it by the range accuracy of the LiDAR Δ_r as follows

$$\Delta_{D_{max}} = \frac{dD_{max}}{dr_{i-1}} \cdot \Delta_r \tag{2.43}$$

According the Velodyne VLP-16 LiDAR datasheet, $\Delta_r = 0.03$ m and $\Delta\phi = 0.4$. Additionally, in this case we used $\lambda = 10°$; so equation 2.43 results in

$$\Delta_{D_{max}} = \frac{\sin(\Delta\phi)}{\sin(\lambda - \Delta\phi)} \cdot \Delta_r = 0.0013 \text{ m}. \tag{2.44}$$

Finally, the range accuracy of D_{max} associated to D_{max} is added to the equation 2.42, which results in

$$D_{max} = r_{i-1} \frac{\sin(\Delta\phi)}{\sin(\lambda - \Delta\phi)} + \Delta_{D_{max}}. \tag{2.45}$$

With D_{max} defined, the adaptive breakpoint detector algorithm can be implemented as will be shown in Algorithm 5 in the next chapter. This algorithm was implemented as a previous task of the CSS algorithm and it is useful to reduce the time complexity of the feature extractor.

2.5 LiDAR Odometry and Mapping (LOAM)

One of the objectives of this thesis is to replace, in the implementation, the original LiDAR Odometry And Mapping (LOAM) feature extractor with the CSS algorithm. To achieve this objective however, it is important to understand how the replacement of the original feature extractor could affect the whole LOAM algorithm. To do this, a summary of the components of the LOAM algorithm will now be presented.

The LOAM algorithm, proposed by J. Zhang [41], is one of the main algorithms used in the state of the art in SLAM. In the KITTI Vision benchmark [11], it is ranked first place since 2016, in terms of the estimated trajectory error. This algorithm performs localization and mapping through a feature matching approach using LiDAR data. The original LOAM experiment used a Hokuyo UTM-30LX LiDAR and the description of the geometric algorithms are based on this sensor [42]. In the case of this thesis, all these algorithms are described based on the Velodyne VLP-16 LiDAR, which generates a sparse point cloud.

The LOAM algorithm consists of many software blocks as shown in the diagram of Figure 2.9. This diagram starts from the left side with the LiDAR block returning a set of points $\hat{\mathcal{P}}$. The point cloud registration block is responsible for ordering the previous points according the elevational scan in which they belong. Additionally, this block labels these points with a positive integer value k that indicates the number of the sweep. A sweep occurs when all the lasers of the Velodyne VLP-16 LiDAR complete a rotation from 0° to 360°. Hence, when the software begins to process each point cloud, returned by the LiDAR, that point cloud is a assigned with a sweep $k \in \mathbb{Z}^+ = \{1, 2, ...\}$. Therefore, the output of the point cloud registration block is a point cloud $\mathcal{P}_k$. $\mathcal{P}_k$ contains several points; each point identified with a sub-index i corresponding to the sweep k and is denoted as $p_{(i,k)} \in \mathcal{P}_k$. After that, $\mathcal{P}_k$ feeds the LiDAR odometry block, which performs the feature extraction task and estimates the position of the LiDAR. This block returns a point cloud with the rotational effect of the LiDAR compensated according to the time of scanning of each point (undistorted $\mathcal{P}_k$). Also, this block uses the estimated position of the LiDAR to return a transform update between the LiDAR coordinates and the world coordinates. Finally, the undistorted point cloud $\mathcal{P}_k$ is used by the LiDAR mapping algorithm to estimate the map (map output) and also to return a second transform update. Both transform updates returned by the LiDAR odometry block and the LiDAR mapping block are used by the transform integration block to compute a final transform between the LiDAR coordinates and the global world coordinates.

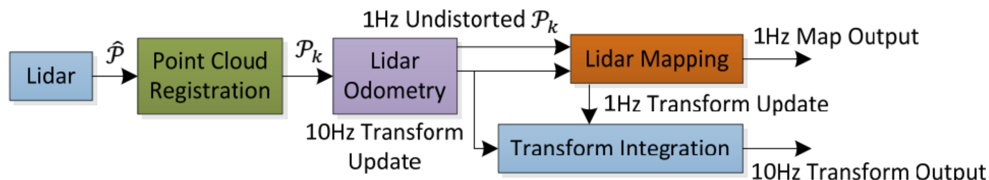


Figure 2.9: Block diagram of the LOAM software system [42].

In this thesis, we only focus on the LiDAR odometry block which is responsible for the task of feature extraction and the estimation of the robot's trajectory.

2.5.1 LiDAR Odometry Estimation

The LOAM software system divides the LiDAR odometry block into 4 tasks: feature extraction, feature point correspondence, motion estimation and the LiDAR odometry algorithm. These are each explained below.

2.5.1.1 LOAM's Feature Extraction algorithm

The LiDAR odometry block is responsible for computing the trajectory of the robot. This block performs its task by finding the translation and rotation between two complete scans of the VLP-16. To find the transform for these translations and rotations, the LOAM algorithm does not use all of the point cloud data, but it finds a special sub-set called *features.*

The VLP-16 and Velodyne LiDARs in general are composed of N individual lasers and this configuration is considered in the implementation. To take advantage of the configuration of this sensor, the feature extraction algorithm processes each laser as an individual elevational scan and extracts non-smooth and smooth points. From here on non-smooth points will be called *sharp* features and smooth points will be called *flat* features. A set of sharp features is denoted by $\mathcal{E}_k = \{e_{(1,k)}, e_{(2,k)}, ..., e_{(n,k)}\}$, where the first sub-index indicates the number of feature and the second one the number of sweep. Additionally, a set of flat features is denoted by $\mathcal{H}_k = \{h_{(1,k)}, h_{(2,k)}, ..., h_{(n,k)}\}$, where the sub-indices show the same information as in the case of sharp features. Both $\mathcal{E}_k$ and $\mathcal{H}_k$ are part of $\mathcal{P}_k$ ($\mathcal{E}_k, \mathcal{H}_k \subset \mathcal{P}_k$).

To decide when a point $p_{(i,k)}$ in $\mathcal{P}_k$ is included in $\mathcal{E}_k$ or $\mathcal{H}_k$, the original paper defines a threshold value c as

$$c = \frac{1}{|\mathcal{R}| \cdot ||p_{(i,k)}||} || \sum_{j \in \mathcal{R}, j \neq i} (p_{(i,k)} - p_{(j,k)}) ||. \tag{2.46}$$

The threshold c in equation 2.46 defines the smoothness of each elevational scan of the LiDAR and accordingly the points are categorized as sharp or flat features. Additionally, $\mathcal{R}$ is the set of consecutive points in one scan, $p_{(i,k)}$ and $p_{(j,k)} \in \mathcal{R}$ and $\mathcal{R} \subset \mathcal{P}_k$. Therefore, the LOAM feature extractor returns flat (smooth) and sharp (non-smooth) points per each elevational scan of a Velodyne LiDAR.

2.5.1.2 Feature Point Correspondence

In feature point correspondence, we have to perform a matching between feature points found at time t_{k+1} with their closest neighbors found at time t_k. t_k denotes the time stamp when a sweep k is completed. To complete this task we need to use $\bar{\mathcal{P}}_k$, $\tilde{\mathcal{E}}_{k+1}$ and $\tilde{\mathcal{H}}_{k+1}$. $\bar{\mathcal{P}}_k$ represents $\mathcal{P}_k$ but it has assigned the time stamp t_{k+1}, $\tilde{\mathcal{E}}_{k+1}$ and $\tilde{\mathcal{H}}_{k+1}$ are the same features as $\mathcal{E}_{k+1}$ and $\mathcal{H}_{k+1}$ but they has assigned the time the sweep was initially scanned.

Feature point correspondence for sharp features is depicted in Figure 2.10. In this figure, the correspondence of the sharp feature $\tilde{e}_{(i,k+1)} \in \tilde{\mathcal{E}}_{k+1}$ is determined by finding two neighbors, $\bar{p}_{(j,k)}, \bar{p}_{(l,k)} \in \bar{\mathcal{P}}_k$, in two consecutive scans (scan 2 and scan 3) and found at time of the previous sweep, t_k.

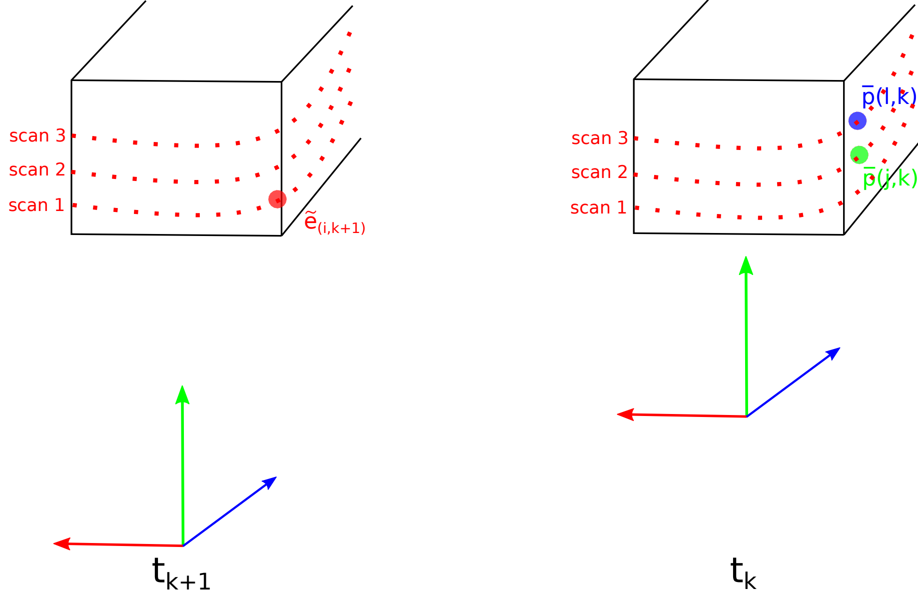


Figure 2.10: Sharp feature correspondence.

Likewise, the feature point correspondence for flat features is depicted in Figure 2.11. In this figure, the correspondence of the flat feature $\tilde{h}_{(i,k+1)} \in \tilde{\mathcal{H}}_{k+1}$ is determined by finding three neighbors, $\bar{p}_{(j,k)}, \bar{p}_{(l,k)}, \bar{p}_{(m,k)} \in \bar{\mathcal{P}}_k$, in two consecutive scans (scan 2 and scan 3) and found at time of the previous sweep, t_k.

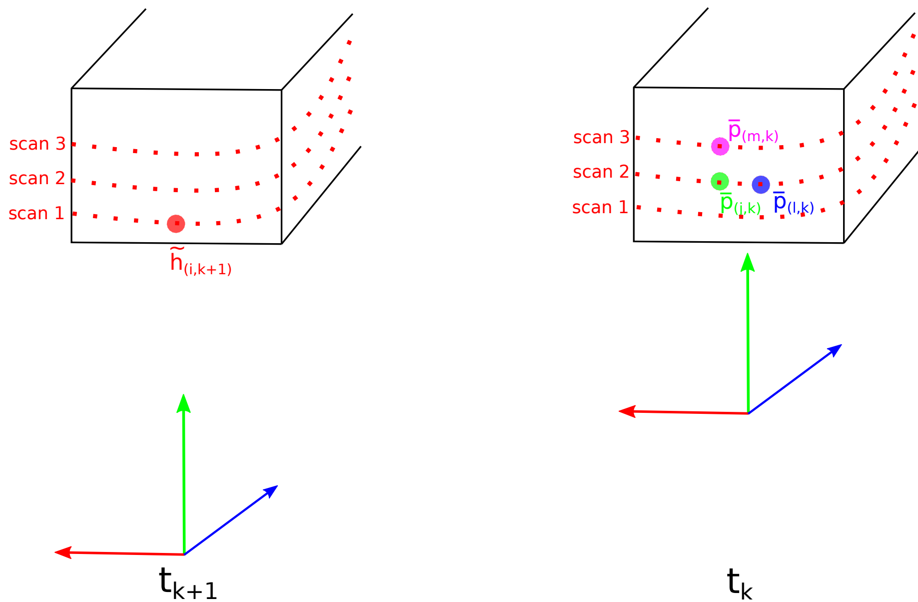


Figure 2.11: Flat feature correspondence.

2.5.1.3 Motion Estimation

To estimate the motion of the robot it is necessary to have features at time t_{k+1} and their corresponding points at time t_k. This task can be completed by calculating the distances from feature points to their corresponding points and *minimizing* them. In the case of sharp features, a point to line distance has to be computed, while in the case of flat features, a point to plane distance is needed.

The LOAM algorithm's authors model the robot motion with constant linear and angular velocities during a sensor sweep k. Then, the robot motion can be defined as the robot pose transform between the times t_{k+1} and t_k denoted as T_{k+1}. T_{k+1} contains the rigid motion of the robot with respect to axes x, y, z, so $T_{k+1} = [t_x, t_y, t_z, \theta_x, \theta_y, \theta_z]$. t_x, t_y, t_z are the translations in the x, y, z directions and $\theta_x, \theta_y, \theta_z$ their rotation angles about the x, y and z axes.

The robot motion is solved with the Levenberg-Marquardt algorithm using the equation:

$$f(T_{k+1}) = d. \tag{2.47}$$

where, f is a function that returns a matrix and each row of it corresponds to a feature point, and d contains the corresponding distances. Then, to use the Levenberg-Marquadt algorithm, the Jacobian matrix of f with respect to T_{k+1} has to be computed. This matrix is denoted as $J = \partial f / \partial T_{k+1}$. Finally equation 2.47 can be solved by minimizing d

$$T_{k+1} \leftarrow T_{k+1} - (J^T J + \lambda diag(J^T J))^{-1} J^T d, \tag{2.48}$$

where λ is a factor determined by the Levenberg-Marquadt algorithm.

2.5.1.4 LiDAR Odometry Algorithm

The final LiDAR odometry algorithm is based on the equations shown in the previous subsection. This algorithm needs as inputs the point cloud from the last sweep $\bar{\mathcal{P}}_k$ and the point cloud $\mathcal{P}_{k+1}$. This procedure is described in algorithm 4.

As shown in algorithm 4, the odometry estimation depends on the Levenberg-Marquardt optimization method and this last algorithm has as inputs the features computed in the previous task. Hence, it implies that the more accurate the features are, the less error the odometry estimation will have. This last statement is the basis of implementing a new feature extractor for the LOAM algorithm.

Algorithm 4 LiDAR Odometry

```
1: procedure LOAMOdometry(P̄_k, P_{k+1})
2:     if at the beginning of the scanning a sweep then
3:         T_{k+1} ← 0
4:     end if
5:     Detect sharp points and flat points in P_{k+1} and assign the points to E_k and H_k
6:     do
7:         for each sharp point in E_k do
8:             Find the correspondence of the point, then compute point to line distance and
               assign those values the matrix of function 2.47
9:         end for
10:        for each flat point in H_k do
11:            Find the correspondence of the point, then compute point to plane distance
               and assign those values the matrix of function 2.47
12:        end for
13:        Update T_{k+1} for a nonlinear iteration based on 2.48
14:    while the non linear optimization does not converges
15:    if at the end of the scanning of a sweep then
16:        Assign each point p_{(i,k+1)} ∈ P_{k+1} with the time stamp t_{k+2} and form P̄_{k+1}
17:        return T_{k+1} and P̄_{k+1}
18:    else
19:        return T_{k+1}
20:    end if
21: end procedure
```

2.5.2 Why the LOAM's Feature Extractor can be improved?

Until now, various general feature extractors were discarded because of their computational complexity which is a result of the *Radius Search* algorithm on which they are based. Our analysis continued with the feature extractors commonly used for SLAM based on two articles [25] [26]. Based on these articles, we concluded that the Curvature Scale Space algorithm gives the best results, which makes it the most suitable for the type of data obtained with the Velodyne VLP-16 LiDAR.

The remaining feature extractor to analyze, according to the literature review, is the feature extractor of the *LiDAR Odometry And Mapping (LOAM)* algorithm itself. This feature extractor has a version specially designed and tested with the VLP-16 LiDAR data used in this thesis. Hence, the core of this section is to show with some examples why this feature extractor can be improved.

As described in section 2.5.1.1, the LOAM feature extractor finds sharp and flat points in the point cloud. To achieve that task, the smoothness of each point is analyzed with equation 2.46. This equation is effective to extract an important quantity of features according to the smoothness. However, the main problem of this feature extractor occurs when it processes point clouds with various levels of range noise, as that shown in Figure 2.7.

Equation 2.46 evaluates the smoothness of each point by summing the difference of each point with the rest of points in the same elevational scan. In other words, *the smaller c is, the higher smoothness it has*. Although this equation gives good results when the noise is uniform, it can confuse sharp and flat features when the range noise varies in some segments of the point cloud. To show this issue some examples will be evaluated.

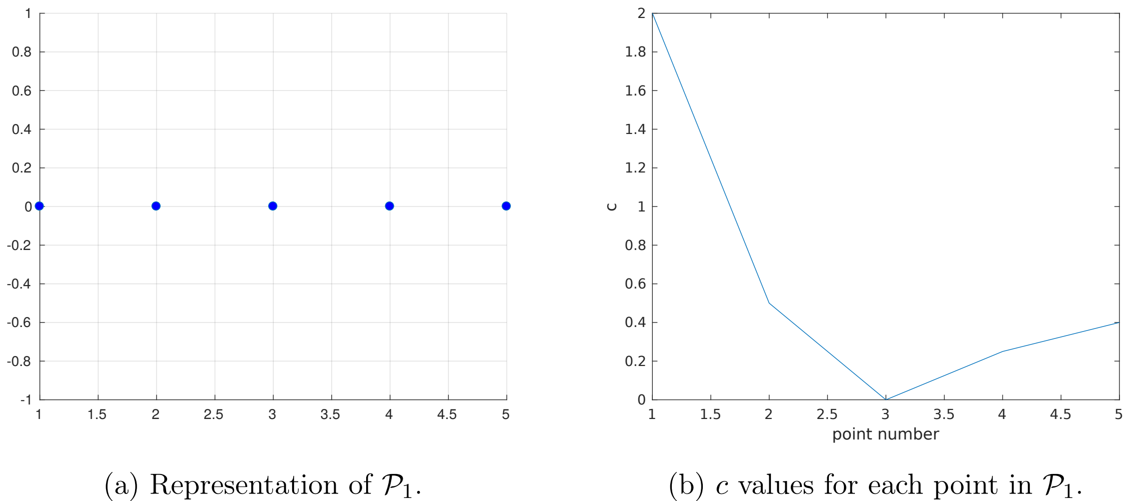
Before beginning with the examples, it is important to recall that, in the original implementation of LOAM's feature extractor with the VLP-16 LiDAR, each elevational scan is processed separately. This characteristic makes the smoothness, c, to be affected the most by the Euclidean components x and y. Therefore, in the following examples only 2D point clouds are used. Each point cloud is represented by $\mathcal{P}_i$. For each point in it, c is calculated with equation 2.46.

Example 1: Let $\mathcal{P}_1$ be a 2D point cloud with 5 elements that defines a horizontal line with no range noise, whose values can be seen in table 2.3.

Table 2.3: Elements of point cloud $\mathcal{P}_1$.

index	1	2	3	4	5
x	1	2	3	4	5
y	0	0	0	0	0

Point cloud $\mathcal{P}_1$ and its smoothness c for each point can also be graphically represented as shown in Figure 2.12.



(a) Representation of $\mathcal{P}_1$.

(b) c values for each point in $\mathcal{P}_1$.

Figure 2.12: Point Cloud $\mathcal{P}_1$ and its smoothness c.

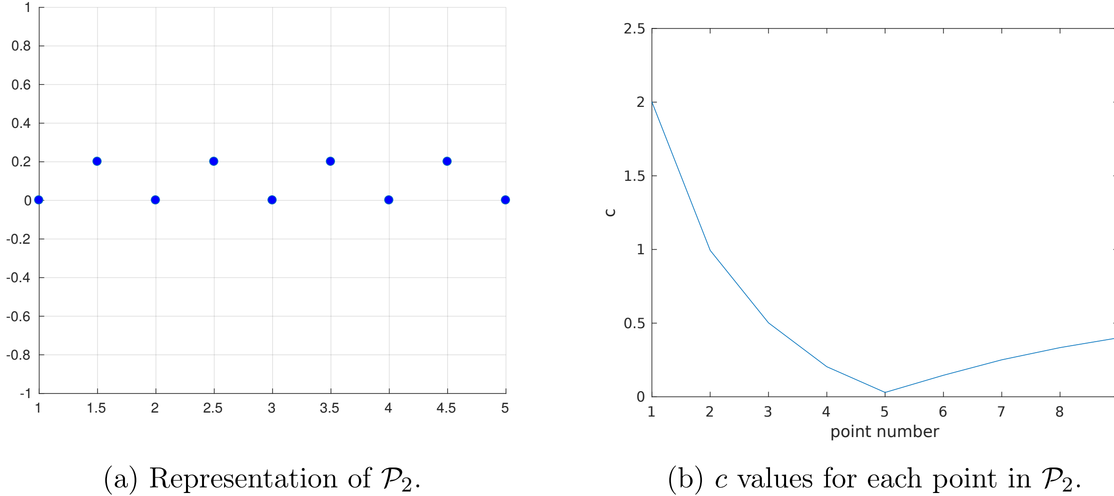
The computation of equation 2.46 for the point cloud $\mathcal{P}_1$ can be seen graphically represented in Figure 2.12b. In this figure, if we want to determine which point is a sharp feature, first we would have to define a threshold. Since these points represent a line, a possible threshold would be 2, because it is the maximum value of c reached in Figure 2.12b (this threshold will be used from here on), therefore all c values above 2 would represent a sharp feature. Hence, $\mathcal{P}_1$ would not return any sharp feature.

Example 2: Let $\mathcal{P}_2$ be a 2D point cloud with 9 elements that define a line with some spikes that represent range noise and whose values can be seen in table 2.4.

Table 2.4: Elements of point cloud $\mathcal{P}_2$.

index	1	2	3	4	5	6	7	8	9
x	1	1.5	2	2.5	3	3.5	4	4.5	5
y	0	0.2	0	0.2	0	0.2	0	0.2	0

Point cloud $\mathcal{P}_2$ and its smoothness c for each point can also be graphically represented as shown in Figure 2.13.



(a) Representation of $\mathcal{P}_2$.

(b) c values for each point in $\mathcal{P}_2$.

Figure 2.13: Point Cloud $\mathcal{P}_2$ and its smoothness c.

The computation of equation 2.46 for the point cloud $\mathcal{P}_2$ can be seen graphically represented in Figure 2.13b. In this figure, as also occurred for the point cloud $\mathcal{P}_1$, if we consider the threshold in the previous example, no sharp feature would be found in $\mathcal{P}_2$. In spite of the noise presented in $\mathcal{P}_2$, Figure 2.12b and Figure 2.13b look very similar.

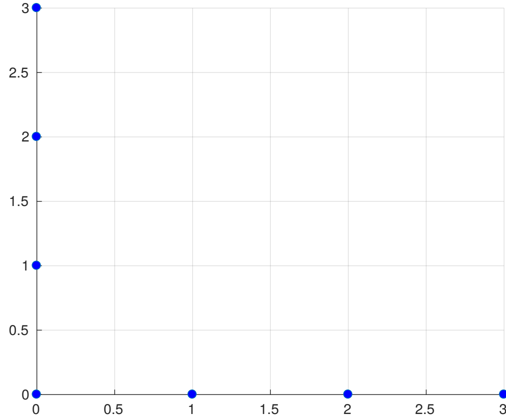
Example 3: Let $\mathcal{P}_3$ be a 2D point cloud with 11 elements that define a corner with no range noise and whose values can be seen in table 2.5.

Table 2.5: Elements of point cloud $\mathcal{P}_3$.

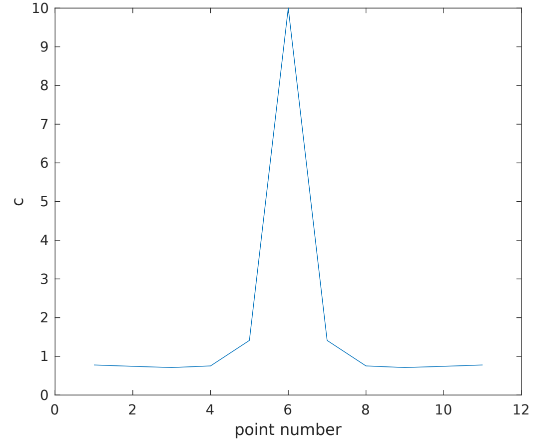
index	1	2	3	4	5	6	7	8	9	10	11
x	0.00	0.00	0.00	0.00	0.00	0.00	1.00	2.00	3.00	4.00	5.00
y	5.00	4.00	3.00	2.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00

Point cloud $\mathcal{P}_3$ and its smoothness c for each point can also be graphically represented as shown in Figure 2.14.

The computation of equation 2.46 for the point cloud $\mathcal{P}_3$ can be seen graphically represented in Figure 2.14b. Point cloud $\mathcal{P}_3$ is completely different from $\mathcal{P}_1$ and $\mathcal{P}_2$, consequently,



(a) Representation of $\mathcal{P}_3$.



(b) c values for each point in $\mathcal{P}_3$.

Figure 2.14: Point Cloud $\mathcal{P}_3$ and its smoothness c.

c presents a high value exactly in the point $(0, 0)$. This point, with index 6, is the corner of $\mathcal{P}_3$, so equation 2.46 correctly recognizes the corner.

Example 4: Let $\mathcal{P}_4$ be a 2D point cloud with 13 elements that define a corner with range noise represented with some spikes and whose values can be seen in table 2.6.

Table 2.6: Elements of point cloud $\mathcal{P}_4$.

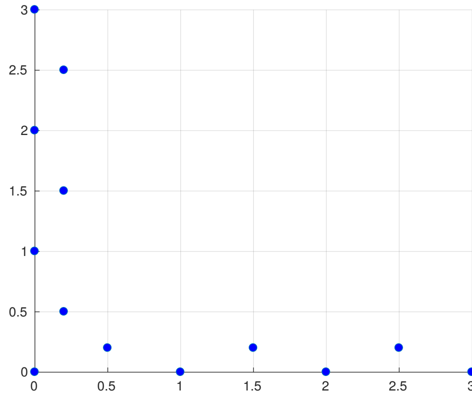
index	1	2	3	4	5	6	7	8	9	10	11	12	13
x	0.00	0.20	0.00	0.20	0.00	0.20	0.05	0.50	1.00	1.50	2.00	2.50	3.00
y	3.00	2.50	2.00	1.50	1.00	0.50	0.05	0.20	0.00	0.20	0.00	0.20	0.00

Point cloud $\mathcal{P}_4$ and the smoothness c for each point can also be graphically represented as shown in Figure 2.15.

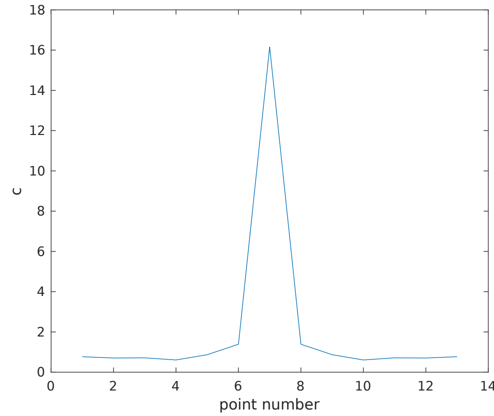
The computation of equation 2.46 for the point cloud $\mathcal{P}_4$ can be seen graphically represented in Figure 2.15b. In this Figure, as also occurred for the point cloud $\mathcal{P}_3$, the least smooth point is located in the middle of the point cloud. In spite of the noise presented in $\mathcal{P}_4$, Figures 2.14b and 2.15b look very similar. In this case, equation 2.46 does not have any problem with range noise.

Example 5: Let $\mathcal{P}_5$ be a 2D point cloud with 14 elements and whose values can be seen in table 2.7. $\mathcal{P}_5$ contains 2 corners in the 4th and 9th point. Additionally, the range noise in form of spikes is only present from point 1 to 5 and from point 10 to 14. Point cloud $\mathcal{P}_5$ and its smoothness c for each point can also be graphically represented as shown in Figure 2.16.

The computation of equation 2.46 for point cloud $\mathcal{P}_5$ can be seen graphically represented in Figure 2.16b. As mentioned before the points that represent corners are located in the



(a) Representation of $\mathcal{P}_4$.

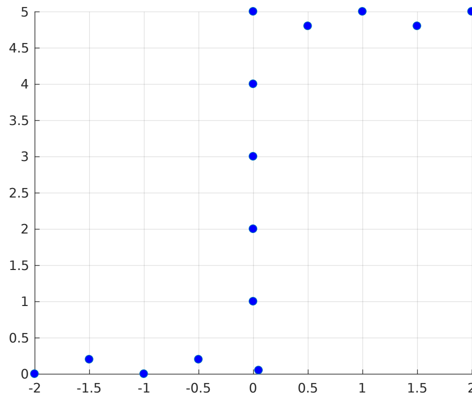


(b) c values for each point in $\mathcal{P}_4$.

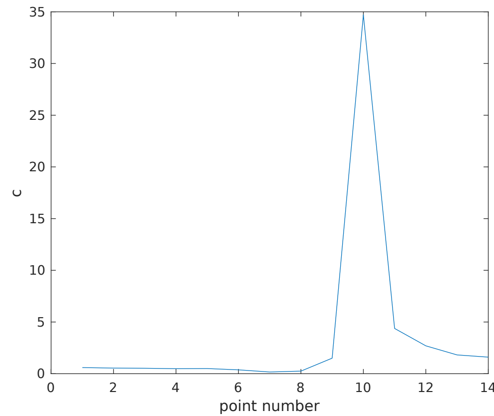
Figure 2.15: Point Cloud $\mathcal{P}_4$ and its smoothness c.

Table 2.7: Elements of point cloud $\mathcal{P}_5$.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x	2.00	1.50	1.00	0.50	0.00	0.00	0.00	0.00	0.00	0.05	-0.50	-1.00	-1.50	-2.00
y	5.00	4.80	5.00	4.80	5.00	4.00	3.00	2.00	1.00	0.05	0.20	0.00	0.20	0.00



(a) Representation of $\mathcal{P}_5$.



(b) c values for each point in $\mathcal{P}_5$.

Figure 2.16: Point Cloud $\mathcal{P}_5$ and its smoothness c.

5th and 10th index of the point cloud. These points are also the ones where the curve is less smooth, however, in Figure 2.16b this lack of smoothness is only detected in the 10th point.

Point cloud $\mathcal{P}_5$ is an example of a real segment of a VLP-16 LiDAR scan that will be seen later in Figure 3.6. If a mobile robot scanned this point cloud, only one feature would be detected instead of two.

As mentioned before, the core of this thesis is to replace the LOAM feature extractor with another feature extraction algorithm. The algorithm chosen after all previous analysis was the Curvature Scale Space (CSS) algorithm. Therefore, in the following chapter, the

CSS algorithm and its adaptation to sparse 3D LiDAR data is explained. Additionally, the magnitude of how this replacement affects the whole LOAM algorithm will be seen through experiments, the results of which will be shown in chapter 4.

2.6 Summary

In this chapter, the most important feature extraction algorithms for point clouds were described. This was achieved by dividing the features extractors into two categories: General feature extractors and features extractors commonly used in SLAM. General feature extractors were discarded because of their computational complexity. In the case of feature extractors commonly used in SLAM, the CSS feature extractor was considered as the most suitable option for the LOAM algorithm because of its robustness to the sensor range noise and the evidence provided in the articles by V. Nguyen et al. [25] and P. Nuñez et al. [26], in which the CSS algorithm demonstrated itself to perform well. This decision was taken after analyzing the LOAM feature extractor with point cloud data similar to that returned by the Velodyne VLP-16 LiDAR.

Chapter 3

Methodology and Implementation

In this chapter, the Curvature Scale Space (CSS) algorithm is presented as a more suitable feature extractor for the LiDAR Odometry and Mapping (LOAM) algorithm. Section 3.1 introduces the term sparsity through the description of the hardware configuration of the Velodyne VLP-16 LiDAR and then it gives an overview of the equipment used in the experiments. This section also describes the methodology used to compare the original LOAM algorithm and the LOAM algorithm with the proposed CSS feature extractor. Section 3.2 describes the implementation details of the CSS algorithm using 3D sparse point clouds. Finally, section 3.3 summarizes the most important points of this chapter.

3.1 Curvature Scale Space with Sparse Point Cloud Data: The New Feature Extractor of the LOAM Algorithm

An element that the CSS feature extractor and the LOAM's feature extractor has in common is that both can be adapted to work with sparse point cloud data. Sparsity in point cloud data is not a well defined term. There are no exact parameters that can define when a point cloud is dense or when it is sparse. Serafin et al. [34] use this term to refer to the point cloud data produced by Velodyne LiDARs. Velodyne LiDARs are known for their sensors such as the VLP-16, HDL-32E and HDL-64E, which are commonly used in robotics. These sensors have a special hardware setup that makes it possible to return points in 3 dimensions. They are formed by N individual lasers placed with a certain inclination angle and with a field of view of 360 degrees.

In this thesis, the Velodyne VLP-16 LiDAR was used for the experiments. This sensor is formed by 16 lasers and the sequence of measurements returned by one individual laser in a 360° rotation is also known as a *ring* or *scan*, so the VLP-16 LiDAR produces 16 scans per each full 3D scan. All these lasers keep rotating to sense their surrounding with a horizontal angular resolution equal to 0.1 to 0.4 degrees. Therefore, the angular resolution in azimuth yields very close points which can be considered, in this case, dense. In the case of the vertical angular resolution, the VLP-16 LiDAR varies from -15 to 15 degrees with a separation of 2

degrees. Hence, we consider this dimension to comprise *sparse* 3D point cloud data. This is because there is a significant inclination between scans that produces a notable separation in the vertical axis and that depends on the range measured by each beam. In Figure 3.1, a diagram of the VLP-16 hardware configuration is shown, where the horizontal resolution is represented by γ and the vertical resolution by θ, r_{i_max} represents the maximum range, which in this case is 100 m.

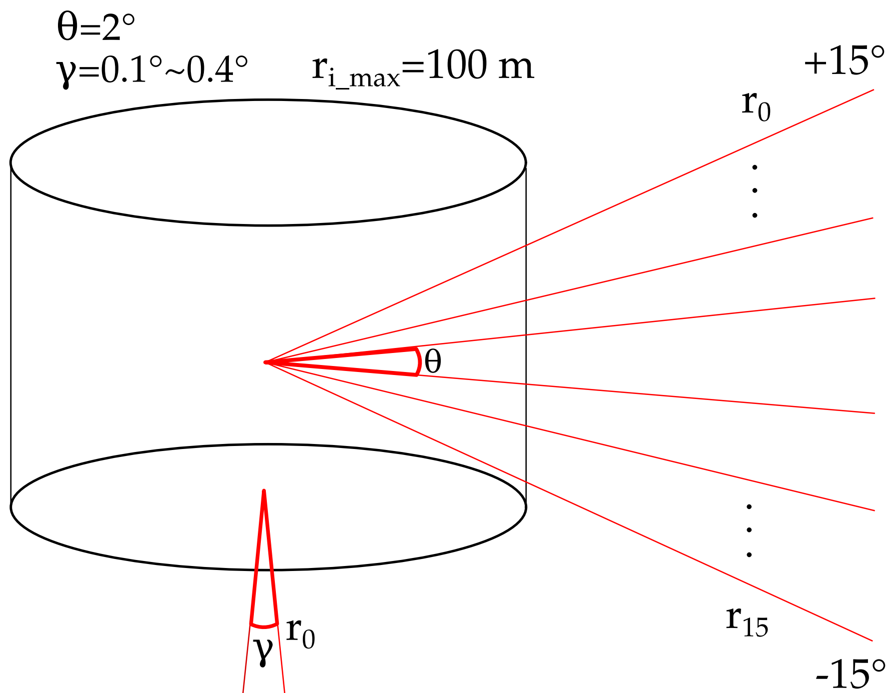


Figure 3.1: VLP-16 hardware configuration.

Before implementing the CSS algorithm with the sparse point cloud data it is important to know where the feature extractors are located in the whole LOAM software system and how this could affect the structure of the experiments. Hence, the insertion of the CSS algorithm as the new feature extractor within the LOAM algorithm is now explained.

Figure 3.2 illustrates the methodology used in the experiments. On the left, this figure shows the hardware used to record the simulated and real data, consisting of a mobile robot Husky A-200, a computer Intel Core i7-7500U CPU @ 2.70GHz $\times$ 4, a Velodyne VLP-16 LiDAR and a Topcon Hyper V GPS. At the center of Figure 3.2, the data obtained with this hardware is compared with two versions of the LOAM algorithm, one which includes the LOAM's feature extractor and the other one that replaces it with the CSS feature extractor. On the right of Figure 3.2, the estimated feature positions and trajectories produced by both algorithms are compared with ground truth features and ground truth trajectories, respectively. For the comparison of features the Optimal Sub-Pattern Assignment (OSPA) metric was used. It is important to note that in this case this comparison was only done for the simulated data. Finally, the comparison of trajectories was carried out using the Absolute Trajectory Error (ATE) metric.

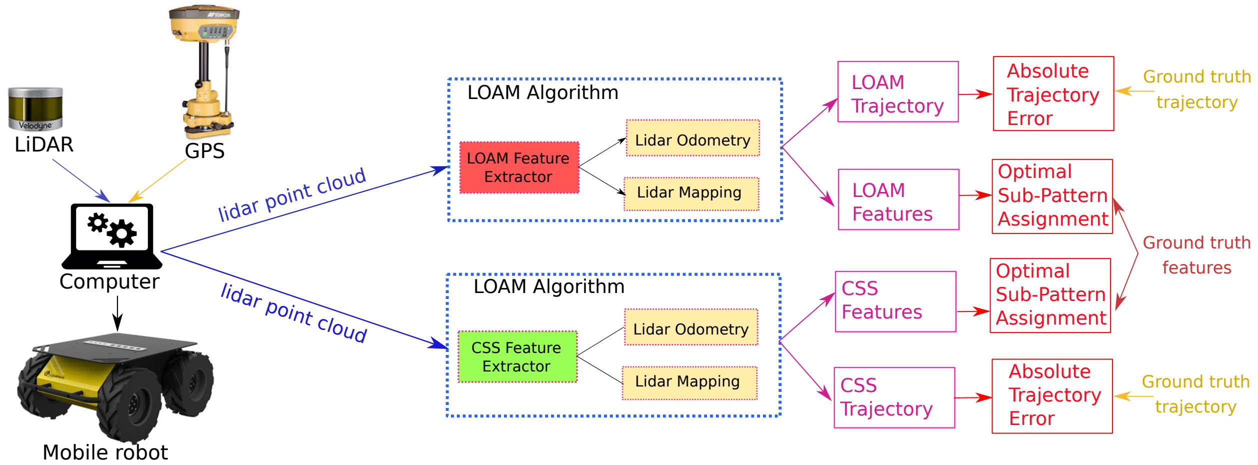


Figure 3.2: Diagram of the methodology used to compare the original LOAM algorithm and the LOAM algorithm with the CSS feature extractor.

3.2 Implementation of the Curvature Scale Space algorithm with 3D Sparse Point Cloud Data

In this section we focus on the details of the adaptation and implementation of the CSS algorithm to function with 3D sparse point cloud data from the Velodyne VLP-16 LiDAR. It is important to mention that, in this implementation, the CSS algorithm only extracts sharp features which correspond to the set $\mathcal{E}_k$. Flat features ($\mathcal{H}_k$) are extracted using the same equation 2.46 that the LOAM's feature extractor uses. Therefore, to process point cloud data from the VLP-16 LiDAR, the task was divided into 4 steps as follows:

1. Divide the 3D point cloud into its 16 scans.
2. Segment the point cloud of each scan using the breakpoint detection algorithm.
3. Pre-compute Gaussian kernels.
4. Implement an algorithm to find the curvature extrema.

As mentioned before, the Velodyne VLP-16 LiDAR is a composition of 16 individual lasers. This configuration was exploited for the use of the CSS algorithm. The idea is that, given a whole point cloud, we divide it into 16 separate point clouds. To complete this task, we can follow two approaches. The first is to compute the elevation angle of each point to determine which point belongs to a scan. The second approach is to consider how the hardware of the VLP-16 LiDAR organizes its data and accordingly separate the points by scans. In this thesis, we followed the second approach to take advantage of all the data sent by the LiDAR.

Each scan of the VLP-16 LiDAR belongs to a 'channel'. Each channel consists on a single 905 nm laser emitter and detector pair. The VLP-16 LiDAR includes within its data the channel from which each point was detected. However, libraries such as the Point Cloud Library (PCL) treat point clouds from different sensors as general point clouds, so there is no data type which includes any notion of a channel as a field in this library. Therefore, in this thesis we considered a new data type that includes the channel each point belongs to.

Each point in the point cloud data type of PCL has generally four fields: x, y, z and intensity. The first three components refer to the Euclidean position of the point and the fourth one is the reflected signal strength of that point. However, if we want to know from which laser a point is detected, it is necessary to extract its channel by accessing the two following bytes of the LiDAR raw data and organizing it according to Table 3.1. Using this table, we can create a new data type composed of five fields: x, y, z, intensity and channel. Performing this task avoid computing extra steps that can affect the processing time of the feature extractor.

Table 3.1: Laser IDs and their angles.

Laser ID	Vertical Angle
0	-15°
1	1°
2	-13°
3	-3°
4	-11°
5	5°
6	-9°
7	7°
8	-7°
9	9°
10	-5°
11	11°
12	-3°
13	13°
14	-1°
15	15°

Each point belonging to a laser ID maintains its 3 Euclidean components, however, the original CSS algorithm is designed to process 2D data. Therefore, to enable this algorithm to process VLP-16 3D LiDAR points, we only consider the Euclidean components x and y of each laser in the computation of the curvature. This approach was chosen because of two reasons. Firstly, related work in the literature [19] [14] [41], which also used Velodyne LiDARs, adopted this approach; and secondly, because most objects present in common environments have high curvature values in the x and y Euclidean components.

The first reason to use each scan separately is based on previous work that also used Velodyne LiDARs. One of these articles is the original implementation of the LOAM algorithm for the VLP-16 LiDAR [41]. In this article, the author used three types of Velodyne LiDARs: VLP-16, HDL-32E and HDL-64E; and on his implementation he divides the Velodyne's point clouds into scans. Another article that considers this approach is by Y. Li and E. Olson, who used this type of LiDAR data to extract vertical corners for different values of z (vertical axis) using the Kanade-Tomasi corner detector [19]. Finally, J. Im used a Velodyne HDL-32E to extract vertical corners based on the Iterative-End-Point-Fit (IEPF) algorithm and using only the eight upper scans of the LiDAR [14].

The second reason to divide whole point cloud into scans is that the majority of point cloud segments present high curvature values that depend mostly in the variation of the Euclidean components x and y. Although point cloud segments of a single scan present non-zero values in the Euclidean component z, the variation in this component commonly does not represent high curvature values and it is not as significant as in the other two components. Additionally, high curvature values existing in the Euclidean components x and y are mainly present in common objects like corners of walls, lamp posts or tree trunks. Other high curvatures values like those that can be extracted if we consider the y and z or x and z components are mainly formed by the intersection of the walls and the floor and the intersection of the walls and the roof, and they are not detected continuously. Finally, considering the computation of curvatures in the three Euclidean components x, y and z would represent an increase in the computational complexity of the CSS algorithm, which could be an issue according to the analysis done in section 2.2.4.

Hence, to divide the whole point cloud from the Velodyne VLP-16 into scans, we organize the point cloud in 16 scans following the previous hardware considerations shown in table 3.1. Figure 3.3 shows a complete point cloud obtained with a Velodyne VLP-16 in an indoor environment. In this representation each scan is labeled with a different color. A closer view of how the whole point cloud was organized in different scans is shown in Figure 3.4.

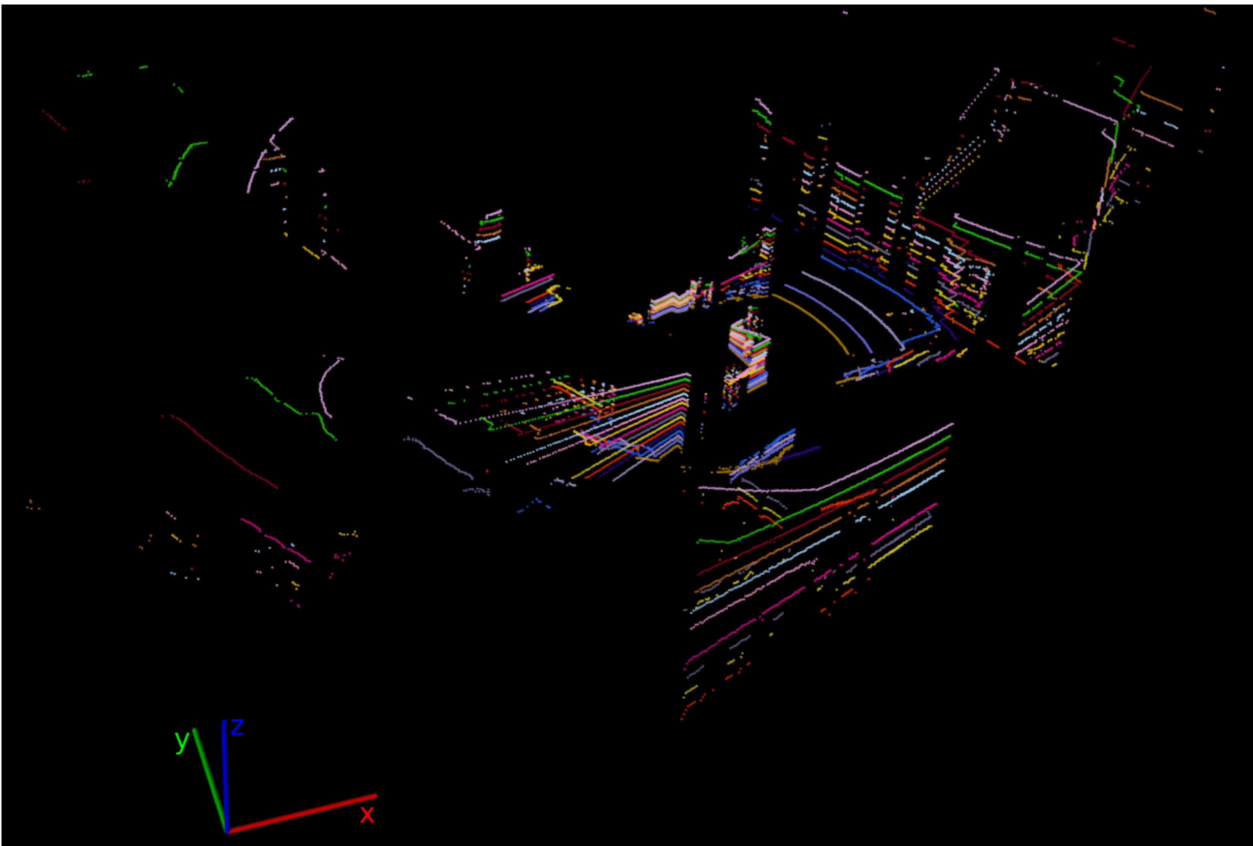


Figure 3.3: Complete point cloud taken with a Velodyne VLP-16 in an indoor environment (each scan is shown with a different color).

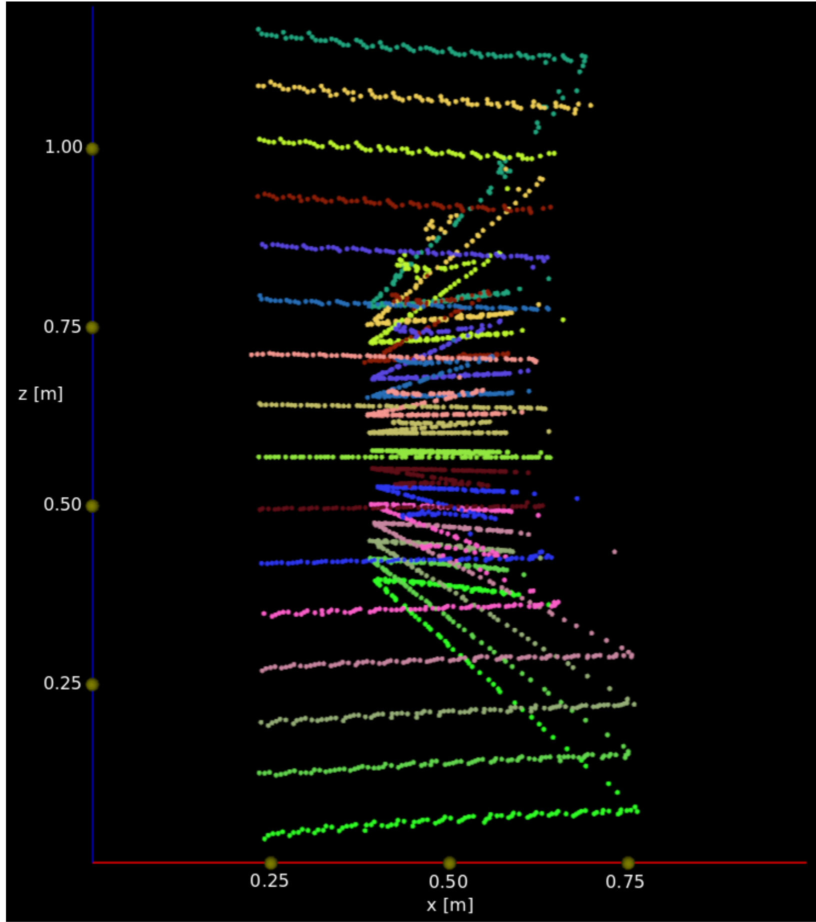


Figure 3.4: Closer view of part of the point cloud of figure 3.3.

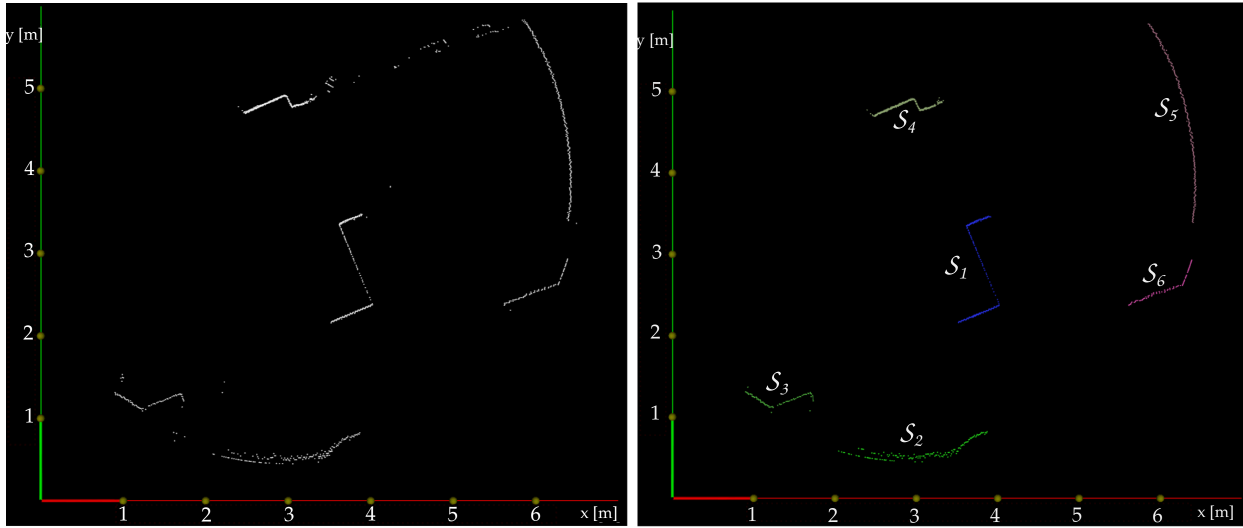
The following task is to divide point clouds from each scan into segments. The objective of dividing each scan into segments is to facilitate the detection of curvature extrema with the CSS algorithm and to decrease its computational complexity. The algorithm used in this thesis is shown in algorithm 5, and the derivation of D_{max} is shown in section 2.4.

The adaptive breakpoint detector needs two inputs: A point cloud of a single scan $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, where p_i represents the ith point of $\mathcal{P}$ with Euclidean components x and y and the range accuracy associated to D_{max}, $\Delta_{D_{max}}$, which was calculated in equation 2.44. This last input is required to determine D_{max} because, according to the original paper [6], taking into account the range accuracy make the breakpoint detector more robust to obstacles close to the sensor. Therefore, $\Delta_{D_{max}}$ was set to 0.0013 m as calculated in the previous chapter.

Also, this algorithm uses a flag b_i for each p_i that indicates when p_i is the beginning or the end of a segment $\mathcal{S}_j$, where j indicates the number of segment. It is important to note that $\mathcal{S}_j \subset \mathcal{P}$ and its number of elements were delimited from 40 to 400 points to avoid segments that did not represent any object and could not consistently appear in time. An example of how a point cloud from one scan projected on a plane xy (Figure 3.5a) is divided into segments of different colors (Figure 3.5b) is shown in Figure 3.5.

Algorithm 5 Adaptive Breakpoint Detector

1: **procedure** AdaptiveBreakpointDetector($\mathcal{P}$)
2: $\quad b_1 \leftarrow FALSE$
3: $\quad D_{max} = r_{i-1}\frac{\sin(\Delta\phi)}{\sin(\lambda-\Delta\phi)} + \Delta_{D_{max}}$
4: $\quad$ **for** $i \leftarrow 2$ to n **do**
5: $\quad\quad$ **if** $||p_i - p_{i-1}|| > D_{max}$ **then**
6: $\quad\quad\quad b_i \leftarrow TRUE$
7: $\quad\quad\quad b_{i-1} \leftarrow TRUE$
8: $\quad\quad$ **else**
9: $\quad\quad\quad b_i \leftarrow FALSE$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: **end procedure**



(a) 2D projection of the first scan of the point cloud of Figure 3.3.

(b) Point cloud of Figure 3.5a divided into segments.

Figure 3.5: Segmentation of one scan using the Adaptive Breakpoint Detector.

Figure 3.5b contains 6 point cloud segments labeled from $\mathcal{S}_1$ to $\mathcal{S}_6$. A characteristic that is common in this point cloud is the range noise present in each of these segments, even though, some of them represent flat surfaces. An example of a segment with range noise can be seen in figure 3.6 which is a zoomed view of the point cloud segment $\mathcal{S}_1$ projected on a plane xy.

At this point, we have several segments per scan. With the data in this form, we can use the CSS algorithm. To use this algorithm, first we have to parametrize the curve segment as proposed by [21] to obtain different t_i values that vary from 0 to 1 as shown in equations 3.1 and 3.2.

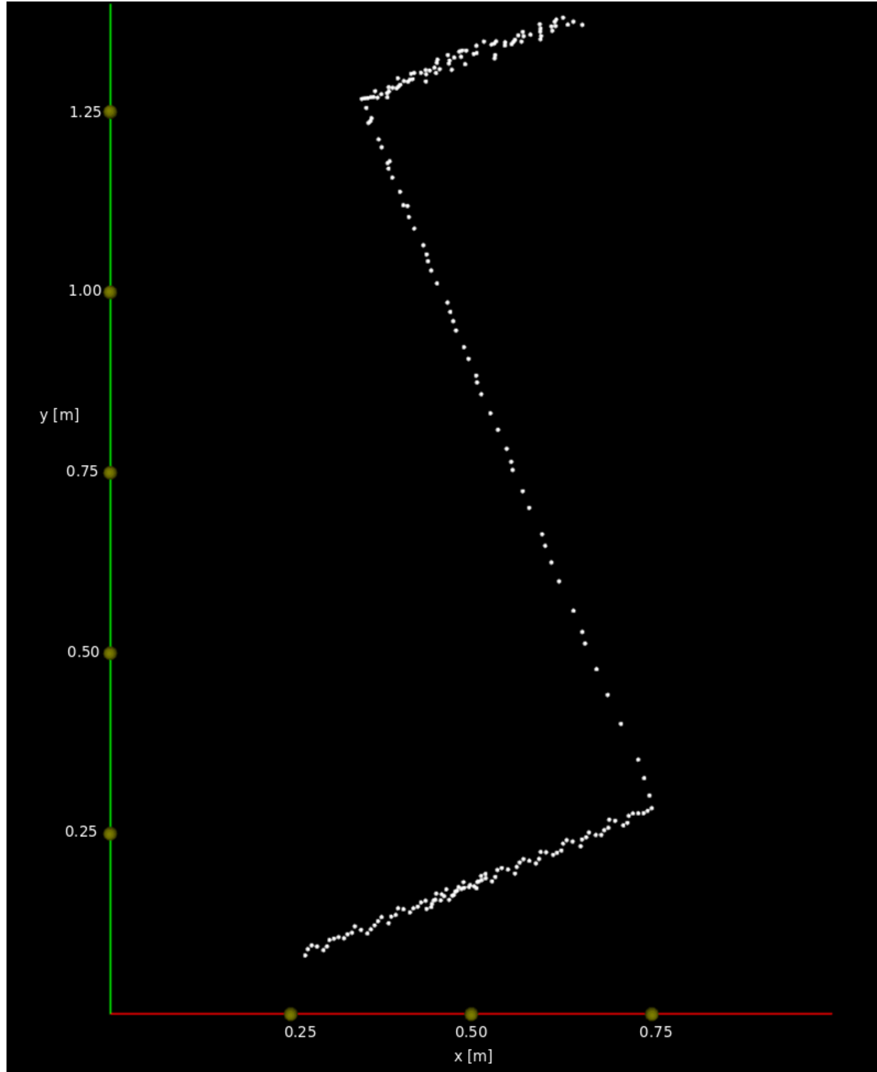


Figure 3.6: 2D projection of the point cloud segment $\mathcal{S}_1$.

$$d_1 = 0 \quad \text{and} \quad d_i = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}, \tag{3.1}$$

$$D_p = \sum_{i=1}^{n} d_i \qquad t_1 = 0 \qquad t_i = \frac{1}{D_p} \sum_{j=1}^{i} d_j. \tag{3.2}$$

In previous equations, the sub-index i varies from 1 to n, where n is the number of points of the curve segment $\mathcal{S}_j$. After the parametrization, the point cloud segment is expressed in its Euclidean components x and y as a function of t using the equation

$$C = \{x(t), y(t)\}. \tag{3.3}$$

It is then necessary to convolve $x(t)$ and $y(t)$ with the Gaussian derivatives using

$$\dot{X}(t,\sigma) = \frac{\partial X(t,\sigma)}{\partial t} = \frac{\partial[x(t) * g(t,\sigma)]}{\partial t} = x(t) * \frac{\partial g(t,\sigma)}{\partial t} \tag{3.4}$$

$$\ddot{X}(t,\sigma) = \frac{\partial^2 X(t,\sigma)}{\partial t^2} = x(t) * \frac{\partial^2 g(t,\sigma)}{\partial t^2} \tag{3.5}$$

$$\dot{Y}(t,\sigma) = \frac{\partial Y(t,\sigma)}{\partial t} = y(t) * \frac{\partial g(t,\sigma)}{\partial t} \tag{3.6}$$

$$\ddot{Y}(t,\sigma) = \frac{\partial^2 Y(t,\sigma)}{\partial t^2} = y(t) * \frac{\partial^2 g(t,\sigma)}{\partial t^2}. \tag{3.7}$$

In equations 3.4, 3.5, 3.6 and 3.7, the operator $*$ represents convolution and the function $g(t,\sigma)$ is a Gaussian function that depends on t and its standard deviation σ. Finally, the results of previous convolutions are used to compute the curvature κ of each segment in the 16 scans using the equation

$$\kappa(t,\sigma) = \frac{\dot{X}(t,\sigma)\ddot{Y}(t,\sigma) - \dot{Y}(t,\sigma)\ddot{X}(t,\sigma)}{(\dot{X}(t,\sigma) + \dot{Y}(t,\sigma))^{3/2}}. \tag{3.8}$$

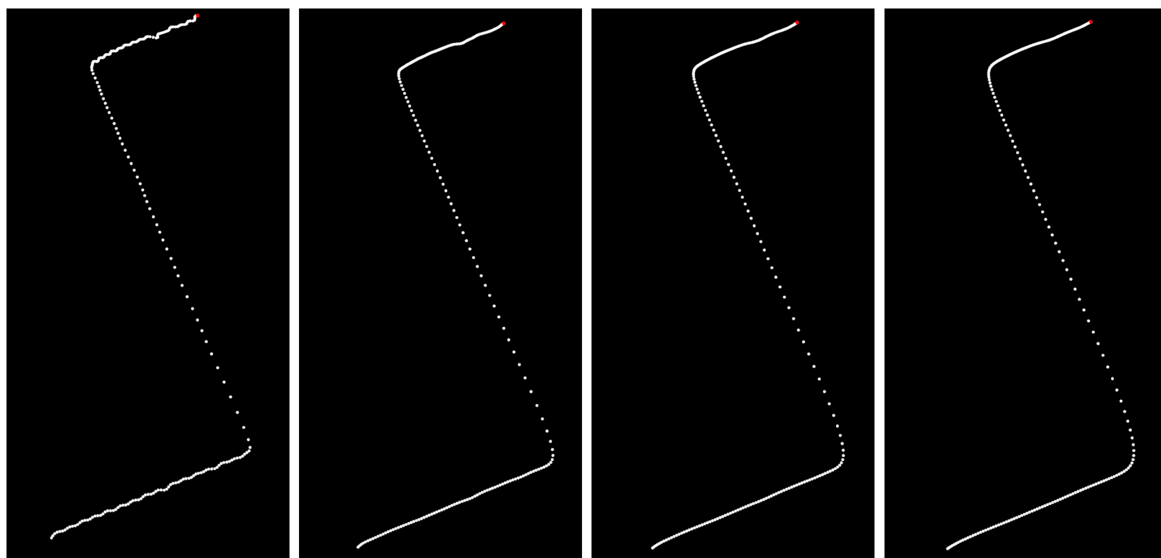
To apply the Gaussian filters, the original Curvature Scale Space article [22] does not specify the necessary number of scales. However, the paper recommends to apply equations 3.4, 3.5, 3.6 and 3.7 to the segments until no curvature extrema are found. More specifically, this paper gives an example with a Gaussian filter with 6 scales from $\sigma = 4$ to $\sigma = 128$. Their article also cites T. Poggio and A. Yuille [29], who state that the number of Gaussian filters applied to a curve segment, depends on the inflection points found at higher levels of smoothing.

In this thesis, we followed the recommendations of the original Curvature Scale Space article to apply the Gaussian filters. Therefore, to apply the Gaussian filters to the curve segments, we defined 8 discretized Gaussian kernels with different numbers of elements. Let G be a set that contains the elements of discretized $g(t,\sigma)$, so the number of elements in G is defined by

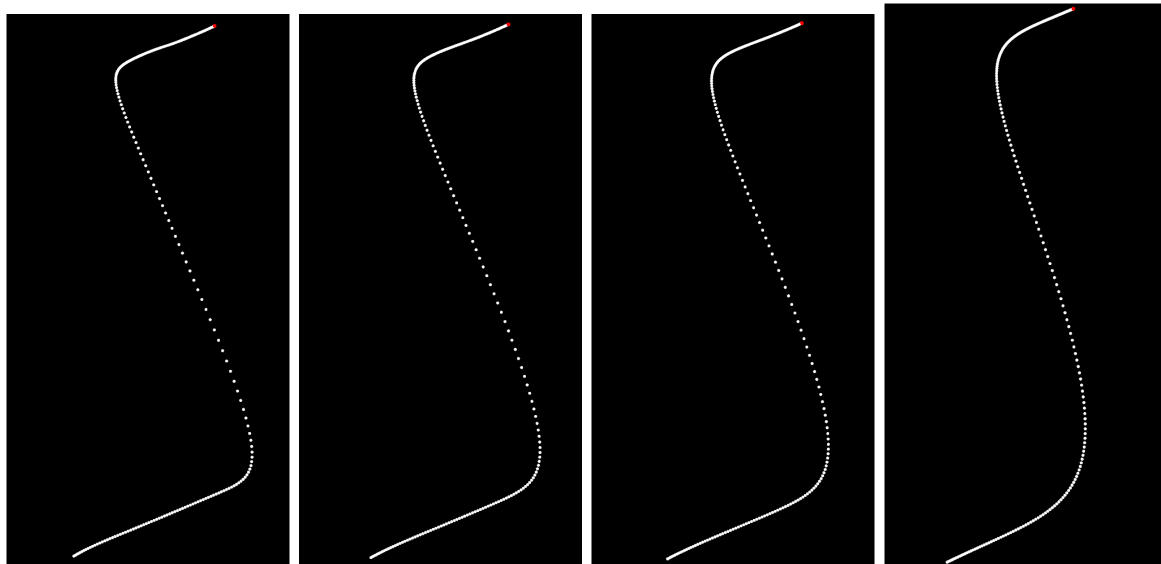
$$|G| = |\mathcal{S}_j| r. \tag{3.9}$$

In equation 3.9, $r \in (0,1)$ and is used to have kernels that change according the quantity of points in the segments. This was carried out because the number of points on each curve segment varied and having fixed widths of Gaussian kernels for all curve segments did not result in correctly smoothed curves. Specifically, we used kernels from $r = 0.05$ to $r = 0.8$. The different levels of Gaussian smoothing applied to the point cloud segment $\mathcal{S}_1$ are shown

in Figure 3.7. This results in 8 different curvature functions κ that can be seen in Figure 3.8, where curvature extrema are marked with red circles.

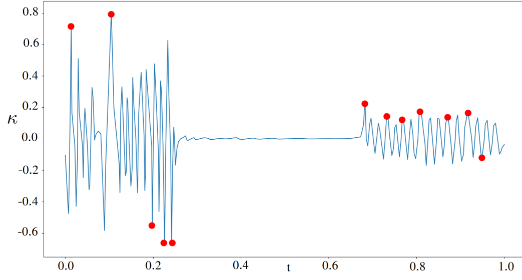


(a) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.05$). (b) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.1$). (c) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.15$). (d) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.2$).

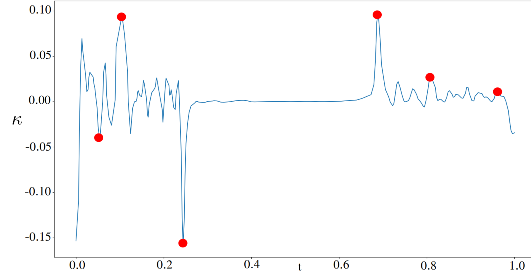


(e) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.3$). (f) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.4$). (g) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.5$). (h) Smoothed curve segment $\mathcal{S}_1$ ($r = 0.8$).

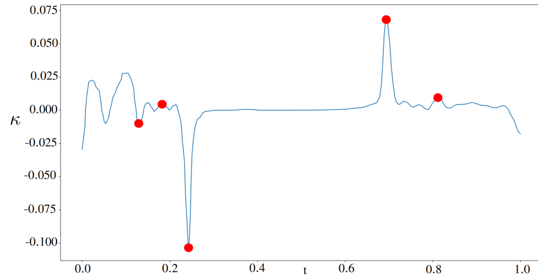
Figure 3.7: 2D projection of the point cloud segment $\mathcal{S}_1$ (Figure 3.6) at various levels of Gaussian smoothing.



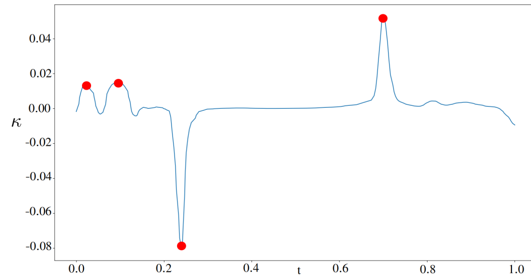
(a) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.05$.



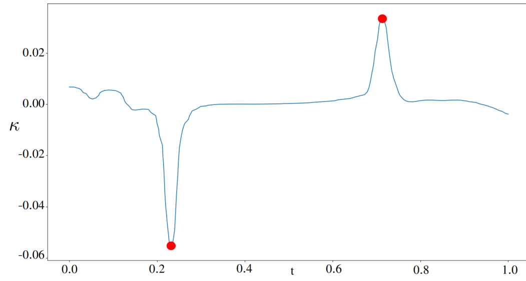
(b) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.10$.



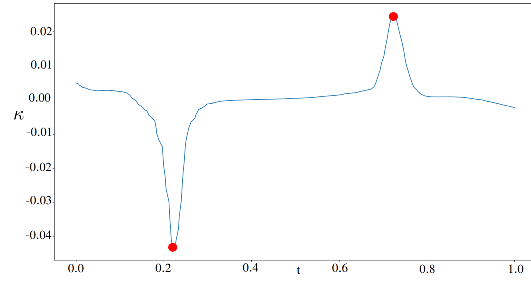
(c) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.15$.



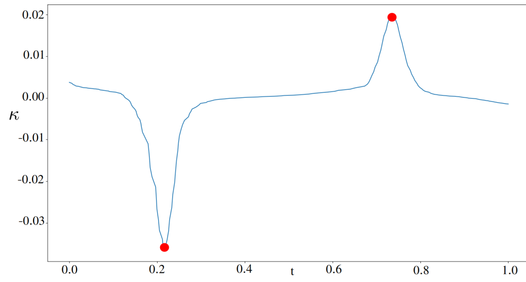
(d) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.20$.



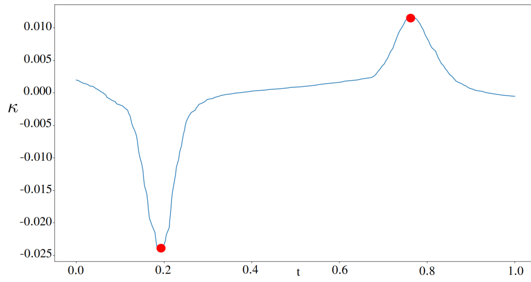
(e) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.30$.



(f) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.40$.



(g) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.50$.



(h) κ for smoothed segment $\mathcal{S}_1$ with $r = 0.80$.

Figure 3.8: Curvature functions κ at various levels of the smoothing for segment $\mathcal{S}_1$.

The task of finding the curvature extrema in functions κ should be time efficient. Therefore, an algorithm that does not increase the computational complexity of the scale space is required. In this thesis, we used a simple approach proposed by R. Madhavan and H.Durrant-

Whyte [21] given by

$$|\kappa_{i-1}| < |\kappa_i| > |\kappa_{i+1}| \quad \text{and} \quad |\kappa_{i-2}| < |\kappa_i| > |\kappa_{i+2}|. \tag{3.10}$$

These conditions evaluate the current curvature value κ_i and compares it with its two neighbors in the left and right side. Finally, we have the curvature scale space that determines which points and at which σ the curvature extrema were found. This space can be represented as a binary image as shown in figure 3.9, where the shaded part represents at which (t, σ) the curvature extrema were found according to the red circles marked in Figure 3.8. The most repeated curvature extrema in Figure 3.9 can be found at approximately $t = 0.2$ and $t = 0.7$. In this case these points represent the corners of the point cloud in figure 3.6.

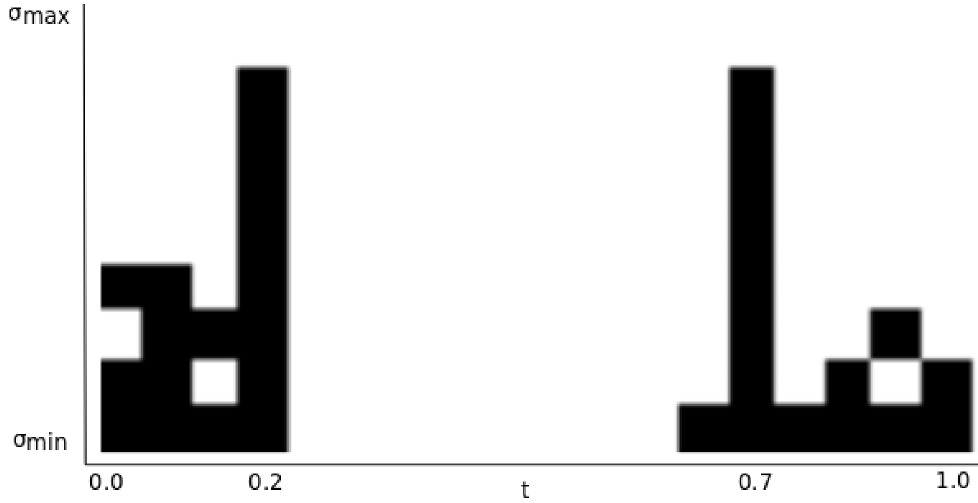


Figure 3.9: Curvature scale space created with the functions κ shown in Figure 3.8.

To extract the points found previously at $t = 0.2$ and $t = 0.7$, we need to iterate the whole scale space. To do this, we have to consider an important note made by Asada and Brady [2], which is also mentioned in [21]. This note states that curvature changes found at different scales, σ, are reliably localized at σ_{min}. However, if they are found at σ_{min} and do not persist across many scales, they are less likely to determine the shape of the curve.

Additionally, R. Madhavan and H.Durrant-Whyte [21] complements the previous idea saying that the identification of the final features occurs at the coarsest scale, $\sigma = \sigma_{max}$, and they are realiably localized at the finest scale, $\sigma = \sigma_{min}$. Moreover, they say that when tracking an extremum, if a choice needs to be made between two extrema at the next finer scale, the extremum with the higher curvature value must be chosen.

Previous ideas from the papers mentioned before indicate mathematically how to find the curvature extrema, however, what they propose are not given in an algorithmic form. Therefore, we took those ideas to develop an algorithm to track the curvature extrema in the scale space, shown in algorithm 6. This algorithm takes as inputs a point cloud segment $\mathcal{S}$ and the curvature extrema data (t, κ) found in the different scales, $\mathcal{F}_i$. Then, it iterates at all scales from σ_{max} to σ_{min} to count how many times features found at the highest scale are repeated at the lowest scale. Since the index, t, of the points found at different scales can

vary, their similarity is found using a threshold which is the maximum between $|t_i - t_{i-1}|$ and $|t_i - t_{i+1}|$. Additionally, the counter and indices are represented as sequences D and I, respectively. Finally, the counter D is evaluated to store only the points with index I_i which have been found in $\mathcal{F}_i$ in as many scales as a second threshold allows. The second threshold indicates in how many scales a curvature extremum must be found to be considered a feature. Finally, features are stored and returned in the point cloud $\mathcal{E}$.

Algorithm 6 Find curvature extrema in scale space

1: **procedure** FindFeatures($\mathcal{S} = \{s_0, s_1, ..., s_n\}$ and $\mathcal{F}_i, i \in [\sigma_{min}, \sigma_{max}]$)
2: $\quad \mathcal{E} = \{\}$
3: $\quad \mathcal{G} = \mathcal{F}_{\sigma_{max}} = \{g_1, g_2, ..., g_{|\mathcal{F}_{\sigma_{max}}|}\}$
4: $\quad D = \{0_1, 0_2, ..., 0_{|\mathcal{G}|}\}$
5: $\quad I = \{\}$
6: $\quad$ **for** all scales starting from $i = \sigma_{max}$ to σ_{min} **do**
7: $\quad\quad$ **for** $j = 1$ to $|\mathcal{G}|$ **do**
8: $\quad\quad\quad$ **if** $g_j \in \mathcal{F}_{i-1}$ according to a threshold 1 **then**
9: $\quad\quad\quad\quad a \leftarrow$ index of $\mathcal{F}_{i-1}$ in $\mathcal{S}$
10: $\quad\quad\quad\quad D_a \leftarrow D_a + 1$
11: $\quad\quad\quad\quad$ Stack a to I
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end for**
14: $\quad$ **end for**
15: $\quad$ **for** $i = 0$ to $|\mathcal{G}|$ **do**
16: $\quad\quad$ **if** $D_i >$ threshold 2 **then**
17: $\quad\quad\quad b \leftarrow I_i$
18: $\quad\quad\quad$ Stack s_b to $\mathcal{E}$
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: $\quad$ **return** $\mathcal{E}$
22: **end procedure**

3.3 Summary

In this chapter, two important parts of this thesis were described. Firstly, the methodology used to insert the CSS feature extractor in the LOAM algorithm was presented. In this part, the metrics used to compare the original LOAM algorithm and the LOAM algorithm with the CSS feature extractor were shown in Figure 3.2. Secondly, the details of the implementation of the CSS feature extractor to process 3D VLP-16 LiDAR data were described. This last part was achieved by using the adaptive breakpoint detector and processing each scan of the VLP-16 LiDAR separately. The results of the execution of these algorithms will be described in the following chapter.

Chapter 4

Results

This chapter presents the results of the comparison of the LOAM's feature extractor and the CSS feature extractor in isolation from the complete LOAM algorithm and the results of the comparison of the integration of both feature extractors with the LOAM algorithm. From here on, the LOAM algorithm using its original feature extractor will be called the Original LOAM algorithm and the LOAM algorithm using the CSS feature extractor will be called the CSS LOAM algorithm. Test scenarios were carried out with simulated data, real data and the KITTI Visual Odometry Benchmark Dataset [11]. For the comparisons, we used the Optimal Sub-Pattern Assignment (OSPA) and the Absolute Trajectory Error (ATE) metrics to measure the feature estimation errors and trajectory errors, respectively.

This chapter is structured as follows. Section 4.1 describes how the OSPA metric works. Section 4.2 describes how the ATE metric works. Section 4.3 presents the results of testing both algorithms with simulated data. Section 4.4 presents the results of testing both algorithms with real data. Finally, section 4.5 presents the results of testing both algorithms with the KITTI Visual Odometry Benchmark Dataset.

4.1 The Optimal Sub-Pattern Assignment (OSPA)

The Optimal Sub-Pattern Assignment (OSPA) [33] is a metric designed for multi-object systems. The OSPA metric computes a distance between two finite sets and is used in robotics and other areas [15] [24] [17]. The OSPA distance, ($\bar{d}_p^{(c)}(X,Y)$), between two finite sets $X = \{x_1, x_2, ..., x_m\}$ and $Y = \{y_1, y_2, ..., y_n\}$ is computed by

$$\bar{d}_p^{(c)}(X,Y) = \left(\frac{1}{n}\left(\min_{\pi\in\Pi_n}\sum_{i=1}^{m} d^{(c)}(x_i, y_{\pi(i)})^p + c^p(n-m)\right)\right)^{1/p}, \tag{4.1}$$

where $d^{(c)}(x_i, y_{\pi(i)}) = \min(c, d(x_i, y_{\pi(i)}))$ denotes the distance between x_i and $y_{\pi(i)}$ cutoff at $c > 0$, Π_n is the set of permutations on $\{1, 2, ..., n\}$ for any $n \in \mathbb{N} = \{1, 2, ...\}$ and $1 < p < \infty$

is used to compute the p-th order average among the distances.

In this work, the OSPA metric was used for the simulated data, where the ground truth of the features was available. In that scenario, we will present the errors produced by the features extracted with the LOAM's feature extractor and the the CSS feature extractor.

4.2 The Absolute Trajectory Error (ATE)

The Absolute Trajectory Error (ATE) is a metric from *The Rawseeds Project*[1]. The ATE metric evaluates the error produced at each instant of time of the Algorithm Under Test (AUT) and the Ground-Truth (GT). According to the official documentation of this metric, the ATE algorithm consist on the steps described in Algorithm 7.

Algorithm 7 Compute Absolute Trajectory Error

1: **procedure** ATE(T, X^{GT}, X)
2: **for** every time instant $t_j \in T$ when a sample of X^{GT} is available **do**
3: Evaluate the pose of the robot at t_j by using the AUT.
4: compute the translation error $d_j = ||trans(x_j) - trans(x_j^{GT})||$
5: compute $\bar{d_j}$ (mean of translation errors $\{d_j\}$)
6: compute σ_{d_j} (standard deviation of translation errors $\{d_j\}$)
7: compute $a_{d_j,3\sigma}$, $b_{d_j,3\sigma}$ (extremes of the 3σ interval of the translation error d_j)
8: ATE $= [\bar{d_j}\ \sigma_{d_j}\ a_{d_j,3\sigma}\ b_{d_j,3\sigma}]^T$
9: **end for**
10: **return** ATE
11: **end procedure**

In this algorithm variable j is used to iterate the inputs. These inputs can be taken as three sets: $T = \{t_1, t_2, ..., t_n\}$, $X^{GT} = \{x_1^{GT}, x_2^{GT}, ..., x_n^{GT}\}$ and $X = \{x_1, x_2, ..., x_n\}$, which represent the time stamps, the ground truth positions and the positions generated by the AUT, respectively.

4.3 Simulated Data

The purpose of the use of the simulated data is to control the range noise added the to measurements of the simulated LiDAR. For this scenario, we used the Robot Operating System (ROS) along with the simulation software Gazebo to simulate 3 components: the Husky A-200 robot platform, the Velodyne VLP-16 LiDAR and the GPS.

To simulate the Husky A-200 robot platform, all the packages[2] corresponding to this robot platform were installed. The VLP-16 LiDAR was simulated using a package dedicated to the

[1] `http://www.rawseeds.org/rs/methods/view//9`
[2] `http://wiki.ros.org/Robots/Husky`

Velodyne LiDAR from Data Speed Inc.[3]. Finally, the GPS was simulated using the plugin *p3d_ base_ controller* of Gazebo.

With all the components of the robot configured, we divided the simulation in two more scenarios:

1. A scenario with the LiDAR stationary.
2. A scenario with the robot in motion.

4.3.1 Scenario with the LiDAR stationary

The objective of testing the simulated VLP-16 LiDAR in this scenario is to evaluate how accurate the LOAM's feature extractor and the CSS feature extractor localize the keypoints and if these features can be consistently detected in time. Both feature extractors were tested in isolation from the complete LOAM algorithm.

The predefined environment was composed of the robot platform mounted with the LiDAR at the origin and three walls that formed two corners marked with light blue lines as shown in Figure 4.1. These two corners represent sharp features, which can be taken as the location where the smoothness is the lowest (LOAM's feature extractor) or where the curvature is the highest (CSS feature extractor). Since the robot and the LiDAR are static, the ground truth locations of the features are easy to obtain. Let $e_{(i,k)}$ be the ground truth location of the ith sharp feature at sweep k, so this scenario contains two sharp features (corners): $e_{(1,k)} = (3\ 3\ z)^T$ and $e_{(2,k)} = (3\ 10\ z)^T$.

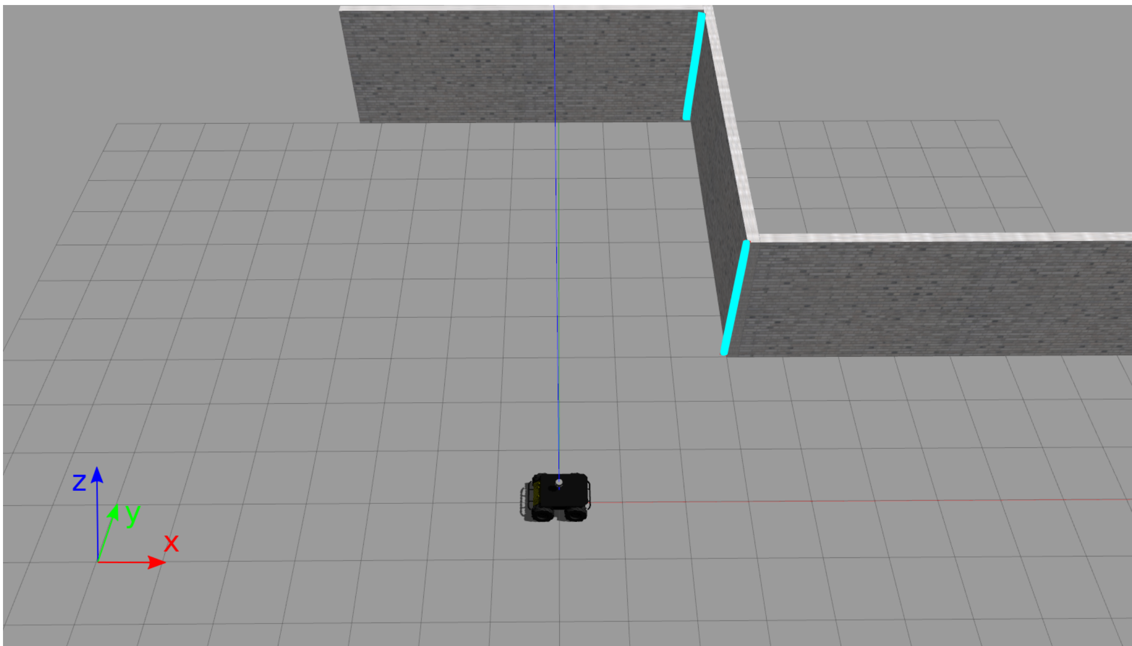


Figure 4.1: Scenario with the LiDAR stationary.

[3] `https://bitbucket.org/DataspeedInc/velodyne_simulator`

It is important to remember that the VLP-16 LiDAR is composed of 16 lasers and the set of measurements of each laser corresponding to 360 degrees is called a scan. So, first we have to determine how many scans of the simulated LiDAR can sense the sharp features. For the feature 1 ($e_{(1,k)}$) the corner was detected in 10 out of 16 scans and for the feature 2 ($e_{(2,k)}$) it was detected in 8 out of 16 scans.

Additionally, we have to take into account that according to the Velodyne VLP-16 datasheet, the sampling frequency of the LiDAR is 10 Hz, approximately, which means that all lasers complete 360 degree rotations 10 times per second. As mentioned in Chapter 2, in the case of the VLP-16 LiDAR, $\mathcal{P}_k$ is the set of points obtained from the 16 lasers in a full 360 rotation, where k indicates the number of sweeps. Therefore, in one second the LiDAR returns 10 $\mathcal{P}_k$ ($k = \{1, 2, ..., 10\}$) which is equal to $300,000$ points, approximately.

Let $\hat{e}_{(1,k)}$ and $\hat{e}_{(2,k)}$ be the estimated positions of the sharp features $e_{(1,k)}$ and $e_{(2,k)}$, respectively; and $\hat{\mathcal{E}}_{i,[1-50]}$ the set of estimated features $\hat{e}_{(i,k)}$ during 50 sweeps (five seconds) for $i = 1, 2$ ($\hat{e}_{(i,k)} \in \hat{\mathcal{E}}_{i,[1-50]}$). It is important to note that a perfect feature extractor should localize 10 $\hat{e}_{(1,k)}$ per sweep k (number of scans where $e_{(1,k)}$ is approximately localized) and 8 $\hat{e}_{(2,k)}$ per sweep k (number of scans where $e_{(2,k)}$ is approximately localized). In other words, the total number of features estimated in 50 sweeps should be 500 for $\hat{e}_{(1,k)}$ and 400 for $\hat{e}_{(2,k)}$.

Regarding LOAM's feature extractor and the CSS feature extractor, first we have to know the correct parameters to use in both algorithms. To do this, it is useful to plot a Receiver Operating Characteristic (ROC) curve. Therefore, Figure 4.2 shows the ROC curves related to LOAM's feature extractor and the CSS feature extractor. This graph was plotted for three different Gaussian noise sources with range noise standard deviations σ_r : 0.01 m, 0.02 m, 0.03 m; each one represented with a different color. It is important to note that the ideal number of true positives is 18, which is the total number of ground truth features mentioned earlier. In the case of LOAM's feature extractor, the curvature threshold was varied from 0.01 to 0.5. In the case of the CSS feature extractor, the number of scales related to the width of the Gaussian kernels was varied from 2 to 16. An important finding related to these ROC curves is that the range noise negatively impacts LOAM's feature extractor more than the CSS feature extractor. Therefore, for the rest of experiments, the curvature threshold used in LOAM's feature extractor was 0.1 and the number of scales used in the CSS feature extractor was 8.

Hence, the experiment consisted of setting the LiDAR stationary to sense the predefined environment for five seconds (50 sweeps) to find out how close the estimated features are from the ground truth and how many of them are correctly detected in this time period. This experiment was repeated for 5 different Gaussian noise sources with range noise standard deviations σ_r : 0.01 m, 0.015 m, 0.02 m, 0.25 m, 0.03 m. For each noise source, three variables were evaluated as follows:

1. the standard deviation of $\hat{\mathcal{E}}_{(i,[1-50])}$ in x and y ($\sigma_x^{\hat{\mathcal{E}}_{(i,[1-50])}}$ and $\sigma_y^{\hat{\mathcal{E}}_{(i,[1-50])}}$) for $i = 1, 2$,
2. the number of detected features per sweep for each feature (true positives),
3. the total number of false positives.

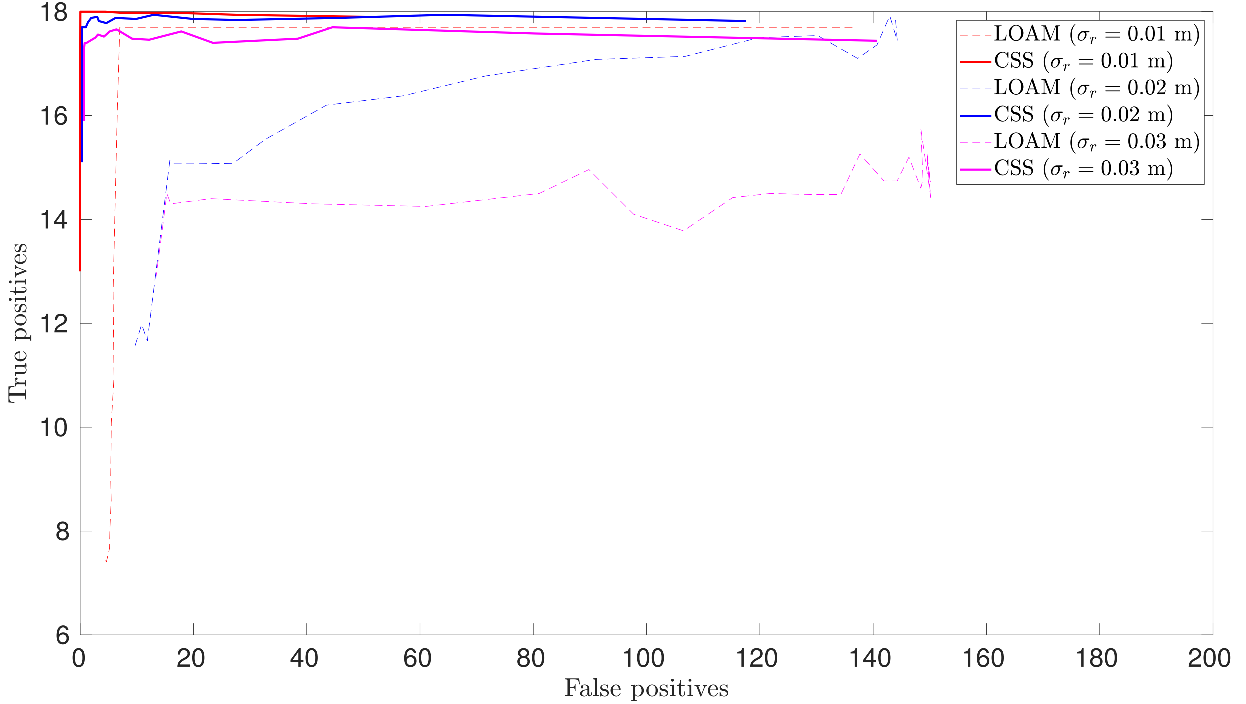


Figure 4.2: True positives vs false positives (ROC curve) for three σ_r evaluated with the LOAM's feature extractor and the CSS feature extractor.

Variable 1 is useful to see how close all the estimated features in $\hat{\mathcal{E}}_{(i,[1-50])}$ are, because the ideal value for $\sigma_x^{\hat{\mathcal{E}}_{(i,[1-50])}}$ and $\sigma_y^{\hat{\mathcal{E}}_{(i,[1-50])}}$ should be zero. Variable 2 indicates if features found by the algorithms can be consistently detected in all sweeps, this value is calculated by dividing the total number of elements in $\hat{\mathcal{E}}_{(1,[1-50])}$ and $\hat{\mathcal{E}}_{(2,[1-50])}$ by 50 (sweeps). The number of features per sweep also indicates when a feature was not detected during all sweeps. Finally, the total number of false positives shows how many estimated features were found apart from $e_{(1,k)}$ and $e_{(2,k)}$ in each sweep, which do not correspond to the ground truth features.

Figures 4.3 and 4.4 show the standard deviation of the estimated features (variable 1) detected with the LOAM's feature extractor and the CSS feature extractor. In these figures, it is important to note that when σ_r is high, $\sigma_x^{\hat{\mathcal{E}}_{(i,[1-50])}}$ and $\sigma_y^{\hat{\mathcal{E}}_{(i,[1-50])}}$ are high, too. Additionally, this figure indicates that estimated features extracted with the CSS feature extractor in 50 sweeps vary in their position less than features estimated with the LOAM's feature extractor.

Figure 4.5 shows the number of estimated features per sweep (variable 2) extracted with the LOAM's feature extractor and the CSS feature extractor in 50 sweeps. This graph is useful to see how many missed detections were found for each feature because, as stated earlier, the correct number of features per sweep is 10 for feature 1 and 8 for feature 2. Therefore, this figure indicates that the CSS algorithm was the feature extractor closer to the correct values.

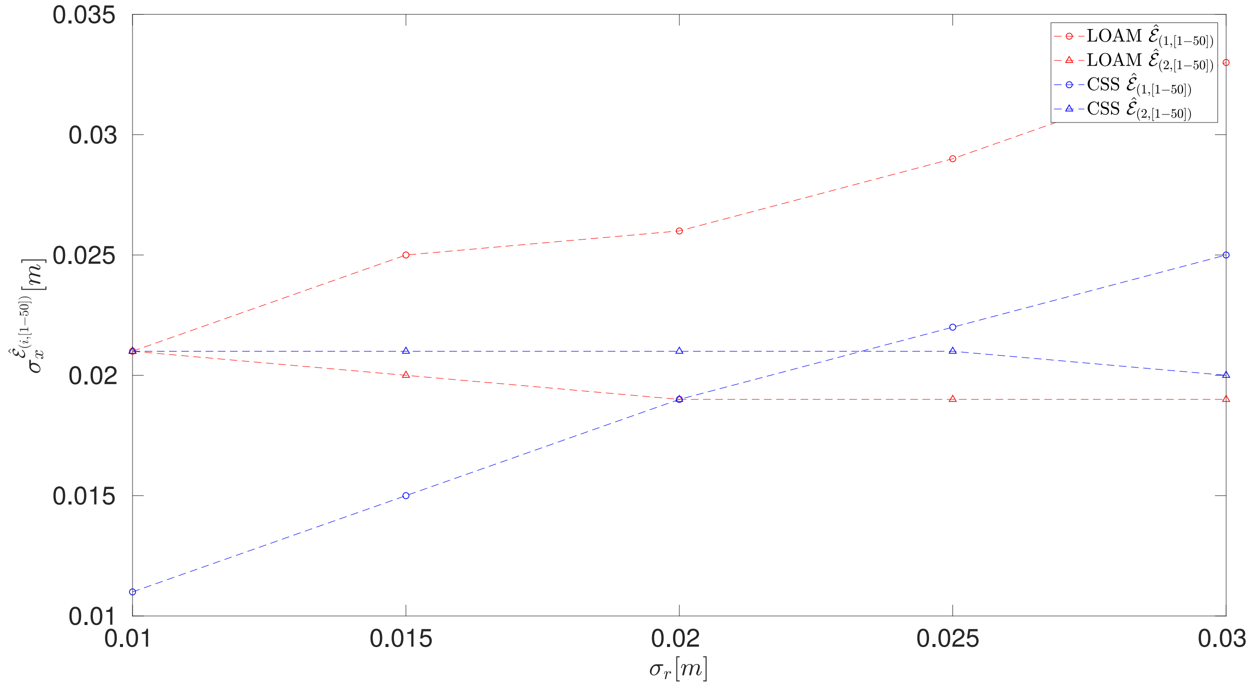


Figure 4.3: Standard deviation in x of estimated features in 50 sweeps.

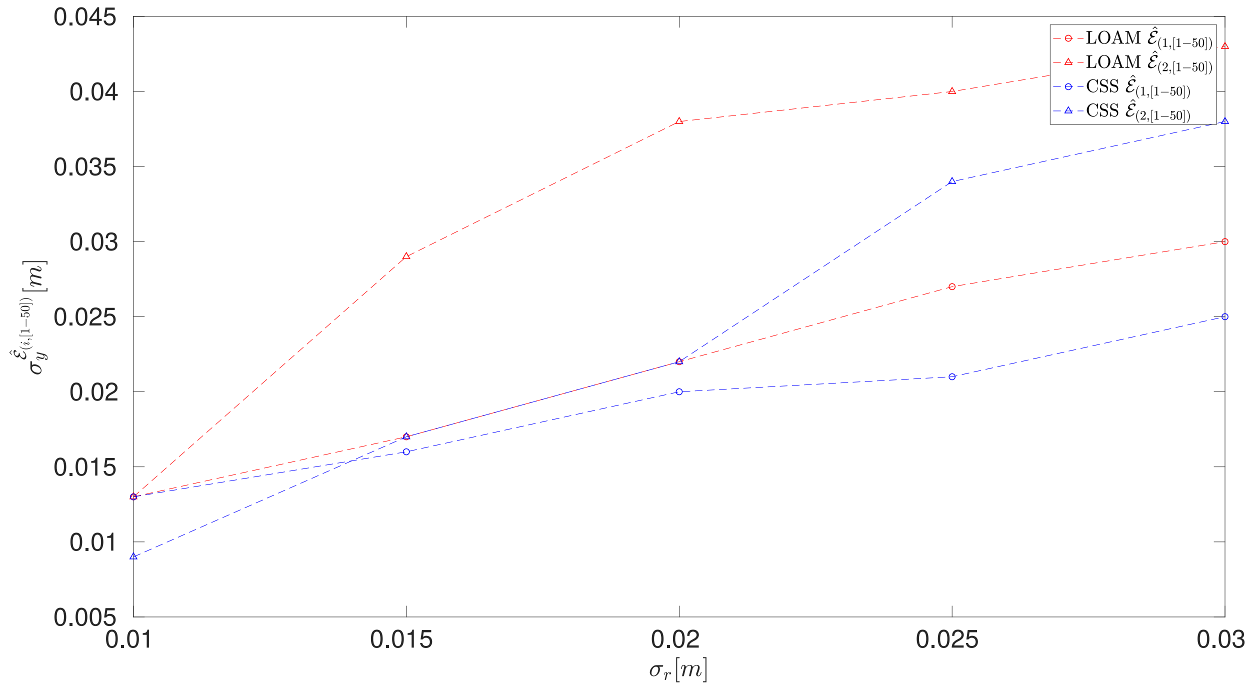


Figure 4.4: Standard deviation in y of estimated features in 50 sweeps.

Finally, Figure 4.6 shows the number of false positive features (variable 3) per sweep extracted with the LOAM's feature extractor and the CSS feature extractor in 50 sweeps. In other words, this figure indicates the number of estimated features detected incorrectly. This figure shows that the LOAM's feature extractor detected incorrectly more sharp features than the CSS algorithm.

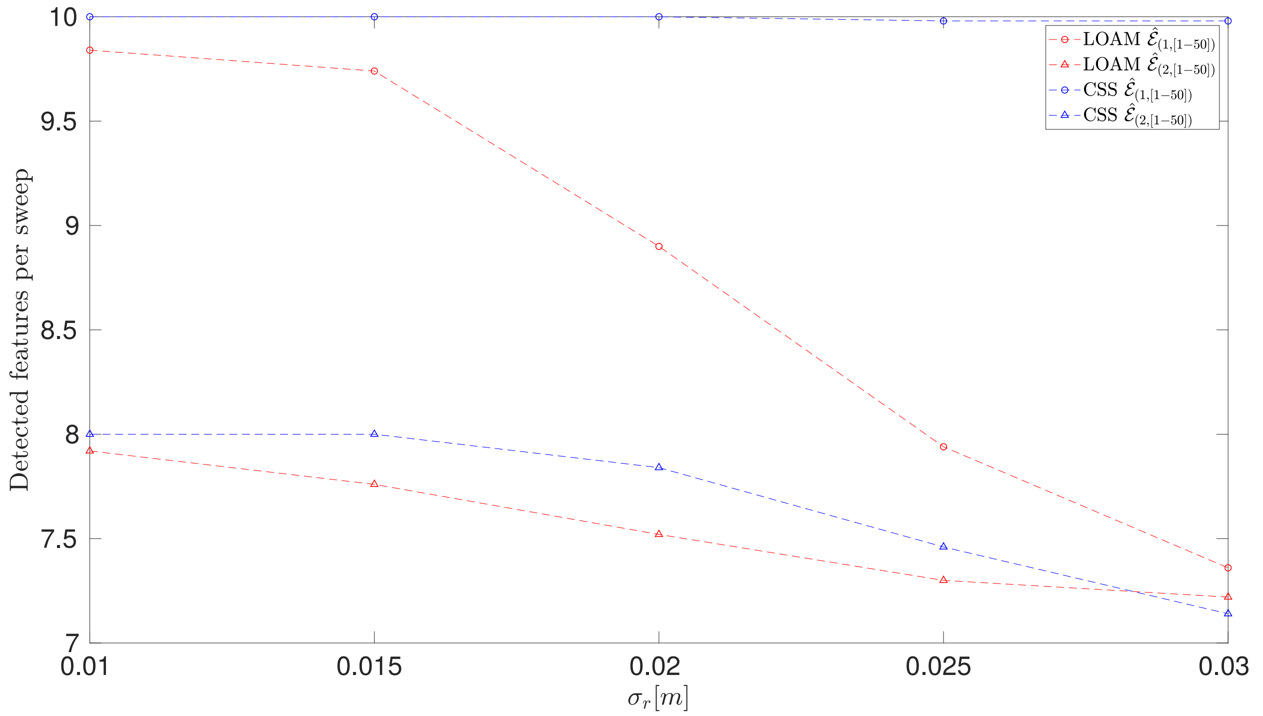


Figure 4.5: Estimated features per sweep extracted with the LOAM's feature extractor and the CSS feature extractor in 50 sweeps.

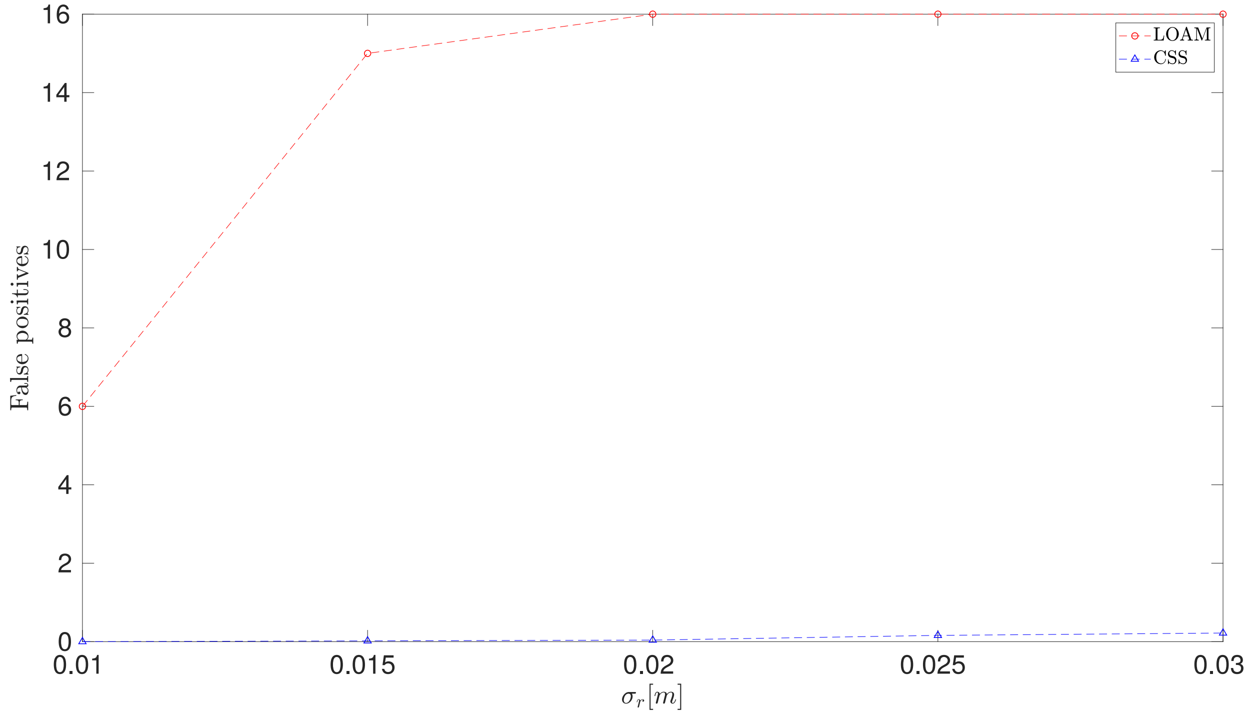


Figure 4.6: False positives features per sweep extracted with the LOAM's feature extractor and the CSS feature extractor in 50 sweeps.

4.3.2 Scenario with robot in motion

The objective of testing the LiDAR mounted on the mobile robot in this scenario was to observe the error between the features detected by the LOAM's feature extractor and the CSS feature extractor with the ground truth features while the robot is in motion. Also, this scenario is useful to observe how the features extracted by the algorithms affect the trajectory estimated by the complete LOAM algorithm.

This scenario is composed of the robot platform mounted with the LiDAR located in $(0\ \ -10\ 0)^T$, which represents a position with Euclidean components x, y, z. Additionally, it was used a set of walls forming 12 corners and boxes of cubic shape forming 28 corners. The layout of this scenario can be seen in figure 4.7 in which the ground truth features are marked with light blue stars.

This experiment consisted on moving the robot around the scenario following the trajectory marked with blue dashed lines. Then, we compared the features extracted with the LOAM's feature extractor and the CSS feature extractor with the features ground truth using the OSPA metric. Finally, the trajectories generated by the original LOAM algorithm and the CSS LOAM algorithm were compared with the ground truth trajectory using the ATE metric.

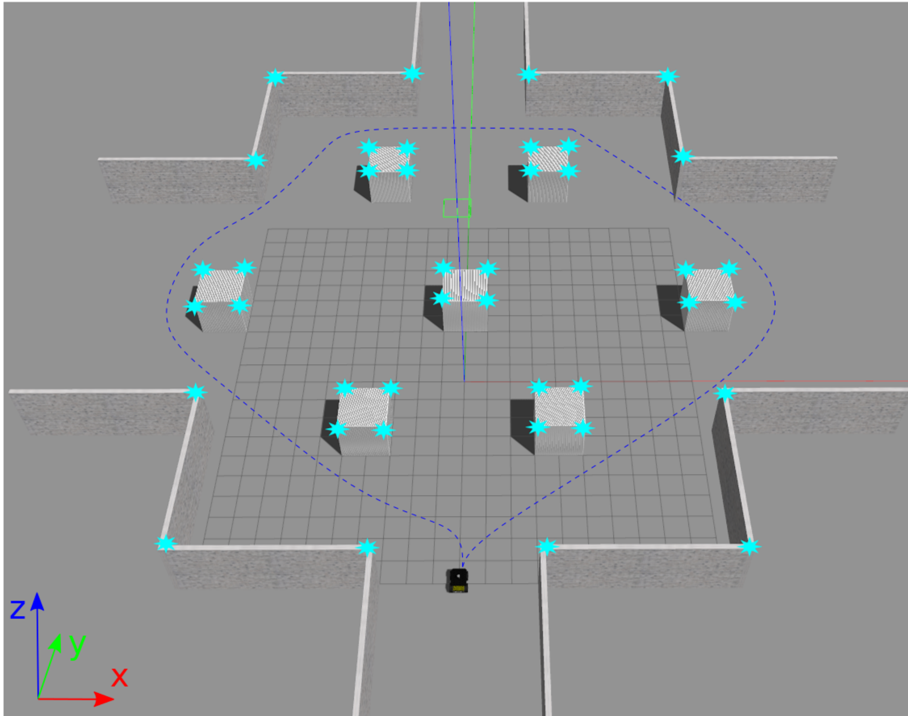


Figure 4.7: Scenario with the robot in motion.

The first metric used was the OSPA metric, which requires two sets. The first set corresponds to the ground truth features and the second set was alternated between the features extracted by the LOAM's feature extractor and the features extracted the CSS feature extractor. This was carried out for three different values of range noise standard deviation σ_r: 0.01 m, 0.02 m and 0.03 m. The parameters c and p, required by OSPA, used in this case

were 5 and 2, respectively.

Figure 4.8 shows the comparison of the OSPA distances between the ground truth features with the features extracted by the LOAM's feature extractor and the CSS feature extractor using $\sigma_r = 0.01$ m. In this figure, the distances obtained with the CSS feature extractor were always lower than those of the LOAM's feature extractor. Both distances are separated by an average value of 0.66 meters and reach a maximum difference of 1.36 meters.

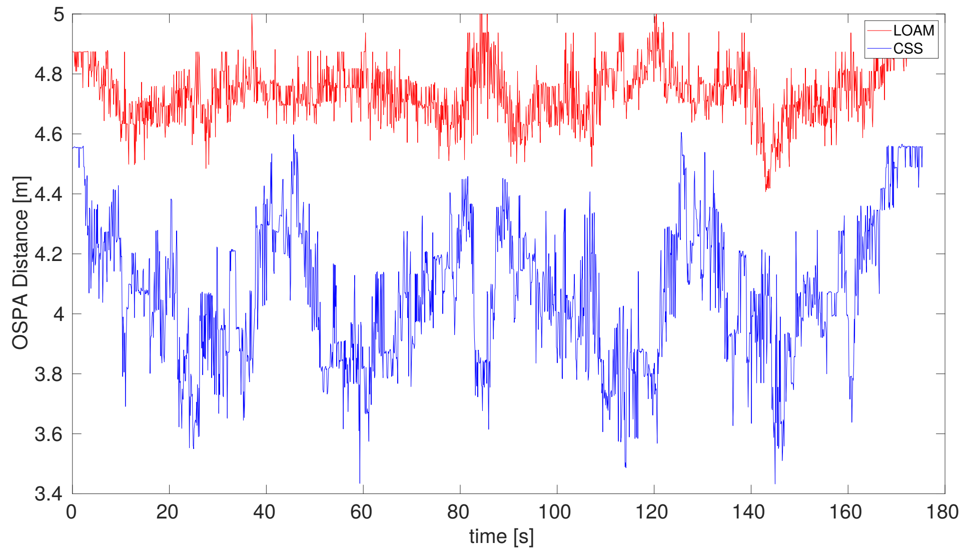


Figure 4.8: Feature estimation error using the OSPA metric ($\sigma_r = 0.01$ m).

Likewise, figure 4.9 shows the comparison of OSPA distances between the ground truth features with the features extracted by the LOAM's feature extractor and the CSS feature extractor using $\sigma_r = 0.02$ m. In this figure, the separation of the distances are not as large as in the previous figure, however, they are separated by an average distance of 0.56 meters and reach a maximum difference of 1.62 meters.

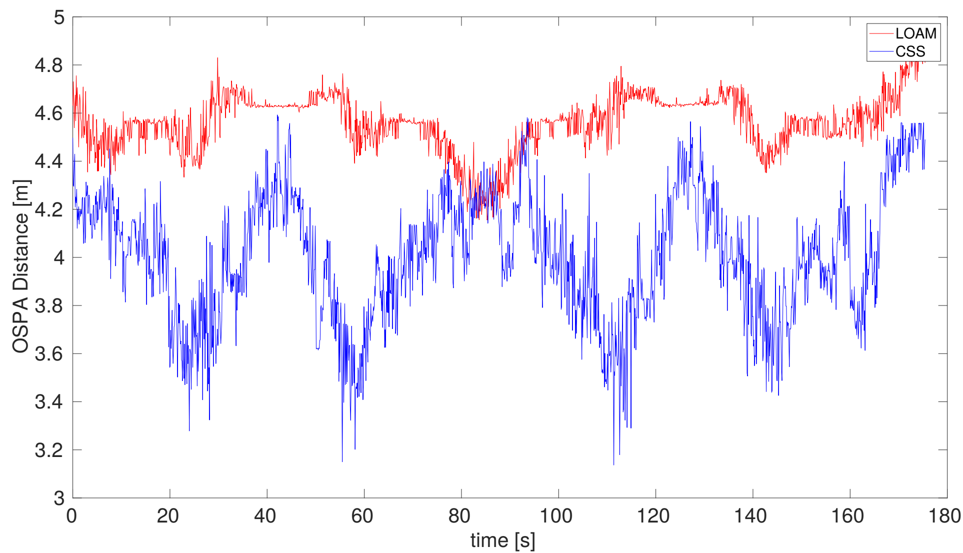


Figure 4.9: Feature estimation error using the OSPA metric ($\sigma_r = 0.02$ m).

Finally, Figure 4.10 shows the comparison of the OSPA distances between the ground truth features with the features extracted by the LOAM's feature extractor and the CSS feature extractor using $\sigma_r = 0.03$ m. In this figure, both distances throughout time are very similar to the previous figure, but in this case they are separated by an average distance of 0.64 meters and reach a maximum difference of 1.65 meters.

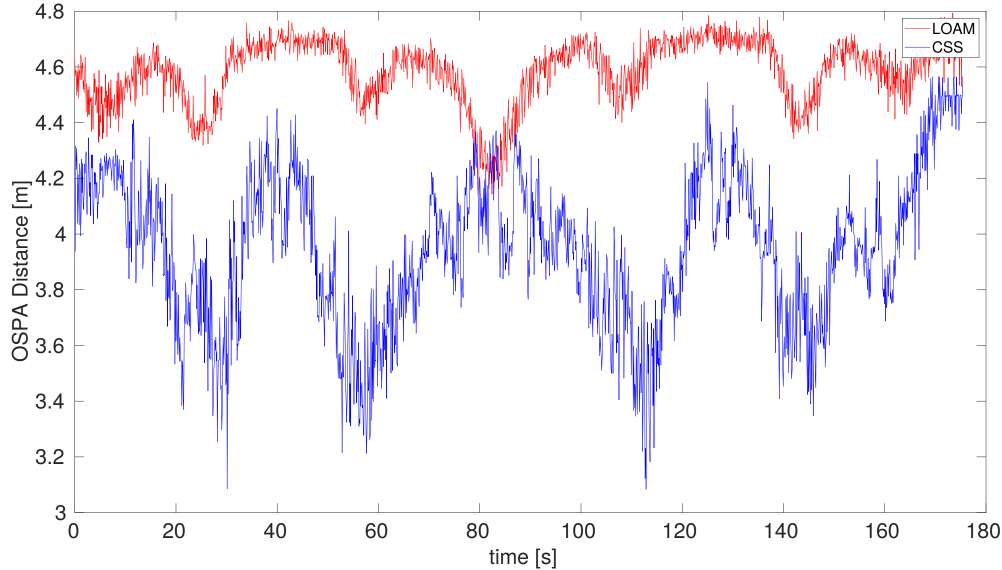
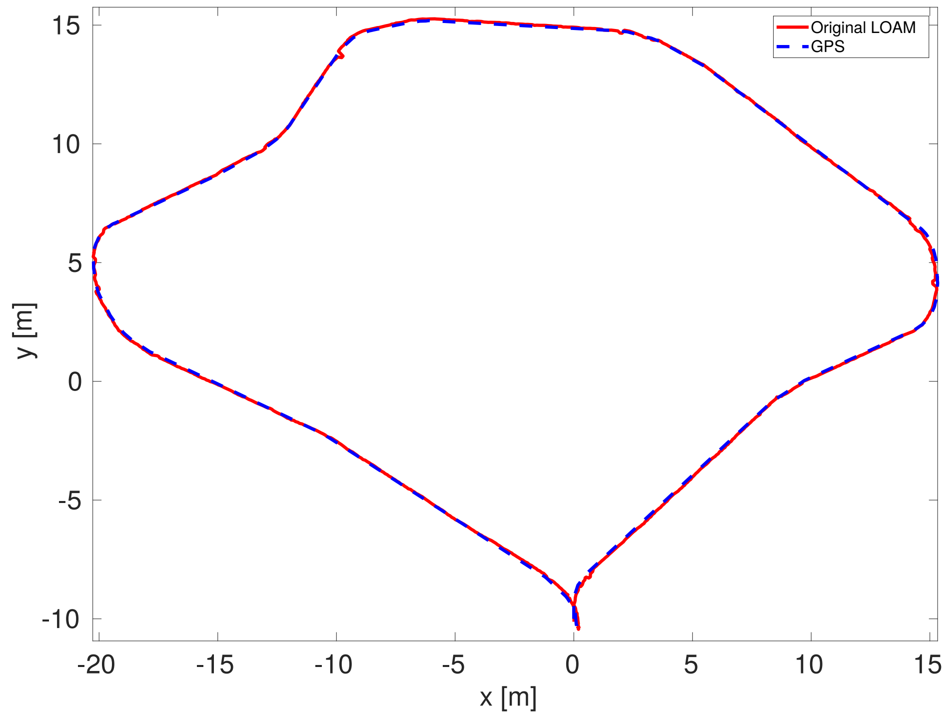


Figure 4.10: Feature estimation error using the OSPA metric ($\sigma_r = 0.03$ m).

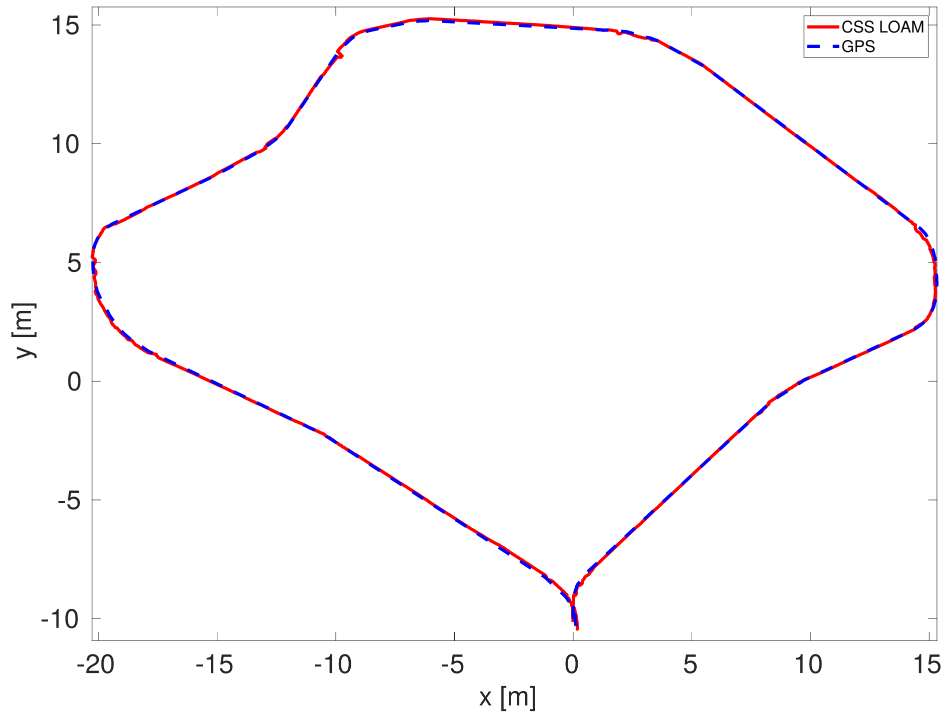
The second metric used in this scenario was the ATE metric. As occurred in the previous experiment, this scenario was also evaluated under 3 different range noise standard deviation (σ_r) values: 0.01 m, 0.02 m and 0.03 m. For each, 2 graphs were plotted:

1. The trajectories estimated with the original LOAM algorithm and the CSS LOAM algorithm compared to the trajectory obtained with GPS.
2. The errors in x and y obtained comparing the original LOAM algorithm and the CSS LOAM algorithm with the GPS throughout time.

The following two figures show the items described above for $\sigma_r = 0.01$ m. The trajectories generated by the original LOAM algorithm and the CSS LOAM algorithm compared with the trajectory obtained with GPS are shown in Figures 4.11a and 4.11b. Both trajectories look very similar. This lack of difference in both algorithms occurs because of the low level of noise added to the sensor data.



(a) Estimated trajectory using the Original LOAM algorithm.

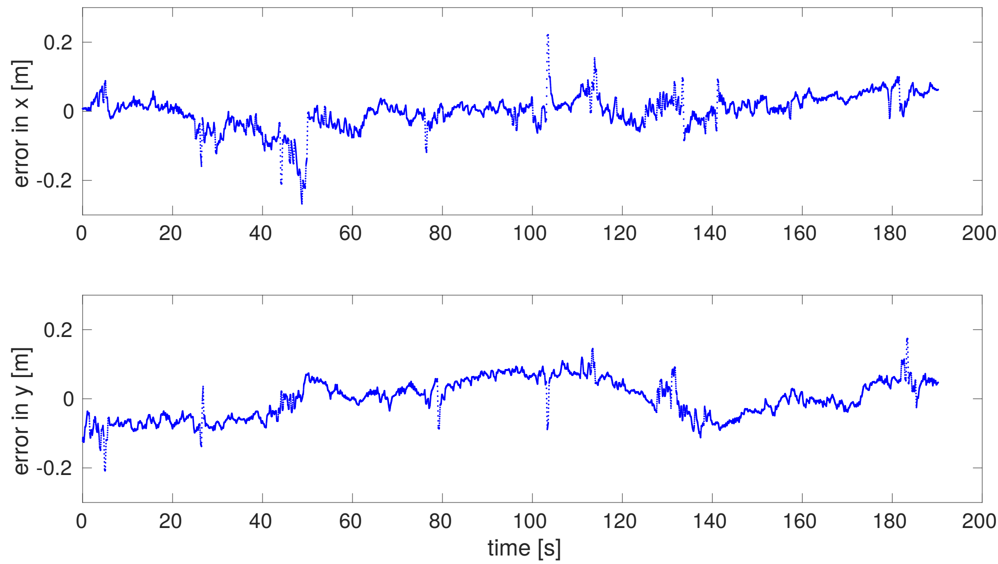


(b) Estimated trajectory using the CSS LOAM algorithm.

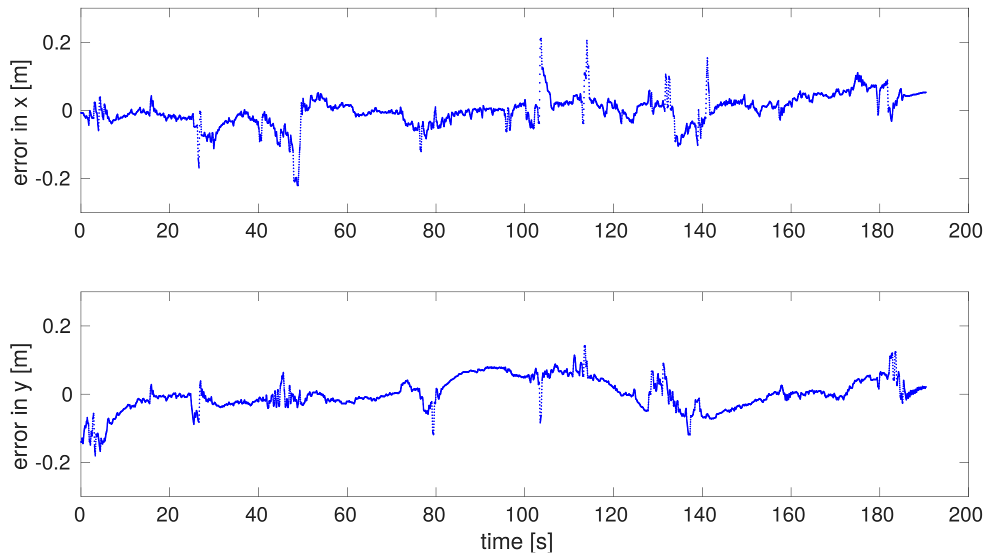
Figure 4.11: Estimated trajectories compared to ground-truth using simulated data (σ_r = 0.01 m).

Figures 4.12a and 4.12b shows the errors in x and y of the original LOAM algorithm and the CSS LOAM algorithm throughout time. In these graphs, errors are very similar, except

that the x error value at time 50 s of the LOAM features's graph (4.12a) almost reaches the value -0.3 m.



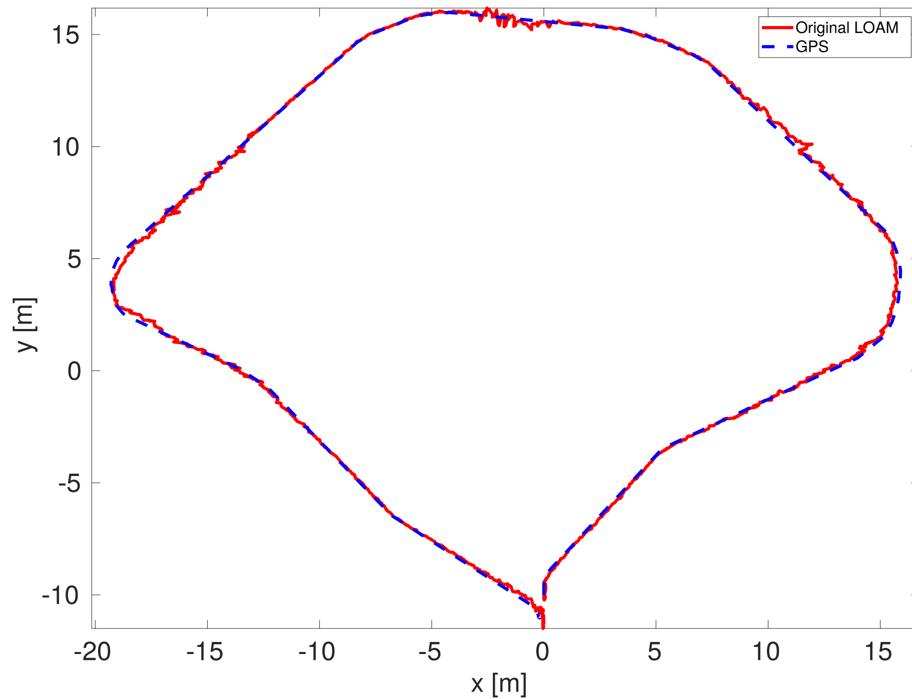
(a) Errors in x, y produced by the Original LOAM algorithm.



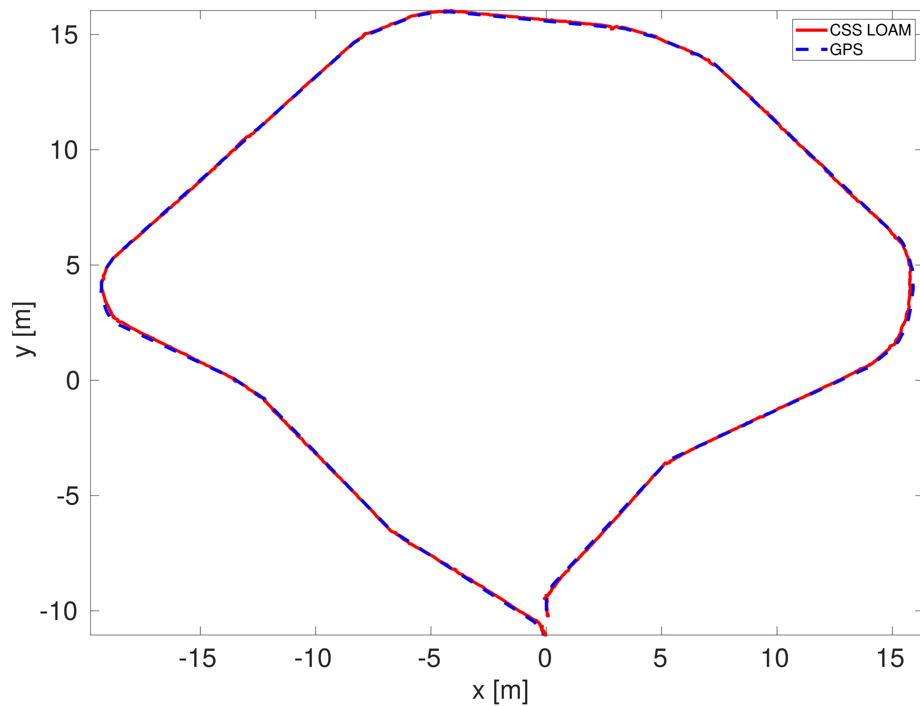
(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.12: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using simulated data ($\sigma_r = 0.01$ m).

In the following two figures the data was evaluated for $\sigma_r = 0.02$ m. Therefore, the trajectories generated by the original LOAM algorithm and the CSS LOAM algorithm compared with the trajectory obtained with GPS are shown in Figures 4.13a and 4.13b. In these figures, the estimated trajectory with the original LOAM algorithm presents high error values in the area near to the point $x = 0$, $y = 15$, approximately. This trajectory error is also represented in the error in y at time 100 s shown in Figure 4.14a.



(a) Estimated trajectory using the original LOAM algorithm.

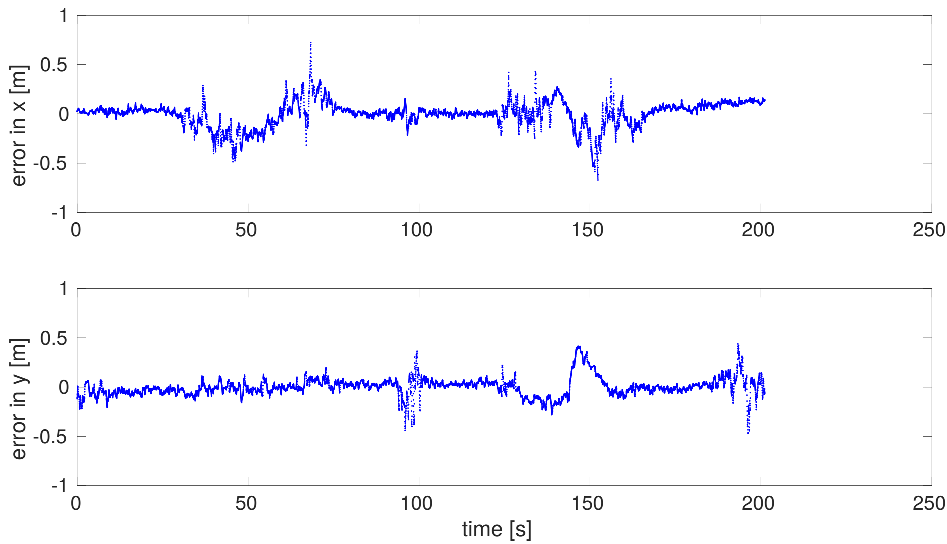


(b) Estimated trajectory using the CSS LOAM algorithm.

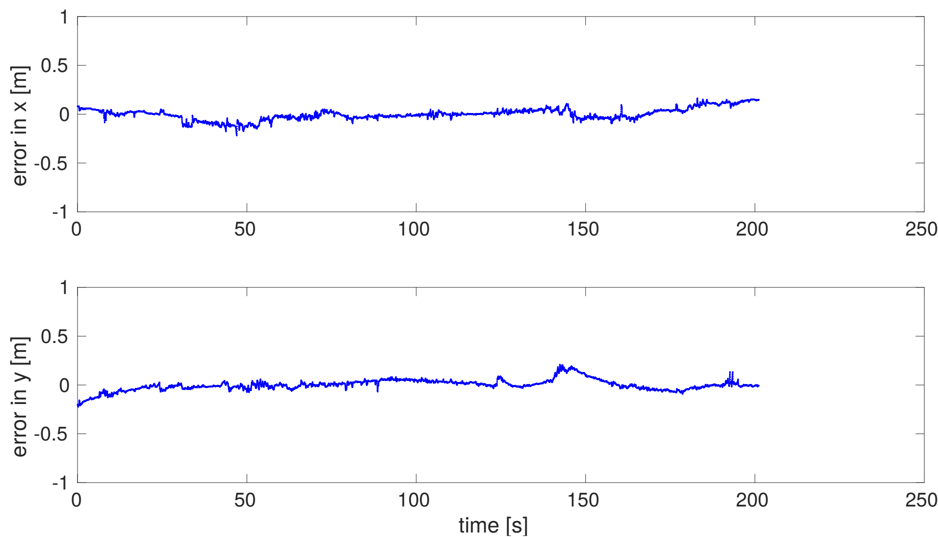
Figure 4.13: Estimated trajectories compared to ground-truth using simulated data ($\sigma_r = 0.02$ m).

Figure 4.14a and 4.14b shows the errors in x and y of the original LOAM algorithm and the CSS LOAM algorithm throughout time. Compared to their pairs using a range noise with $\sigma_r = 0.01$ m, these figures present a more noticeable difference. In the case of the original

LOAM algorithm, some values almost reach -1 and 1, while in the case of the CSS LOAM algorithm, the values vary from -0.3 m to 0.2 m.



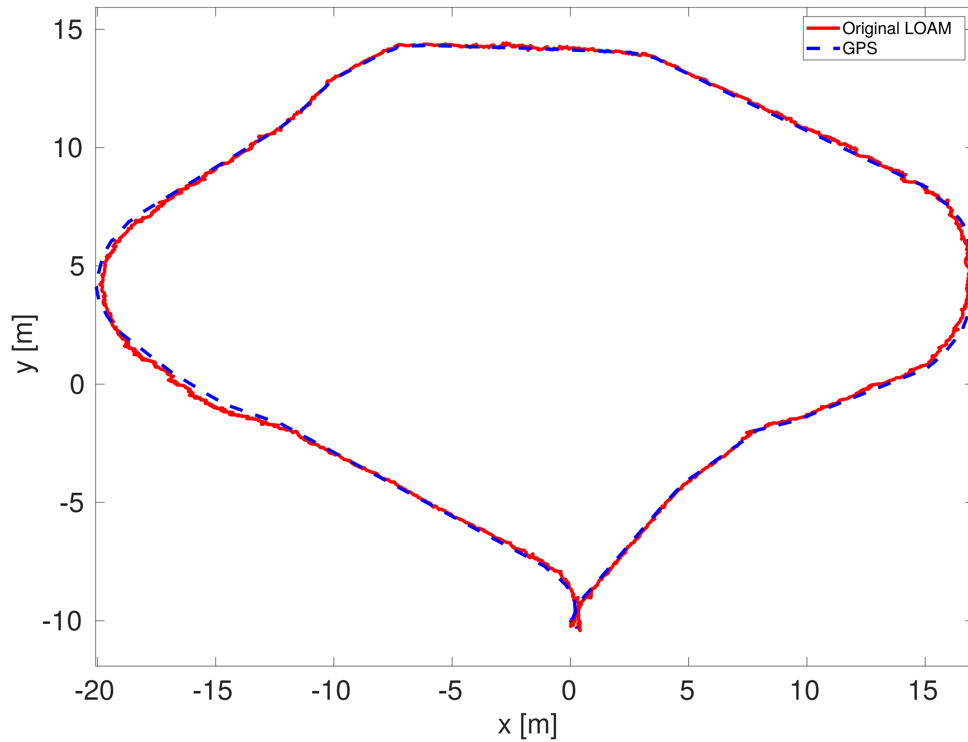
(a) Errors in x, y produced by the original LOAM algorithm.



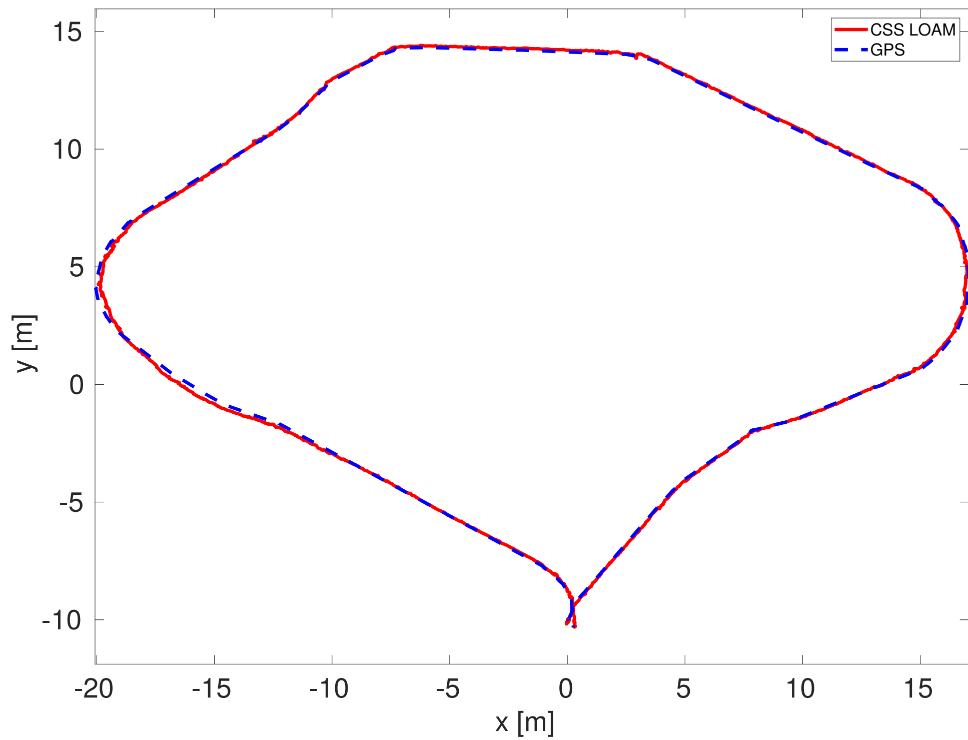
(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.14: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using simulated data ($\sigma_r = 0.02$ m).

The last range noise standard deviation tested was $\sigma_r = 0.03$ m. In this case, the graphs are similar to the previously evaluated case. The trajectories generated by the original LOAM algorithm and the CSS LOAM algorithm compared with the trajectory obtained with GPS are shown in Figures 4.15a and 4.15b. In these figures, errors are present in both trajectories, specially in the area near to the point $x = -15$, $y = 0$, approximately. However, the estimated trajectory produced by the original LOAM algorithm presents a higher error at this point as shown in the error in x at time 160 s of Figure 4.16a.



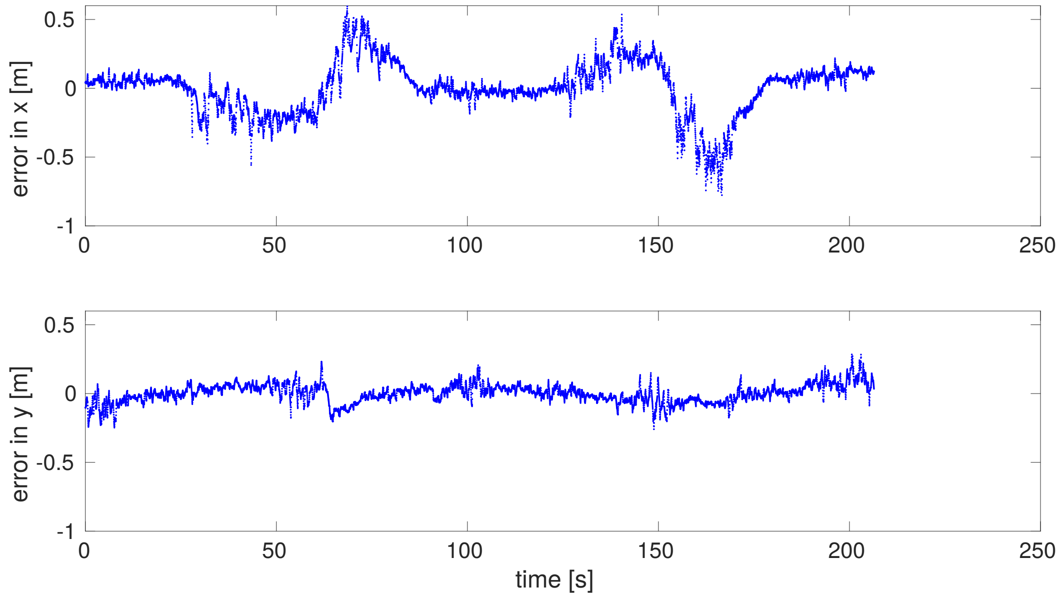
(a) Estimated trajectory using the original LOAM algorithm.



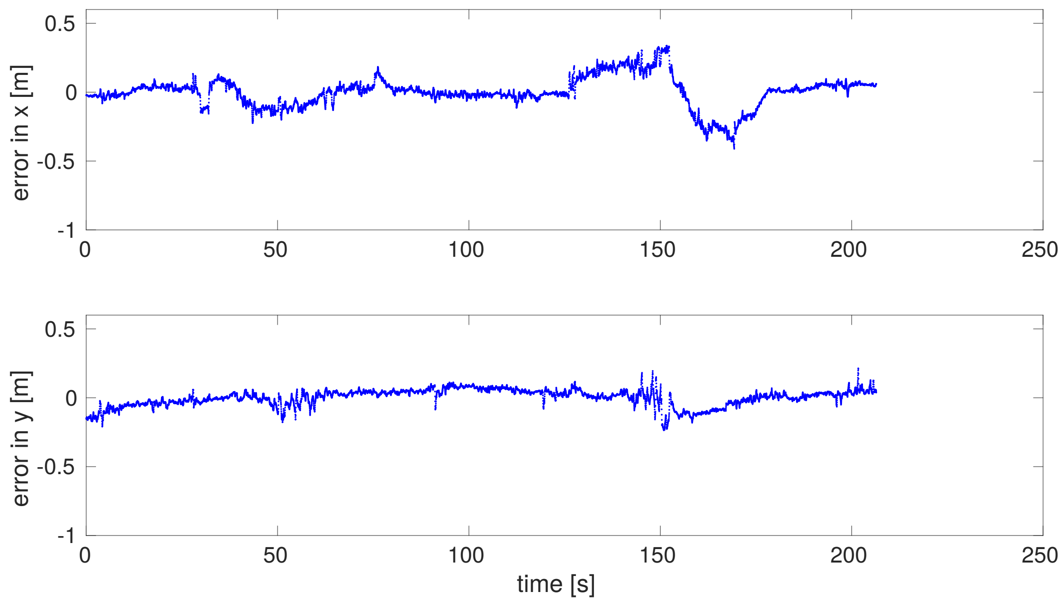
(b) Estimated trajectory using the CSS LOAM algorithm.

Figure 4.15: Estimated trajectories compared to ground-truth using simulated data (σ_r = 0.03 m).

Figures 4.16a and 4.16b show the errors in x and y of the original LOAM algorithm and the CSS LOAM algorithm throughout time. These graphs present noticeable errors as occurred previously. The errors when using the original LOAM algorithm vary from -0.8 m to 0.6 m, while in case of the CSS LOAM algorithm, they vary from -0.3 m to 0.3 m.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.16: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using simulated data ($\sigma = 0.03$ m).

To summarize all the figures shown in this sub-section, figure 4.17 shows the error percentages, obtained with the ATE metric when using the original LOAM algorithm and the CSS LOAM algorithm per each level of noise (σ_r) of the scenario with the robot in motion. For the three range noise standard deviation values evaluated in this scenario, the use of the CSS LOAM algorithm generated error percentage values less than when using the original LOAM algorithm. From this information we can conclude that the range noise of the sensor can significantly affect the trajectory estimated by the original LOAM algorithm.

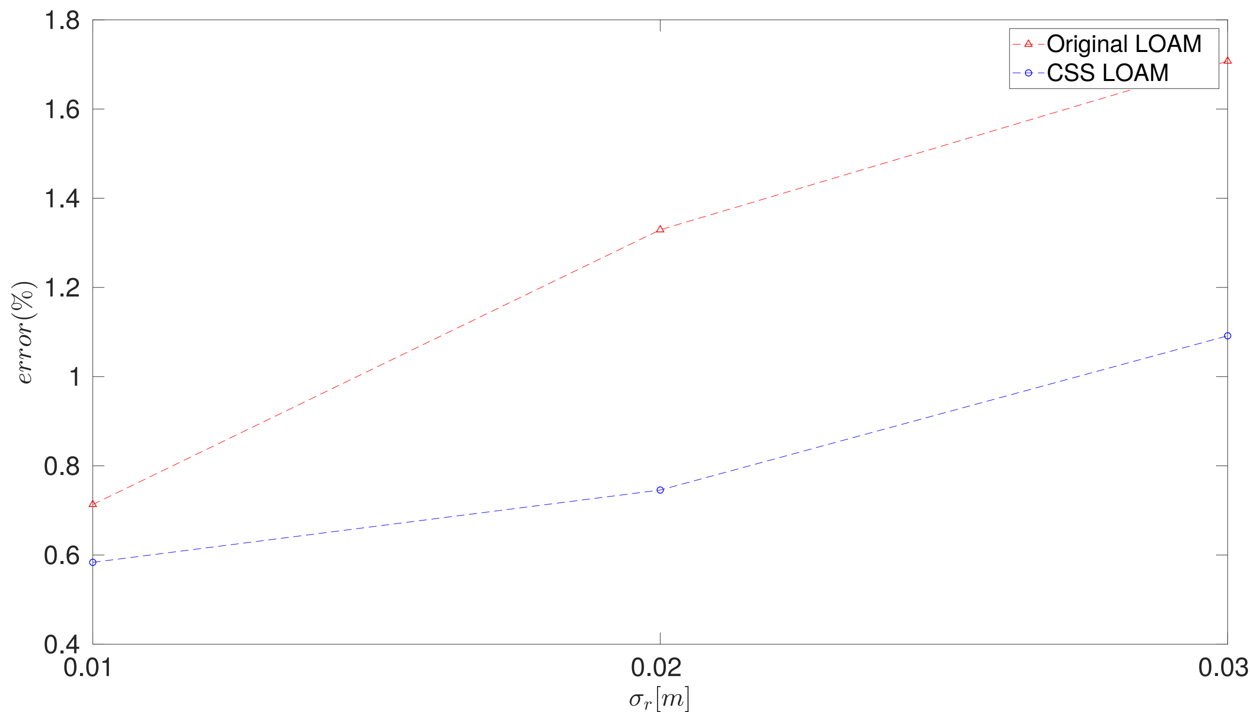


Figure 4.17: Error percentages per each range noise standard deviation value.

4.4 Real Data

The purpose of the use of real data is to try to reproduce the results obtained with the simulated data. As mentioned in the previous chapter, the robotic platform for collecting the experimental real dataset was a Clearpath Husky A-200 robot equipped with a Velodyne VLP-16 LiDAR and a GPS Topcon Hyper V.

The environment used to conduct the experiments was the surroundings of the Universidad de Chile - Campus Beauchef, located in Santiago de Chile's downtown area. This scenario is mainly composed of buildings and trees which can feed the LOAM and CSS feature extraction algorithms with a variety of shapes.

Figure 4.18 shows a satellite image of the environment with the ground truth marked with a dashed blue line. Sensors started to record the data from the bottom-left side of this image (marked with a green square) and ended after passing trough the circular area in the bottom for second time (marked with a red circle).

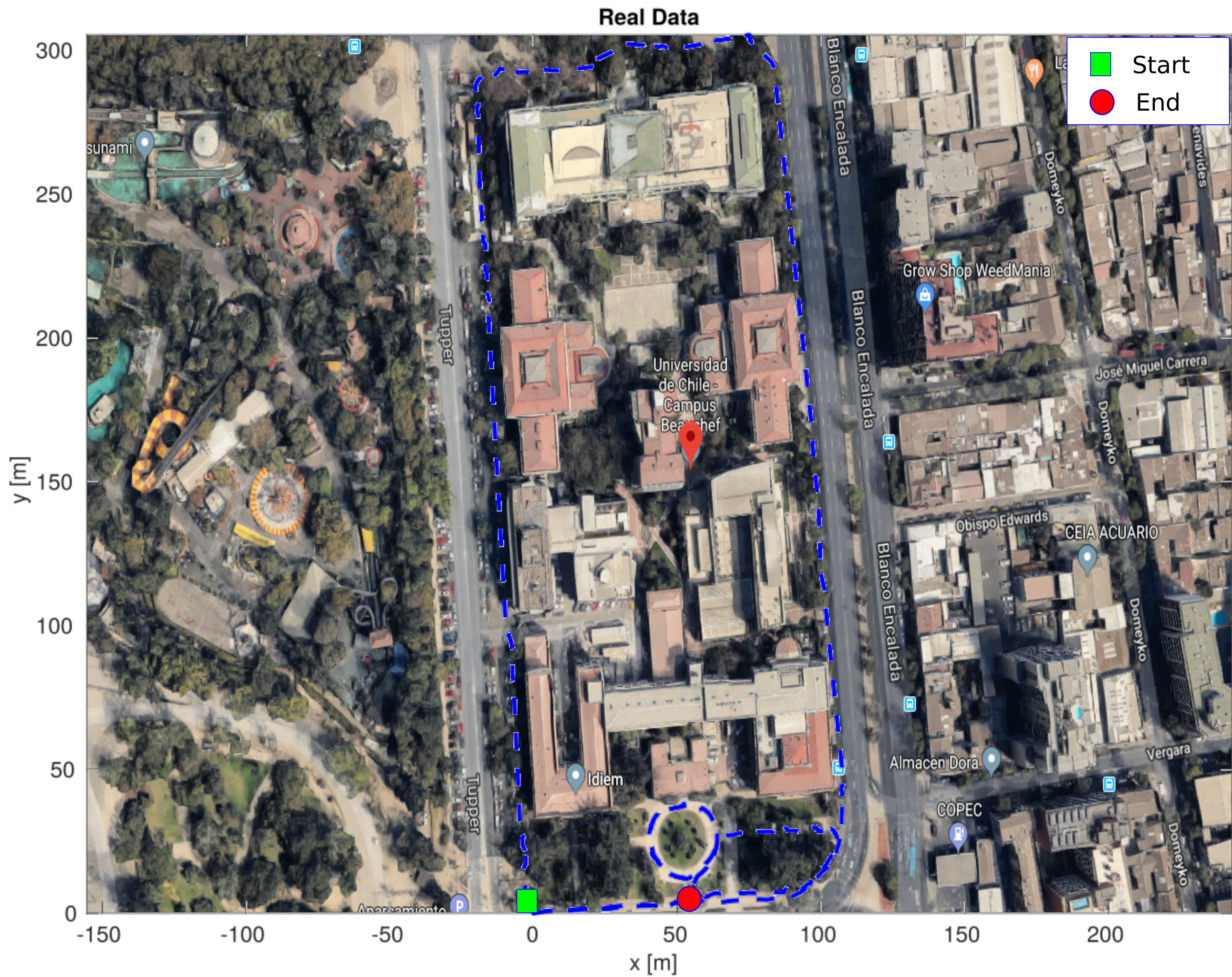
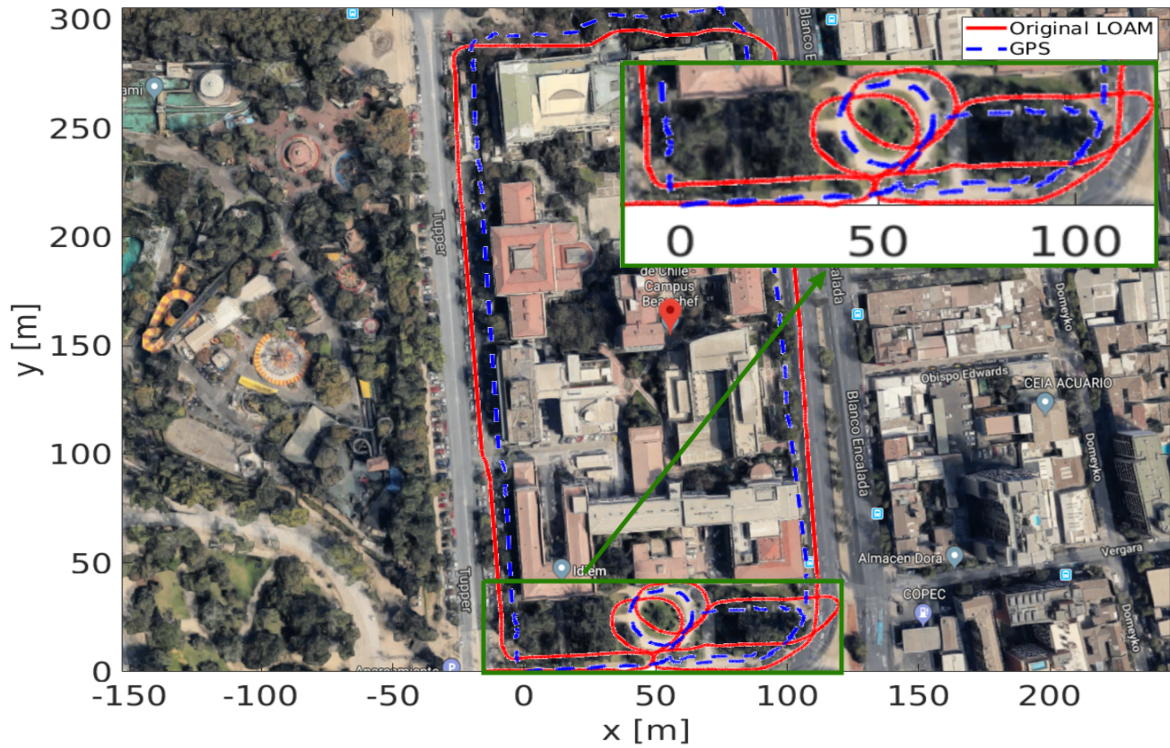
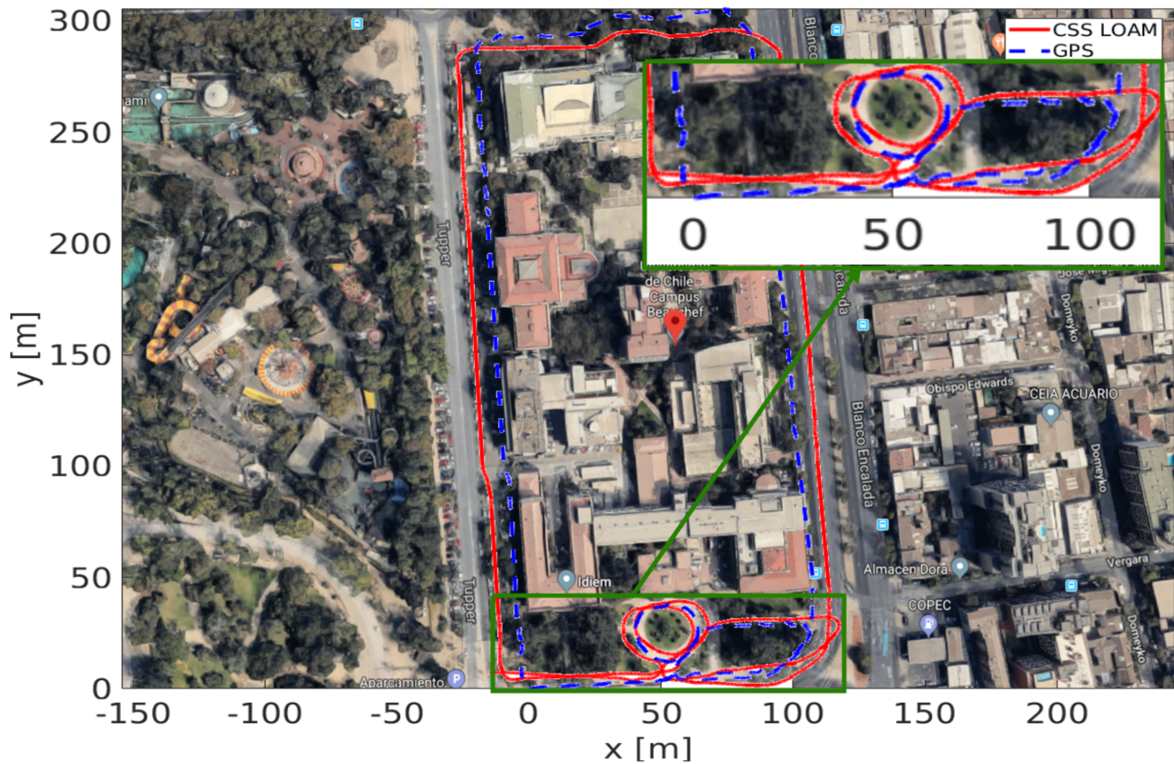


Figure 4.18: Satellite image of Universidad de Chile - campus Beauchef.

To finish these experiments, the complete trajectory of the original LOAM algorithm (red line) and the CSS LOAM algorithm (red line) are compared with the trajectory generated by the GPS (blue dashed line) as shown in figures 4.19a and 4.19b. In these figures, the estimated trajectories of the circular area located at the bottom is shown separately as a zoomed view. This is to show that the trajectory of the CSS LOAM algorithm is aligned better with the blue line than the trajectory of the original LOAM algorithm.



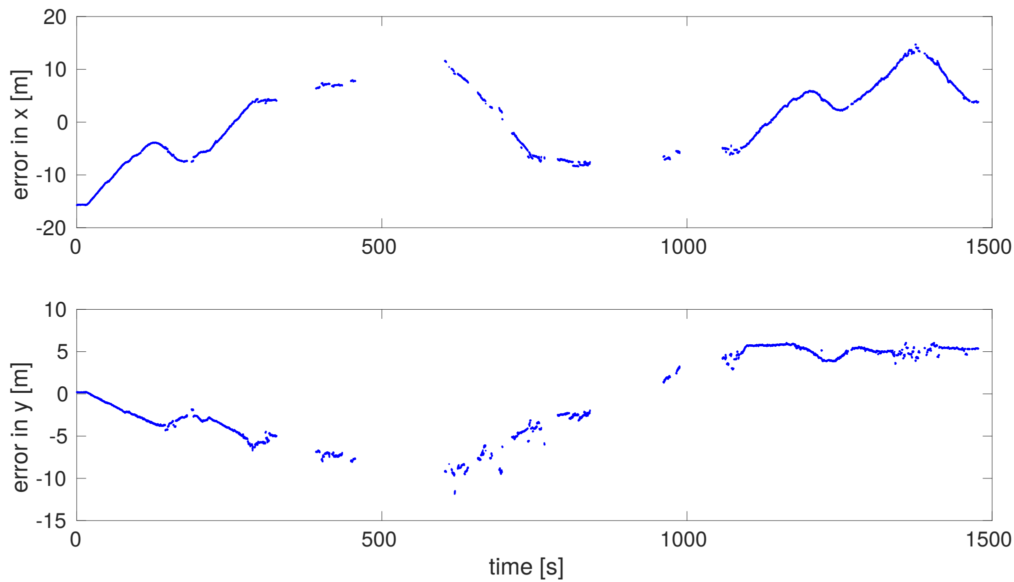
(a) Estimated trajectory using the original LOAM algorithm.



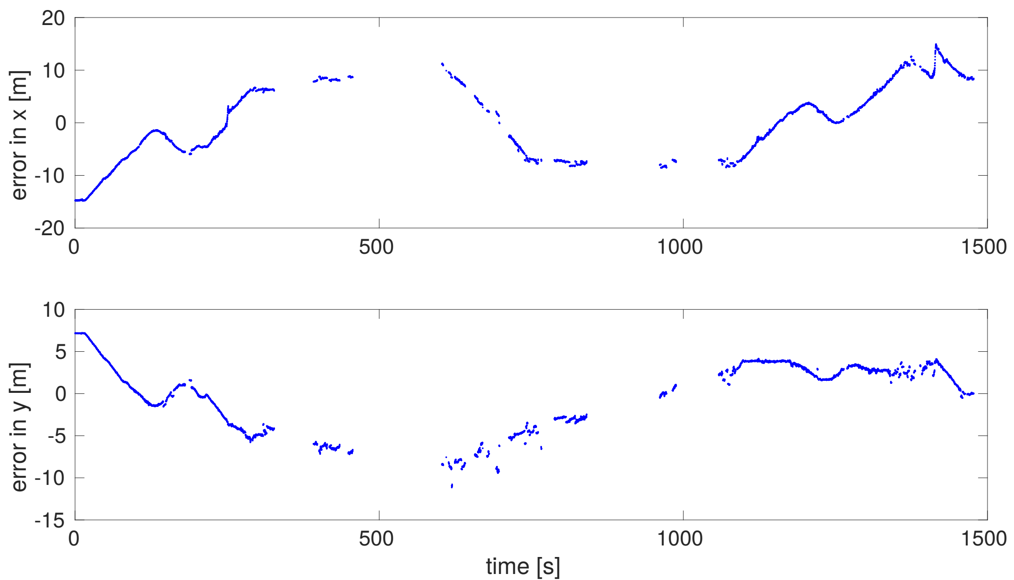
(b) Estimated trajectory using the CSS LOAM algorithm.

Figure 4.19: Estimated trajectories compared to ground-truth using real data.

Figures 4.20a and 4.20b show the errors, calculated with the ATE metric in the x and y coordinates, of the trajectory estimated by the original LOAM algorithm and the CSS LOAM algorithm. These errors correspond to the estimated trajectories shown in figure 4.19. Unlike their pairs in the simulated data, these graphs are not continuous, because the dataset from the GPS was larger than the data from the algorithms under test. Hence, each robot pose generated with the original LOAM algorithm and the CSS LOAM algorithm had to be matched to its correspondent pose obtained from the GPS according to the time stamp. These graphs look very similar except for the last few seconds in the y error corresponding to the CSS LOAM algorithm, which is smaller than its value in the original LOAM algorithm. These last seconds correspond to the trajectory zoomed in Figure 4.19.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.20: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm using real data.

A general estimate of previous translation errors obtained with the ATE metric can be determined from the mean of these errors $\bar{d}_j$. Therefore, the mean of the translation errors corresponding to the original LOAM algorithm was 8.16 meters, while in the case of the CSS LOAM algorithm was 7.25 meters. This means that in average the CSS LOAM algorithm obtained a lower error rate than the original LOAM algorithm with a difference of 0.91 meters.

4.5 KITTI Dataset

Since the LOAM algorithm is the highest ranked algorithm in the KITTI Visual Odometry Benchmark Dataset, we decided to use the same dataset to test the original LOAM algorithm and the CSS LOAM algorithm. As mentioned in the beginning of this work, it is important to the note that the version of LOAM used in these experiments was the one written by the original author, J. Zhang, in 2013 and which is available in the ROS Indigo website.

To run the experiments, we used the odometry dataset composed of a set of .bin files per sequence (from sequence 00 to sequence 10). Additionally, this dataset has the ground truth available for the sequences mentioned before. All these files were converted to point cloud data using the PCL library along with ROS. Finally, the sampling frequency of the real Velodyne HDL-64E LiDAR, which is 10 Hz, was also simulated. It is important to note that the KITTI Visual Odometry Benchmark does not require the simulation of the sampling frequency of the LiDAR. However, we decided to do it to see if the LOAM algorithm is fast enough to process all the data in the real LiDAR's sampling frequency and if it is capable of running correctly in the computer used in these experiments.

The experiments consisted of running the original LOAM algorithm and the CSS LOAM algorithm individually. Then, we compare the trajectory generated by each one with the ground truth using the ATE metric. The use of this metric permitted us to obtain the whole trajectory produced by each algorithm compared to the ground truth and also the errors produced by each algorithm in the Euclidean components x and y throughout time.

Finally, we decided to show the number of features extracted by each algorithm throughout time. It is important to remember that in this work the features replaced in the LOAM's feature extractor were only the sharp features. In the original implementation, these features were divided in two: sharp and less sharp. Hence, this classification was also used to plot the number of features vs time graph.

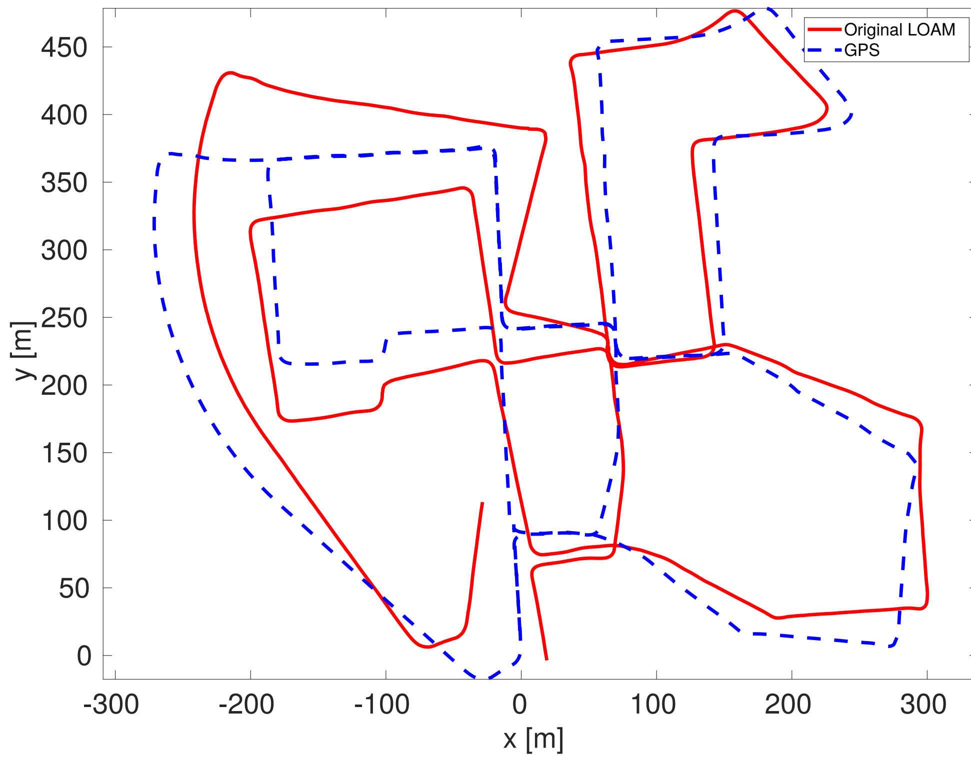
The following figures to be shown are the most representative sequences of the experiments with carried out with the KITTI dataset. The complete experiment with all the sequences can be found in Appendix A. Therefore, figures are shown in 3 parts. First, the trajectory generated with the original LOAM algorithm and the CSS LOAM algorithm compared with the ground truth is shown in figures 4.21, 4.24, 4.27 and 4.30, where the ground truth is represented with a blue dashed line and the trajectory estimated by the algorithms is represented with a red line. Second, the errors in Euclidean components x and y produced by each algorithm throughout time are shown in figures 4.22, 4.25, 4.28 and 4.31. Finally,

the number of features extracted by each algorithm in time are shown in Figures 4.23, 4.26, 4.29 and 4.32. In these last figures, the graphs corresponding to the number of LOAM sharp, LOAM less sharp, CSS sharp and CSS less sharp features are represented with colors blue, orange, green and red respectively.

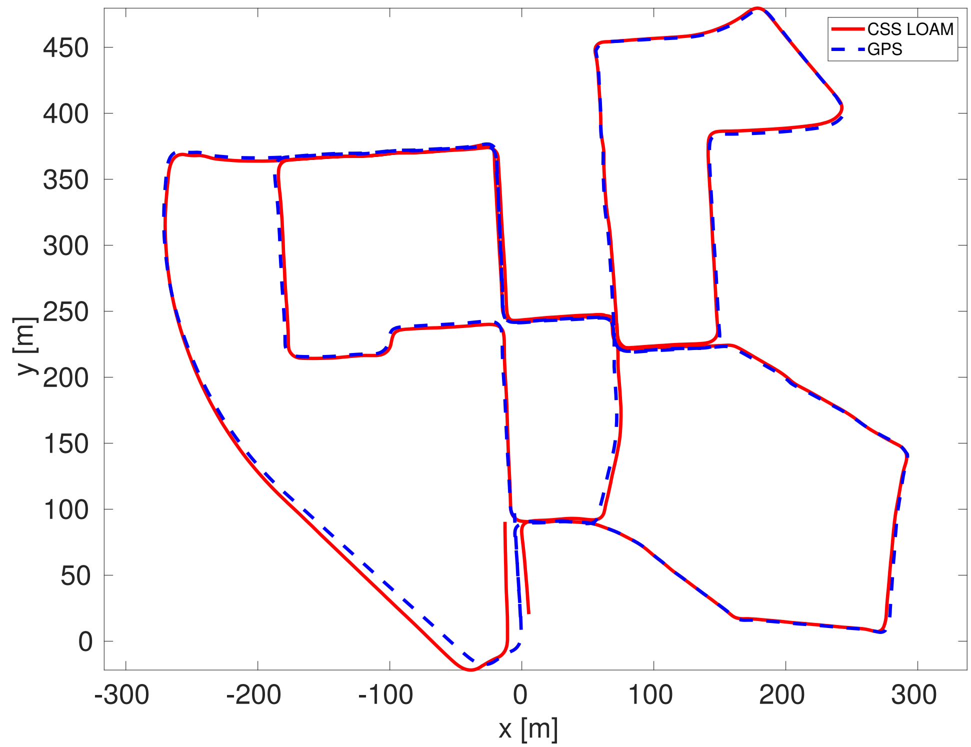
The sequences where the CSS LOAM algorithm obtained an evidently smaller error than original LOAM algorithm were the sequences 00, 05 and 06 which can be seen in figures 4.21, 4.24, 4.27, respectively. These sequences have in common that the number of CSS sharp and CSS less sharp features are almost always close to $1,000$ features as can be seen in figures 4.23, 4.26, 4.29. On the other hand, the sequence where the original LOAM algorithm obtained an evidently smaller error than the CSS LOAM algorithm was the sequence 09, which can be seen in figure 4.30. Unlike the sequences where the CSS LOAM algorithm performed well, the number of features extracted for this sequence through time was far from $1,000$ features. When analyzing this sequence, we noted that some parts of this sequence lacked structures like walls, corners, lamp posts or trees. This made the CSS feature extractor had problems finding features, which resulted in few features extracted. It is important to remember that to perform the feature matching task, the number of features extracted plays a fundamental role. For the rest of sequences the trajectories generated by both algorithms were very similar.

A question that remains after running the experiments with the KITTI Visual Odometry Benchmark Dataset is why the original LOAM algorithm obtained a high error rate for most of the sequences if it is the highest ranked algorithm in this benchmark. The answer to this question can be related to the sampling frequency of the LiDAR used in these experiments. As said before, for these experiments the HDL-64E LiDAR was configured to return 10 $\mathcal{P}_k$ ($k = \{1, 2, ..., 10\}$) per second. This meant that the LOAM algorithms had 0.1 seconds to perform the feature matching process before processing a new point cloud. Although the original LOAM's feature extractor is less computationally expensive than the CSS feature extractor, the LOAM's feature extractor returns more features than the CSS feature extractor. This characteristic makes the feature matching task more computationally expensive.

Finally, to see if the previous issue was affecting the original LOAM algorithm, we analyzed if the software of this algorithm was processing all the point clouds. What we observed was that the software was *dropping* some point clouds, which means that some point clouds were not being processed. Therefore, the quantity of point clouds dropped on each sequence is depicted in figure 4.33. This issue could be solved if we decrease the sampling frequency of the LiDAR or if we use computer with more resources, however, in this work the CSS LOAM algorithm resulted as an option to tackle this issue.

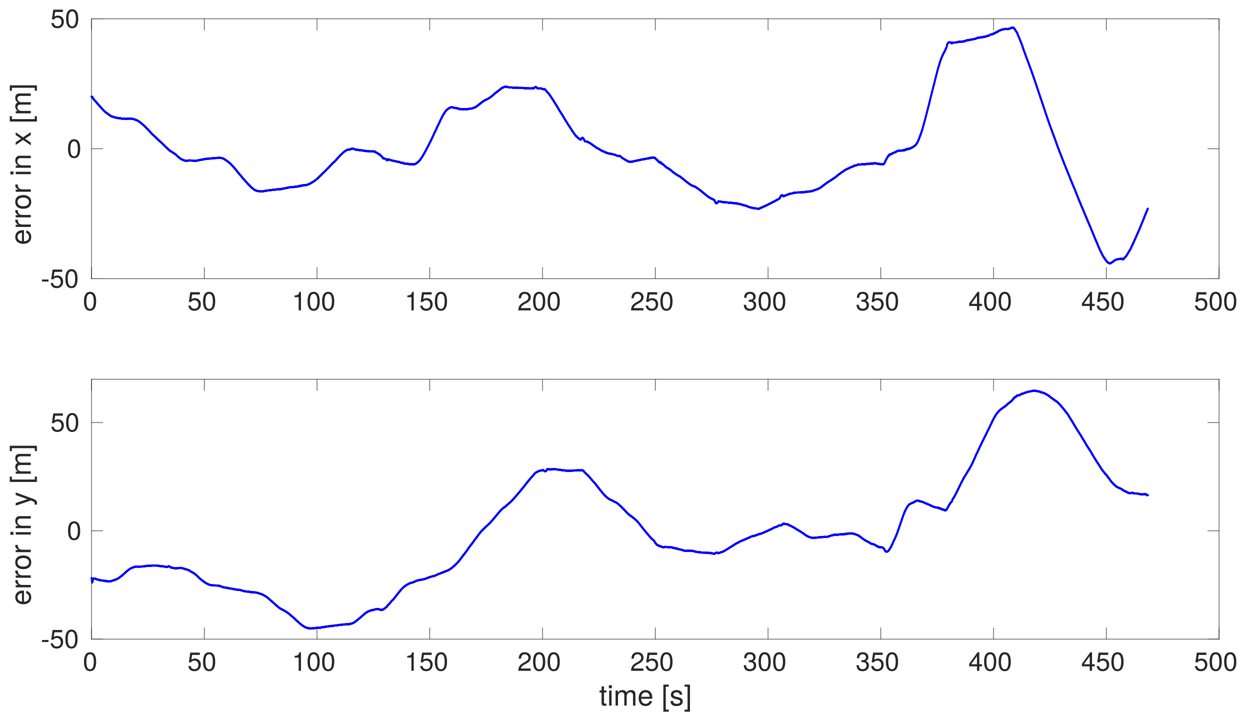


(a) Estimated trajectory using the original LOAM algorithm.

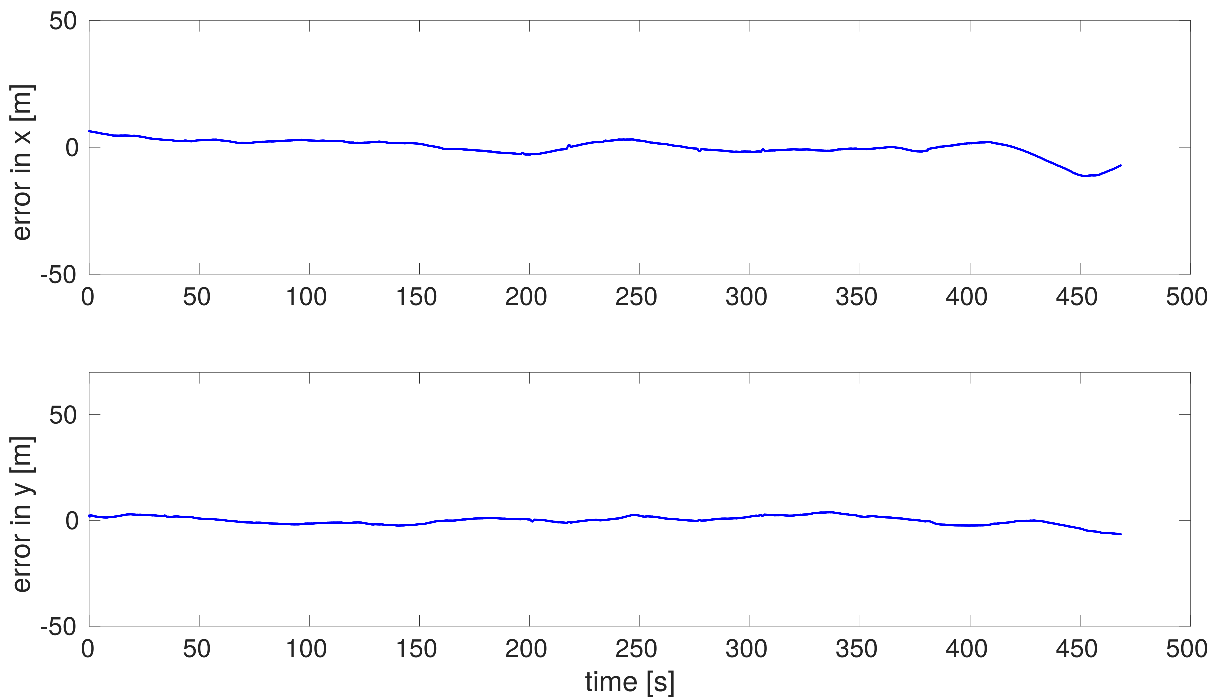


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure 4.21: Estimated trajectories compared to ground-truth in the sequence 00.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.22: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 00.

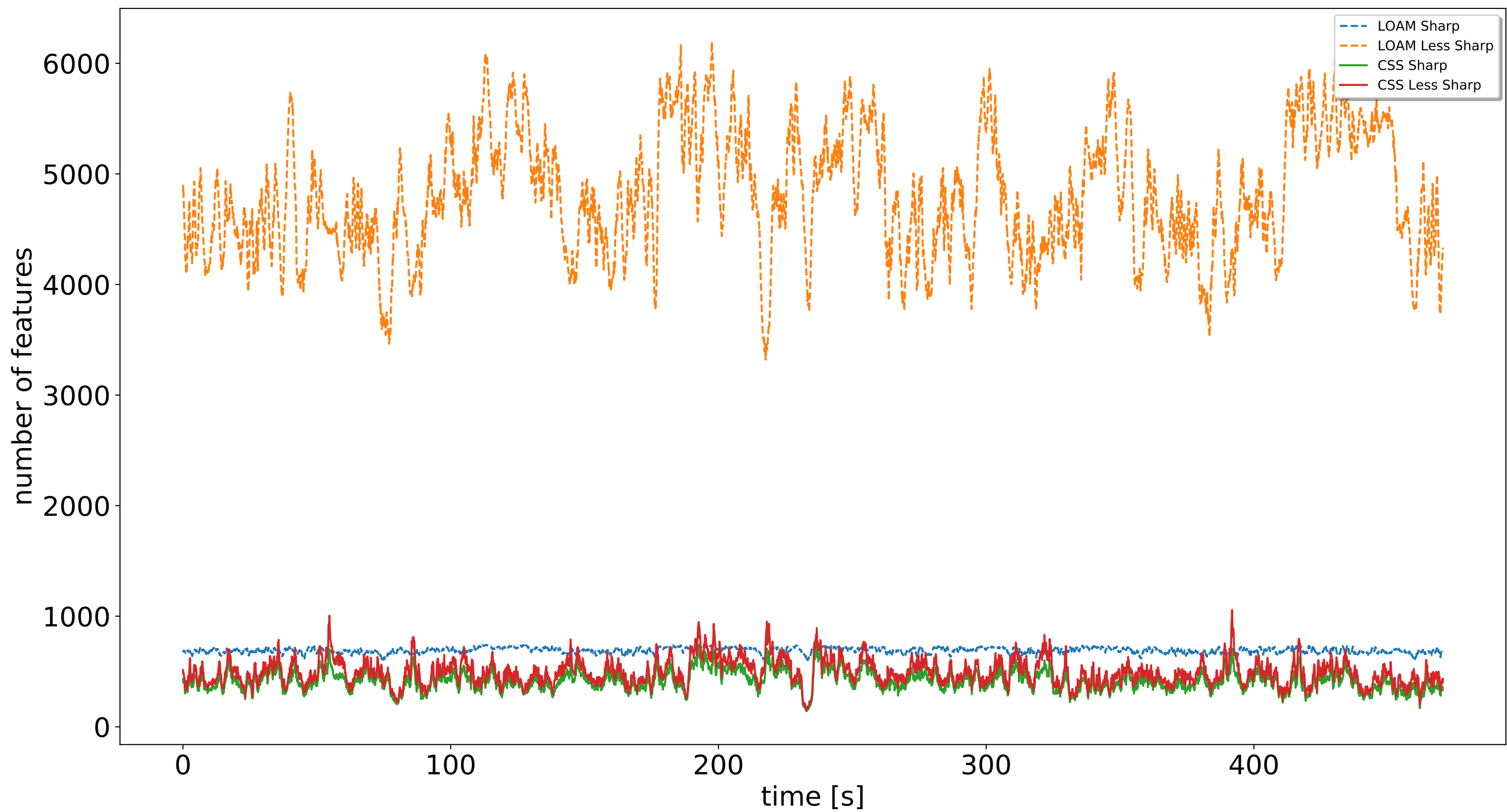
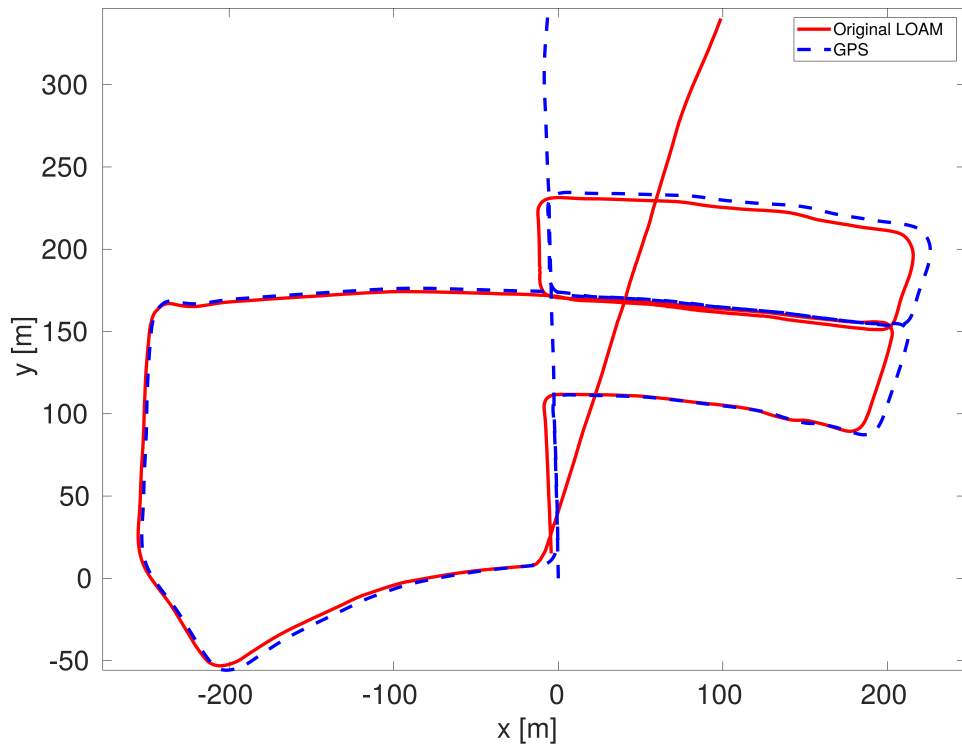
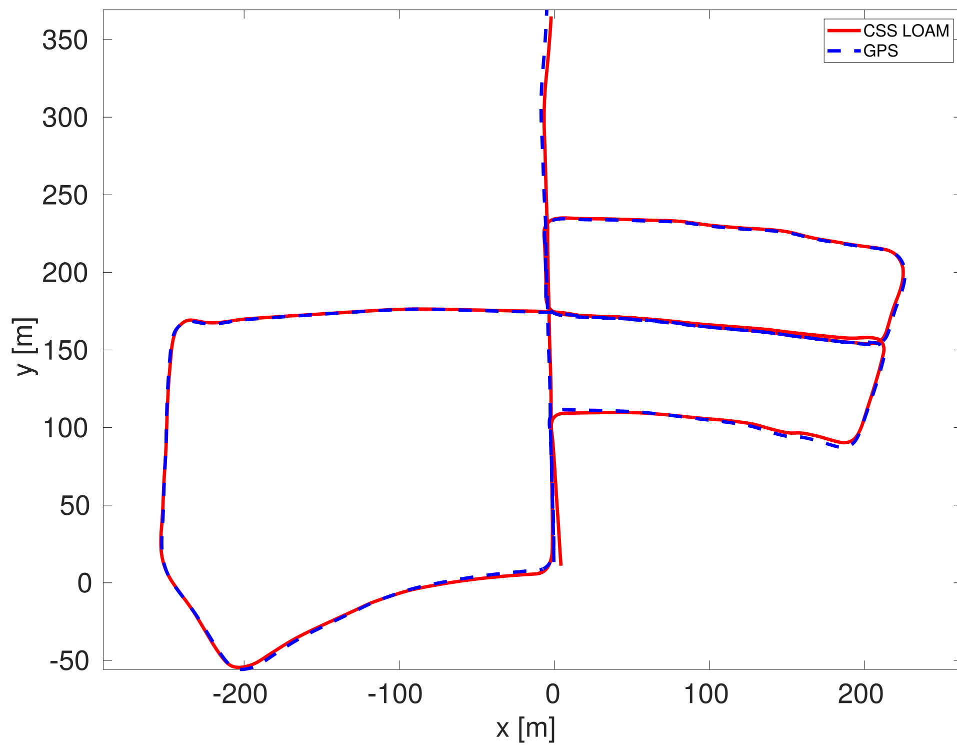


Figure 4.23: Number of features in the sequence 00.

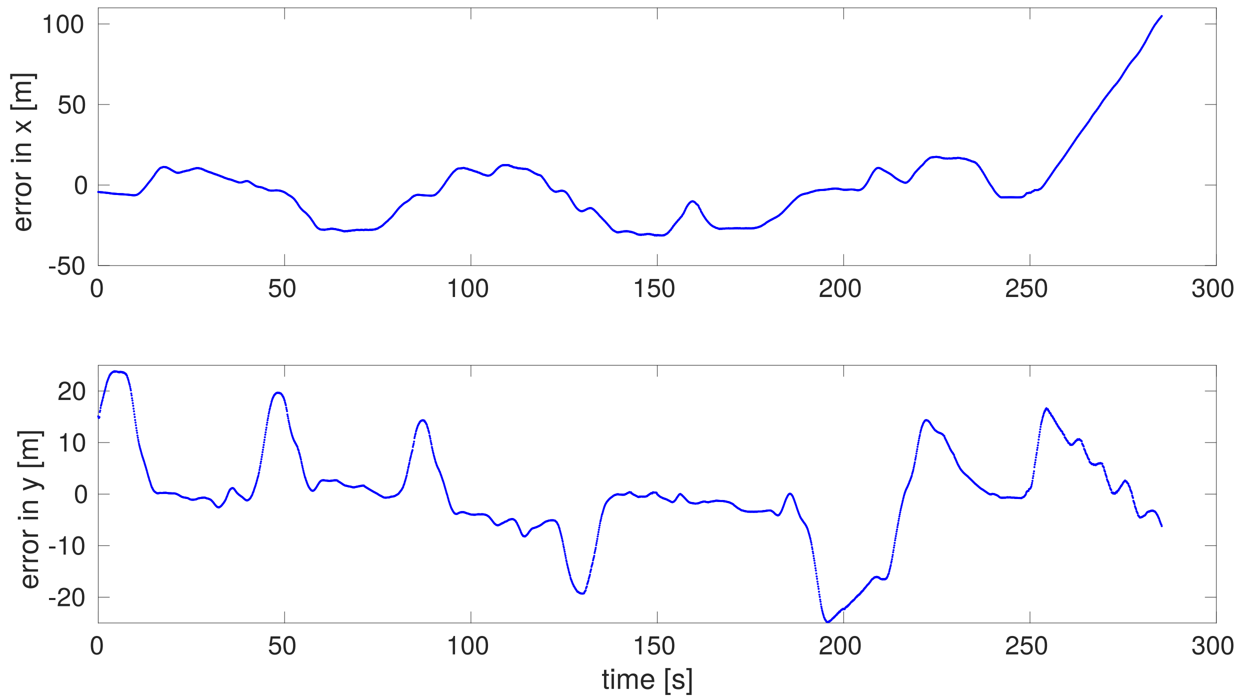


(a) Estimated trajectory using the original LOAM algorithm.

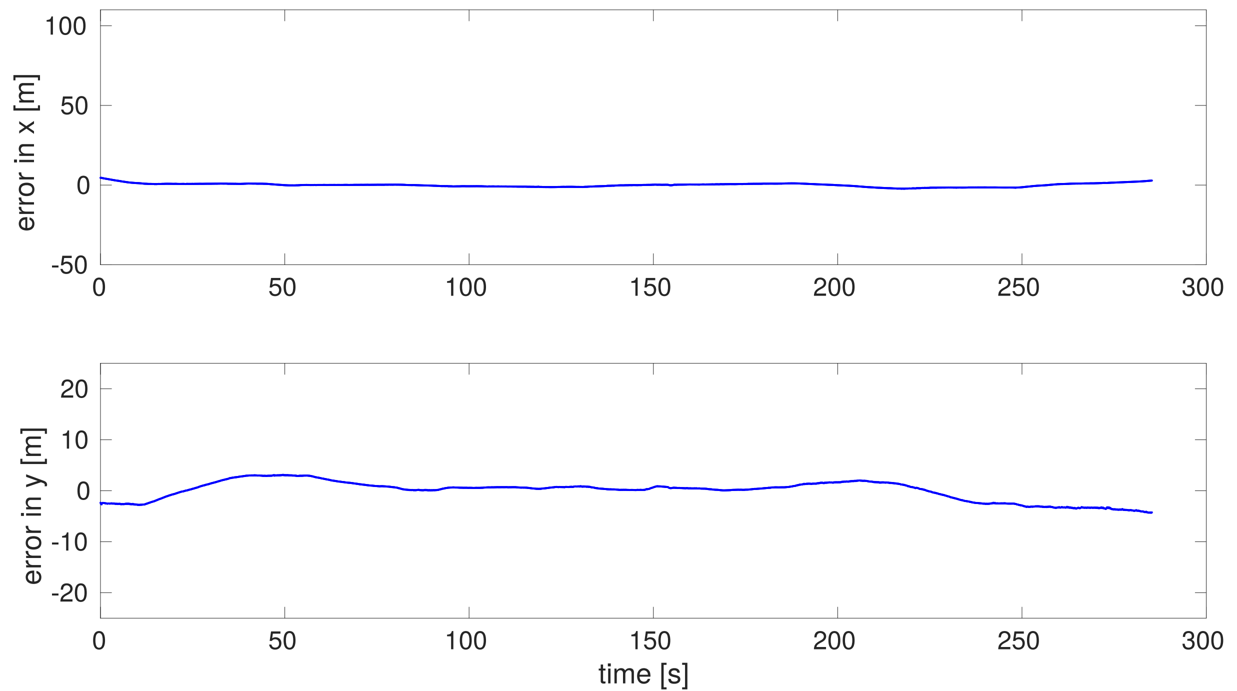


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure 4.24: Estimated trajectories compared to ground-truth in the sequence 05.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.25: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 05.

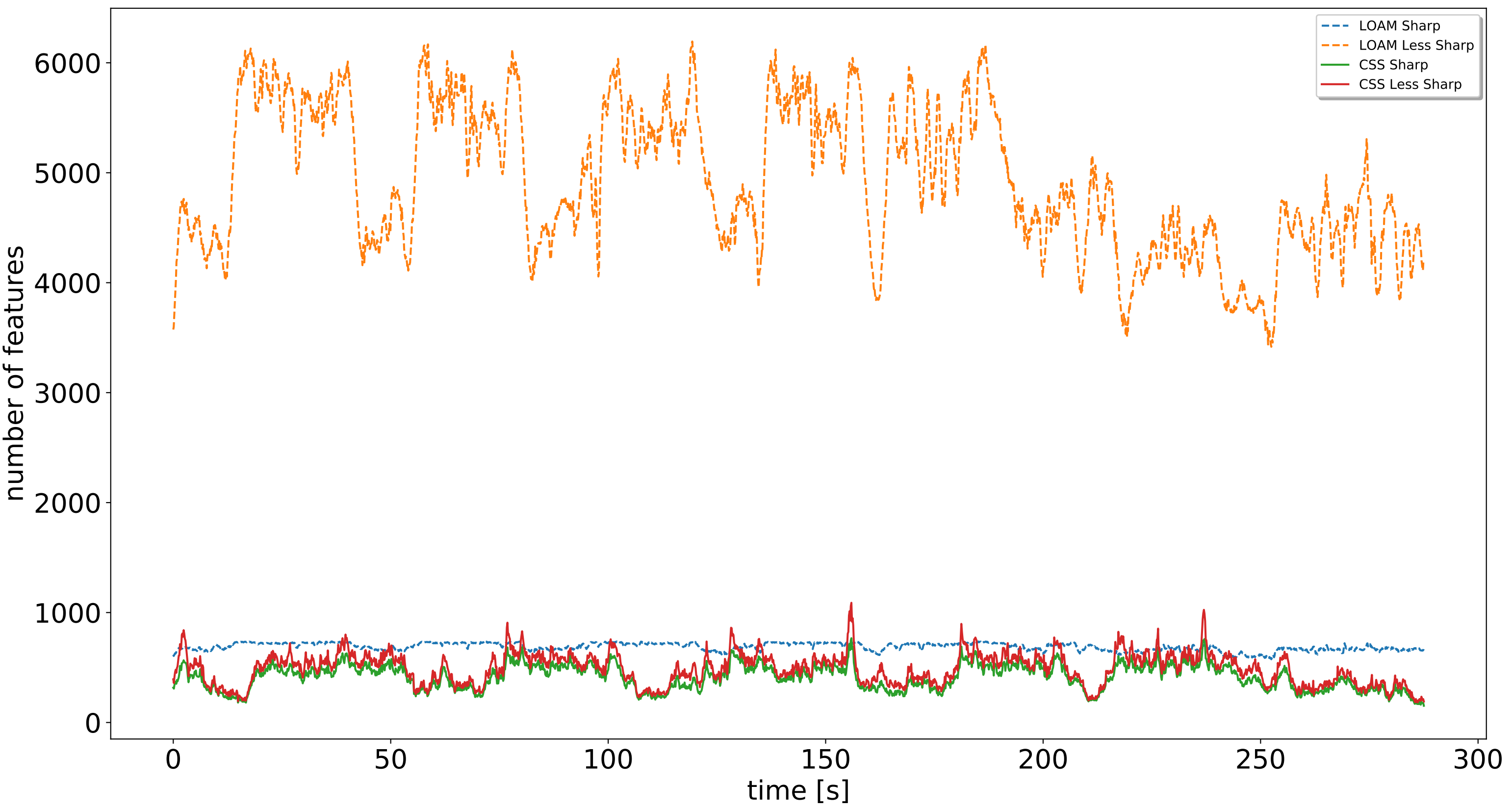
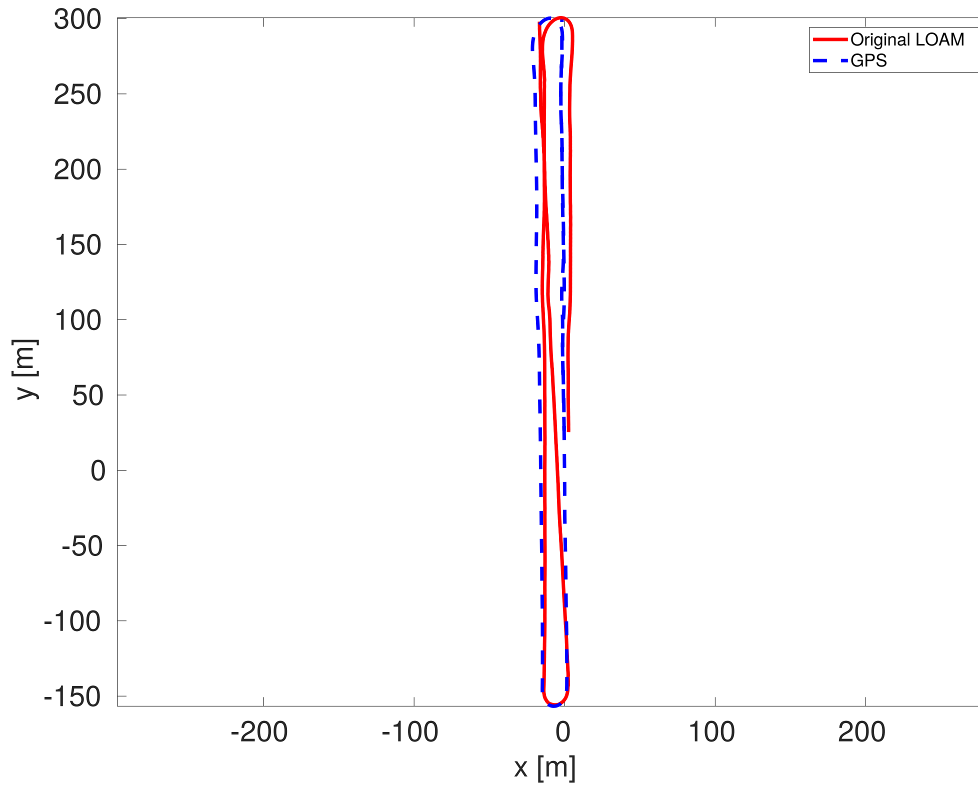
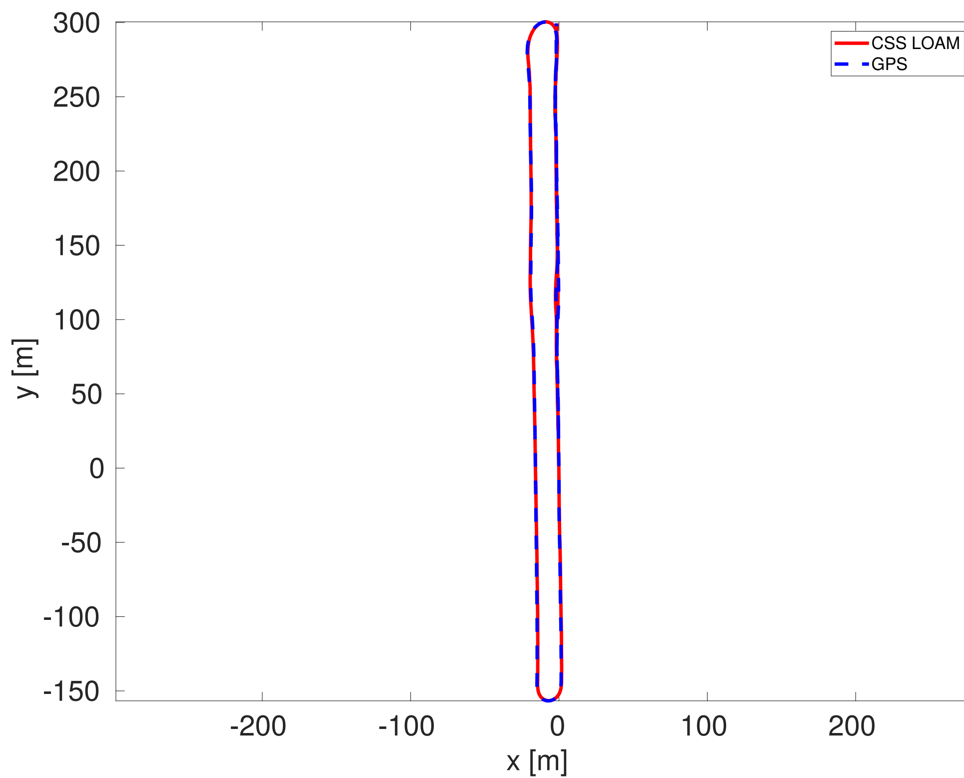


Figure 4.26: Number of features in the sequence 05.

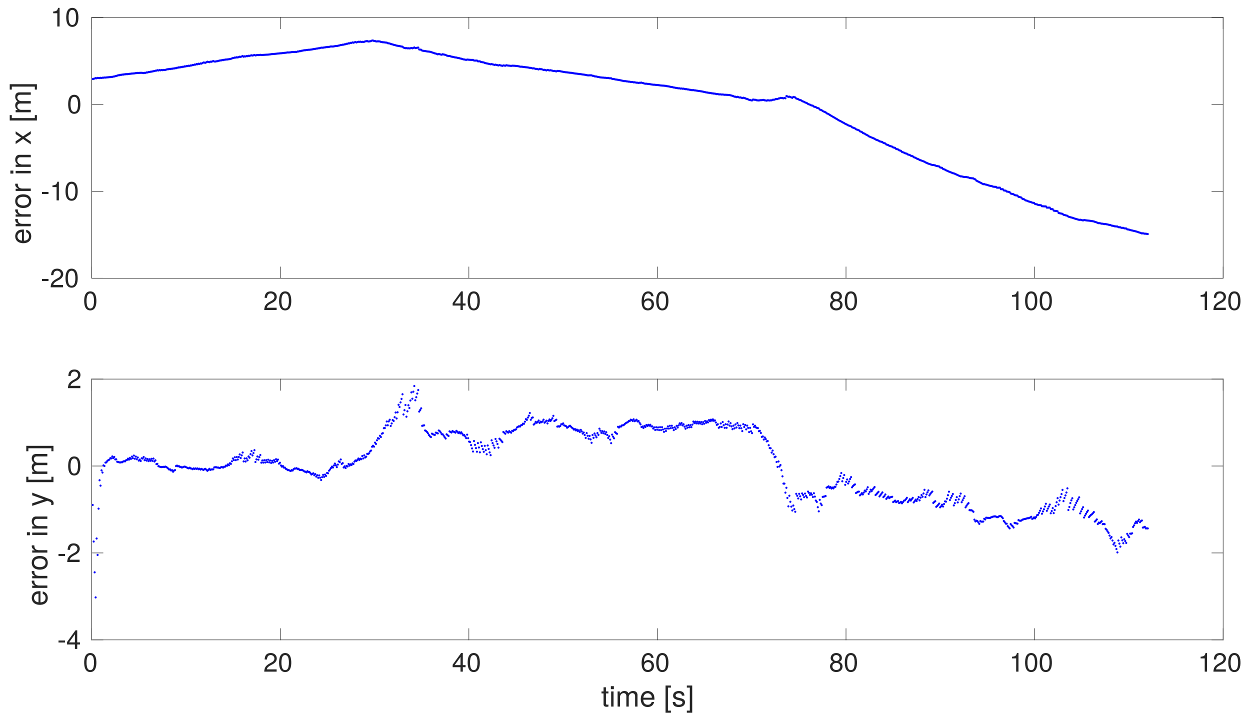


(a) Estimated trajectory using the original LOAM algorithm.

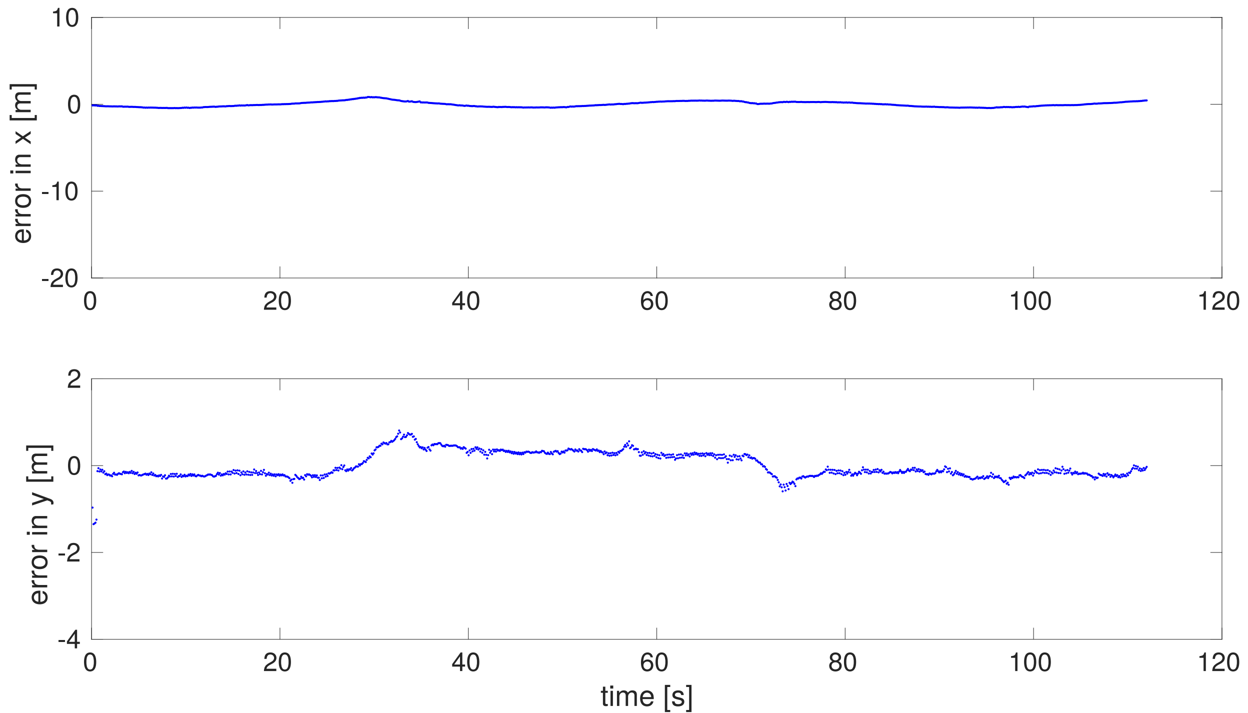


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure 4.27: Estimated trajectories compared to ground-truth in the sequence 06.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.28: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 06.

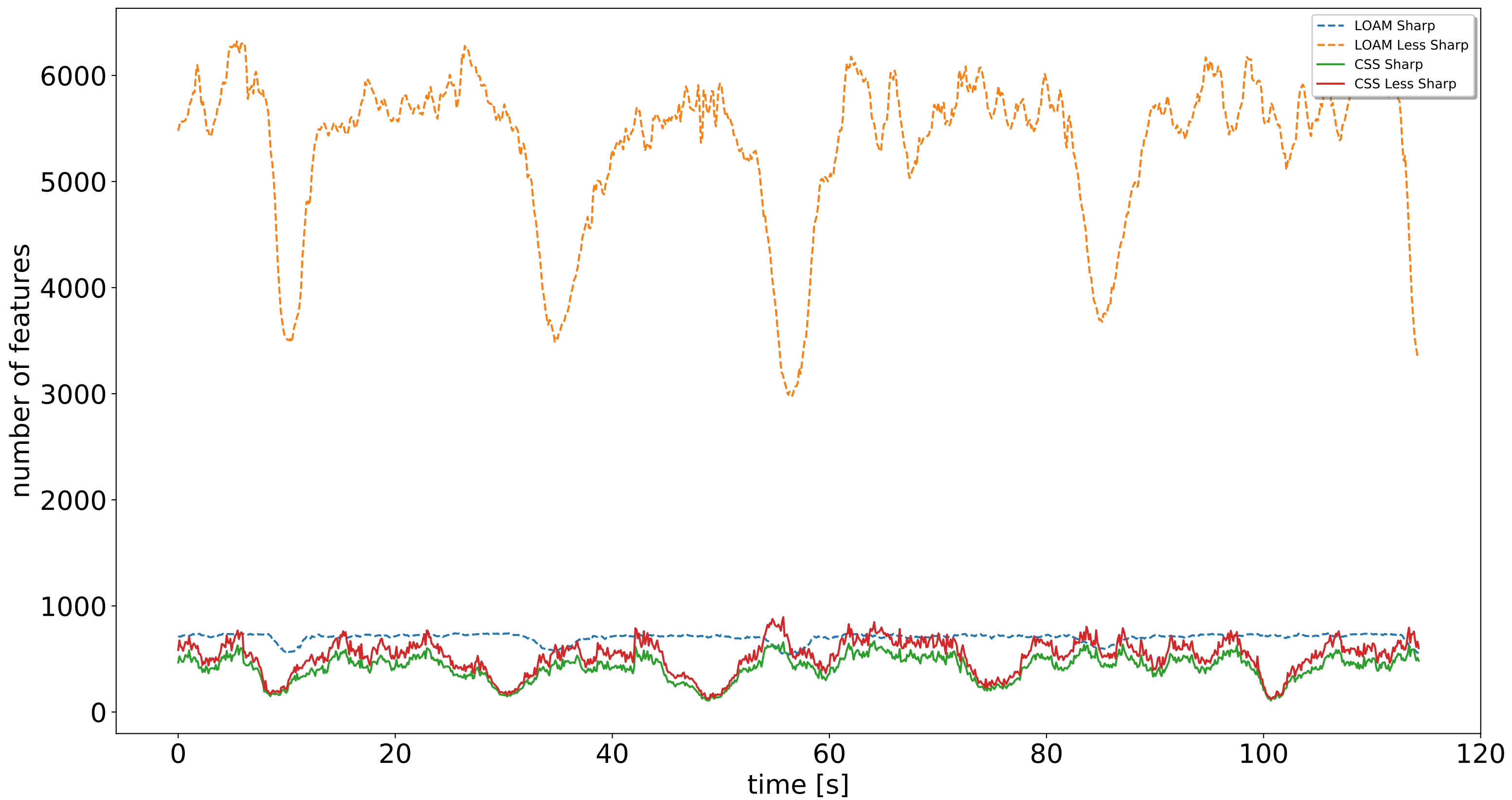
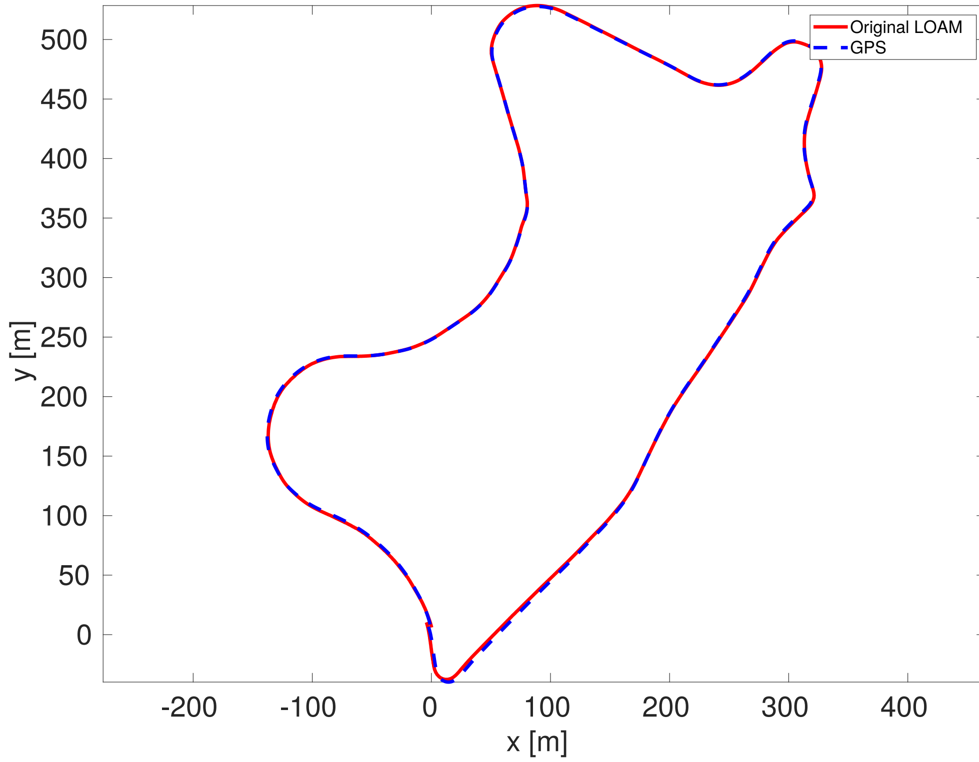
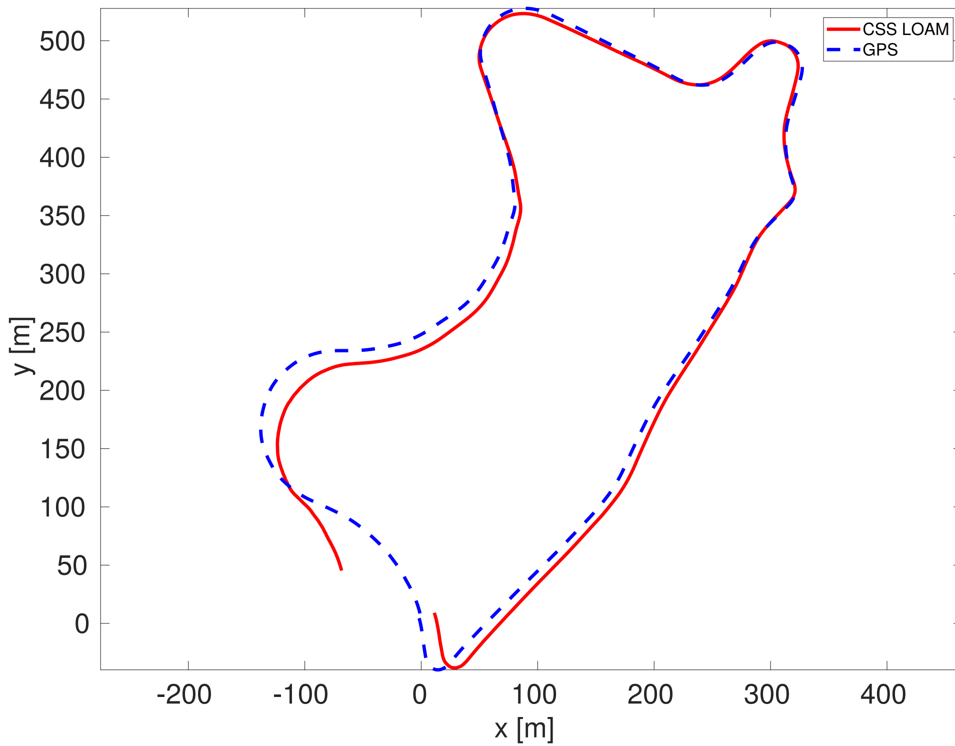


Figure 4.29: Number of features in the sequence 06.

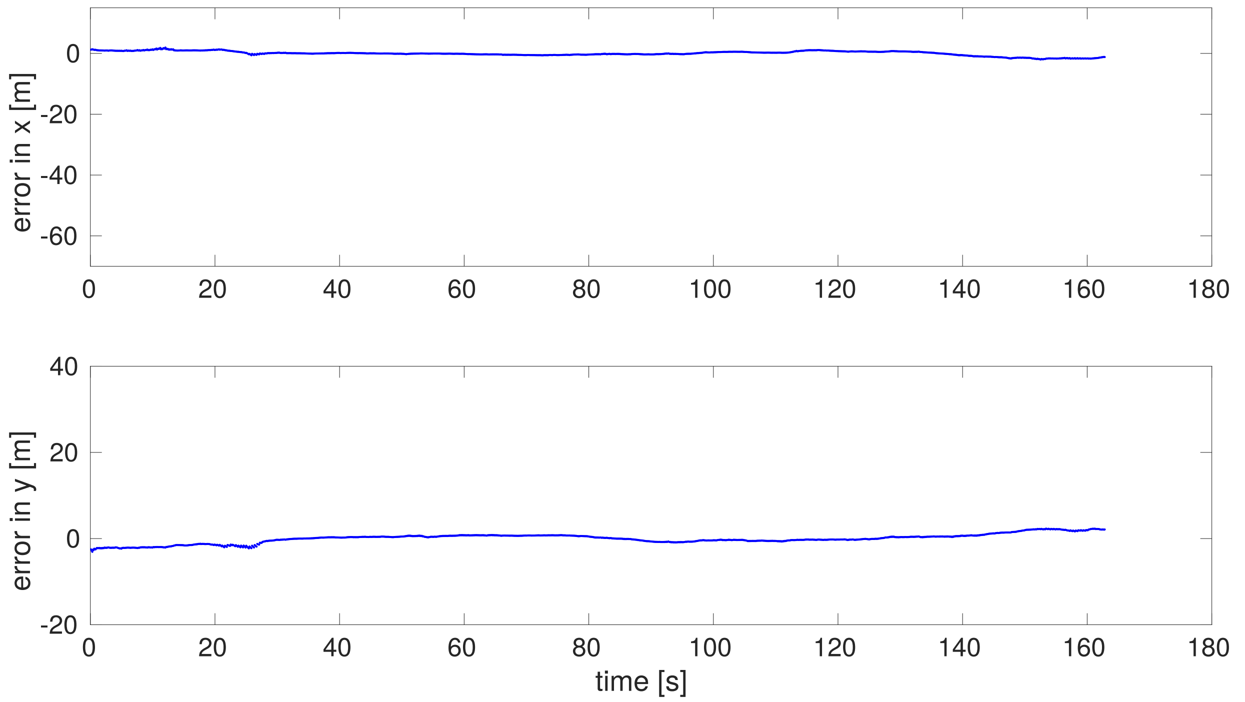


(a) Estimated trajectory using the original LOAM algorithm.

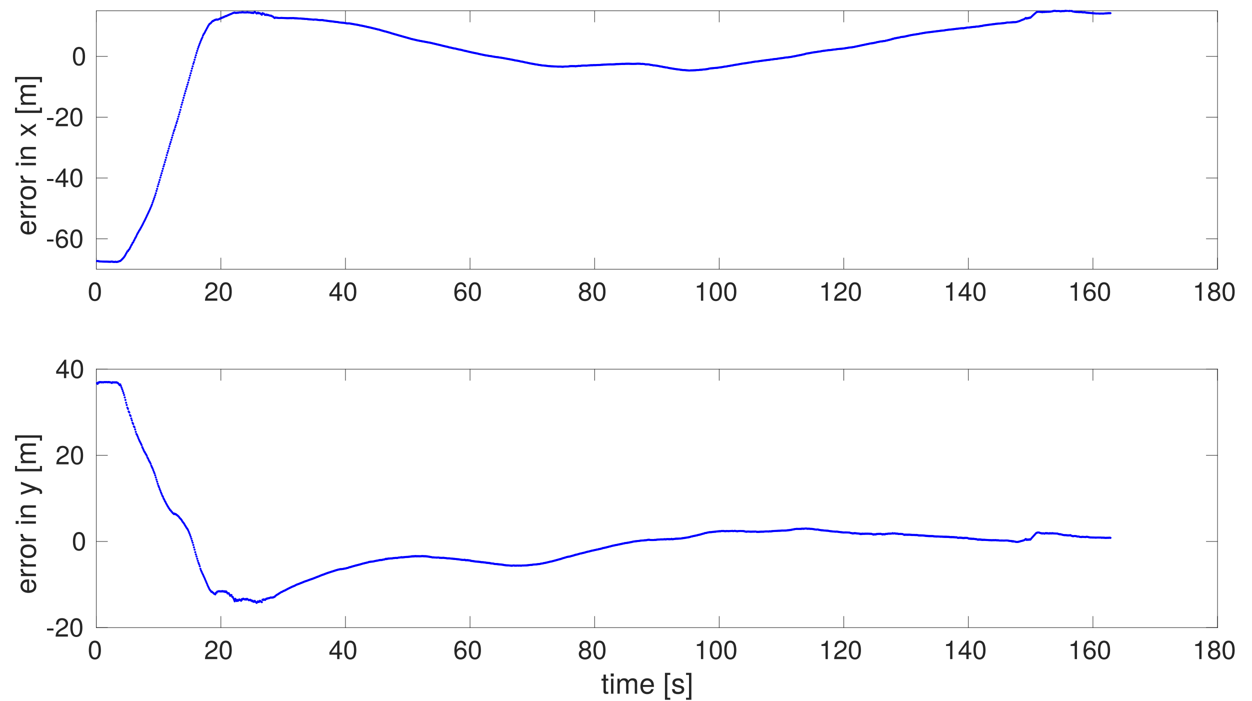


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure 4.30: Estimated trajectories compared to ground-truth in the sequence 09.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure 4.31: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 09.

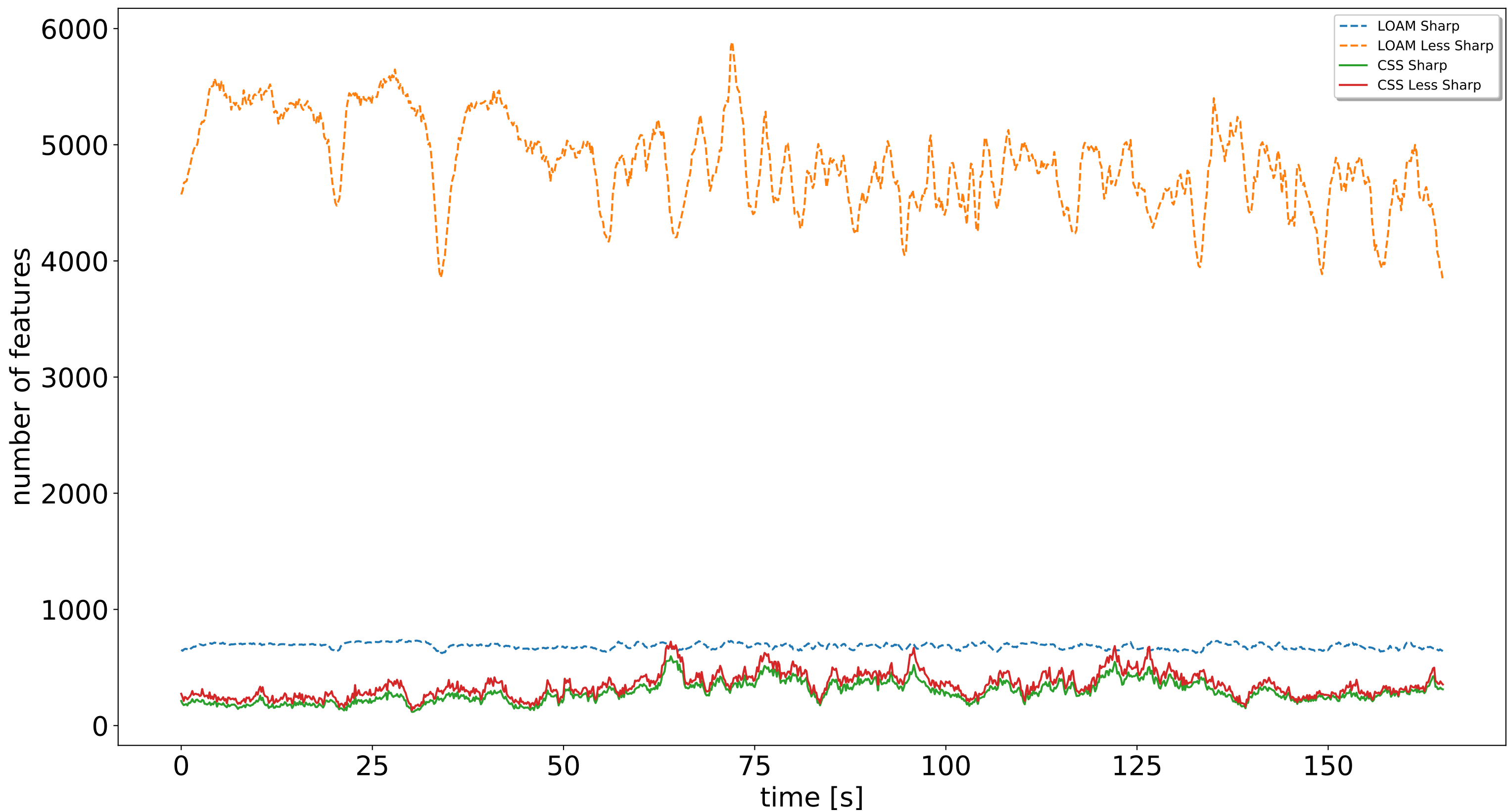


Figure 4.32: Number of features in the sequence 09.

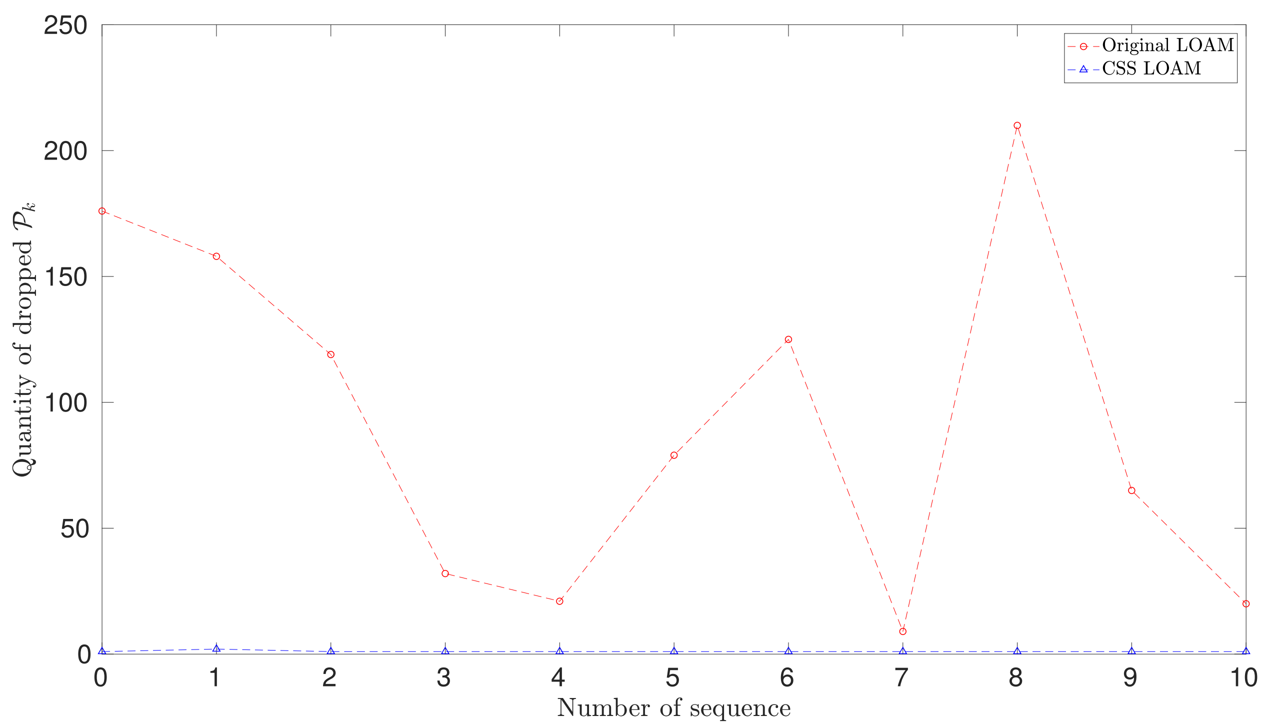


Figure 4.33: Quantity of dropped point clouds, $\mathcal{P}_k$, per sequence.

Chapter 5

Conclusions

This thesis addressed the problem of the misdetection of some features and the failure to account for the sensor range noise in the LiDAR Odometry and Mapping (LOAM) algorithm. To solve this, we proposed to replace the original LOAM's feature extractor with the Curvature Scale Space (CSS) algorithm.

The choice of the CSS algorithm was achieved after the study of the most important feature extraction algorithms in the literature. This study included algorithms such the Harris, SIFT, ISS, RANSAC, Hough Transform, Split and Merge and Adaptive Curvature Estimation algorithms including also the CSS algorithm. Among these algorithms, some of them were discarded because of their computational complexity and others because of the poor results presented in previous literature. Finally, the CSS algorithm was chosen because of its robustness to range noise.

The CSS algorithm was implemented and adapted for the Velodyne 3D LiDAR data. This task was completed by extracting features for each individual elevational scan of the LiDAR. The adapted CSS feature extractor was inserted with success in the LOAM algorithm in replacement of its original feature extractor. The CSS features replaced only the LOAM features that corresponded to the non-smooth or sharp features.

The original LOAM and CSS LOAM algorithms were compared in different scenarios using simulated and real data including the KITTI Dataset. Simulated data was important to test the algorithms with different levels of sensor range noise. For all the evaluated levels of noise, the CSS LOAM algorithm obtained a low error rate according to the Optimal Sub-Pattern Assignment (OSPA) and the Absolute Trajectory Error (ATE) metrics. In the case of the OSPA metric, the mean OSPA error of the CSS LOAM algorithm reached a value equal to 3.94 meters, while the mean OSPA error of the original LOAM algorithm was 4.58 meters. In the case of the ATE metric, the mean error of the CSS LOAM algorithm reached a value equal to 0.06 meters, while the mean error of the original LOAM algorithm was 0.12 meters. In the real data, obtained from the surroundings of the Universidad de Chile, the CSS LOAM algorithm slightly improved the trajectory generated by the original LOAM algorithm. The ATE mean translation difference between the original LOAM and the CSS LOAM algorithms reached a value of 0.91 meters, superior to the CSS LOAM algorithm. This difference mainly

occurred in a specific part of trajectory highlighted in the results section.

A significant finding occurred in the KITTI dataset. In this experiment, the original LOAM algorithm produced high ATE errors in many sequences. This problem occurred because the original frequency, in which the LiDAR returned the point clouds, was taken into account. It is important to note that in the KITTI Benchmark ranking, this factor is not considered. Therefore, the number of LOAM features was very high, which made the task of feature matching computationally complex. This complexity resulted in some point clouds not being processed because of the frequency in which the LiDAR was returning the point clouds. This issue did not occur with the CSS LOAM algorithm, because the number of features was lower and the feature matching task could be completed before the arrival of a new point cloud. Additionally, the results obtained by the CSS LOAM algorithm in this dataset depended on the number of features detected. This allows us to conclude that the CSS feature extractor only works well in structured environments where there are variety of objects such as walls, corners, lamp post or trees.

Finally, the adaptation of the CSS feature extractor to process point clouds obtained with any type of 3D LiDAR and its inclusion to other SLAM algorithms is considered as a future work possibility. Additionally, the inclusion of the CSS feature extractor into the LeGO-LOAM algorithm [35] could be considered. The LeGO-LOAM algorithm is a modified version of the LOAM algorithm which is optimized for a horizontally placed LiDAR on a ground vehicle. The main difference between the LOAM algorithm and the LeGO-LOAM algorithm is that the LeGO-LOAM algorithm assumes that there is always a ground plane in the current 3D scan. Although the LeGO-LOAM algorithm does not improve the results of the LOAM algorithm in the KITTI Visual Odometry Benchmark Dataset, it speeds up the processing time of the LOAM algorithm.

Chapter 6

Bibliography

[1] M. D. Adams and A. Kerstens. Tracking naturally occurring indoor features in 2-D and 3-D with lidar range/amplitude data. *International Journal of Robotics Research*, 17(9):907–923, 1998.

[2] Haruo Asada and Michael Brady. The Curvature Primal Sketch. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(1):2–14, 1986.

[3] T. Bailey and Hugh Durrant-whyte. Simultaneous Localization and Mapping (SLAM): Part II. *IEEE Robotics and Automation Magazine*, 13(3):108–117, 2006.

[4] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (SLAM): Part I. *IEEE Robotics and Automation Magazine*, 13(3):108–117, 2006.

[5] G A Borges and M.-J. Aldon. A Split-and-Merge segmentation algorithm for line extractions in 2-{D} range images. *15th International Conference on Pattern Recognition*, pages 441–444, 2000.

[6] Geovany Araujo Borges and Marie-José Aldon. Line Extraction in 2D Range Images for Mobile Robotics. *Journal of Intelligent and Robotic Systems*, 40:267–297, 2004.

[7] J. A. Castellanos and J. D. Tardos. Laser-Based Segmentation and Localization for a Mobile Robot. *Robotics and Manufacturing: Recent Trends in Research and Applications*, 6:101–108, 1996.

[8] Filipe Silvio and Alexandre Luis. A Comparative Evaluation of 3D Keypoint Detectors. In *9th International Conference on Computer Vision Theory and Applications*, pages 145–148, 2014.

[9] Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with. *Communications of the ACM*, 24:381–395, 1981.

[10] Orazio Gallo, Roberto Manduchi, and Abbas Rafii. CC-RANSAC: Fitting planes in the presence of multiple surfaces in range data. *Pattern Recognition Letters*, 32(3):403–410,

2011.

[11] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[12] C. Harris and M. Stephens. A Combined Corner and Edge Detector. *Procedings of the Alvey Vision Conference 1988*, pages 23.1–23.6, 1988.

[13] P.V.C. Hough. Method and means for recognizing complex patterns, 1962.

[14] Jun Hyuck Im, Sung Hyuck Im, and Gyu In Jee. Vertical corner feature based precise vehicle localization using 3D LIDAR in Urban area. *Sensors (Switzerland)*, 16(8), 2016.

[15] Felipe Inostroza. The estimation of detection statistics in simultaneous localization and mapping. Master's thesis, Universidad de Chile, 2015.

[16] P. Jensfelt and H. Christensen. Laser Based Position Acquisition and Tracking in an Indoor Environment. In *IEEE International Symposium on Robotics and Automation*, 1998.

[17] Anis Koubaa. *Robot Operating System (ROS) The complete reference*. Springer, 2016.

[18] Yangming Li and Edwin B. Olson. A general purpose feature extractor for Light Detection and Ranging data. *Sensors (Switzerland)*, 10(11):10356–10375, 2010.

[19] Yangming Li and Edwin B. Olson. Extracting general-purpose features from LIDAR data. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1388–1393, 2010.

[20] David G Lowe. Distinctive Image Features from Scale Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[21] R. Madhavan and H. F. Durrant-Whyte. Natural landmark-based autonomous vehicle navigation. *Robotics and Autonomous Systems*, 2004.

[22] Farzinand Mokhtarian and Alan Mackworth. Scale-Based Description and Recognition of Planar Curves and Two-Dimensional Shapes. (1), 1986.

[23] Hans Moravec. Obstacle Avoidance and Navigation in the Real World by Seeing Robot Rover. *Tech Report CMU-RI-TR-3, Carnegie-Mellon University, Robotics Institute*, 1980.

[24] John Stephen Mullane, Ba-Ngu Vo, Martin David Adams, and Ba-Tuong Vo. *Random Finite Sets for Robot Mapping & SLAM: New Concepts in Autonomous Robotic Map Representations*. Springer, 2011.

[25] V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart. A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics. pages 1929–34, 2005.

[26] P. Núñez, R. Vázquez-Martín, J. C. del Toro, A. Bandera, and F. Sandoval. Natural landmark extraction for mobile robot navigation based on an adaptive curvature estimation. *Robotics and Autonomous Systems*, 56(3):247–264, 2008.

[27] Theodosios Pavlidis and Steven L. Horowitz. Segmentation of Plane Curves. *IEEE Transactions on Computers*, C-23(8):860–870, 1974.

[28] S. T. Pfister, S. I. Roumeliotis, and J. W. Burdick. Weighted Line Fitting Algorithms for Mobile Robot Map Building and Efficient Data Representation. In *IEEE International Conference on Robotics and Automation*, 2003.

[29] Tomaso A Poggio and a L Yuille. Scaling theorems for zero crossings. *IEEE transactions on pattern analysis and machine intelligence*, 8(1):15–25, 1986.

[30] W. Prawira, E. Nasrullah, S. R. Sulistiyanti, and F. X.A. Setyawan. The detection of 3D object using a method of a Harris Corner Detector and Lucas-Kanade Tracker based on stereo image. *ICECOS 2017 - Proceeding of 2017 International Conference on Electrical Engineering and Computer Science: Sustaining the Cultural Heritage Toward the Smart Environment for Better Future*, pages 163–166, 2017.

[31] Søren Riisgaard and Morten Rufus Blas. SLAM for Dummies. *A Tutorial Approach to Simultaneous Localization and Mapping*, 22(June):1–127, 2004.

[32] Radu Bogdan Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). *IEEE International Conference on Robotics and Automation*, pages 1 – 4, 2011.

[33] Dominic Schuhmacher, Ba Tuong Vo, and Ba Ngu Vo. A consistent metric for performance evaluation of multi-object filters. *IEEE Transactions on Signal Processing*, 56(8 I):3447–3457, 2008.

[34] Jacopo Serafin, Edwin Olson, and Giorgio Grisetti. Fast and robust 3D feature extraction from sparse point clouds. *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (October):4105–4112, 2016.

[35] Tixiao Shan and Brendan Englot. LeGO-LOAM : Lightweight and Ground-Optimized Lidar Odometry and Mapping on Variable Terrain. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[36] Ivan Sipiran and Benjamin Bustos. Harris 3D: A robust extension of the Harris operator for interest point detection on 3D meshes. *Visual Computer*, 27(11):963–976, 2011.

[37] P Skrzypczynski. Building geometrical map of environment using IR range finder data. *Intelligent Autonomous Systems*, pages 408–412, 1995.

[38] Fan Tang, M Adams, J. Ibanez-Guzman, and W.S. Wijesoma. Pose invariant, robust feature extraction from data with a modified scale space approach. *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, pages 3173–3179 Vol.3, 2004.

[39] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):1999–2000, 2002.

[40] Zhong Yu. Intrinsic shape signatures: A shape descriptor for 3D object recognition. *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops 2009*, pages 689–696, 2009.

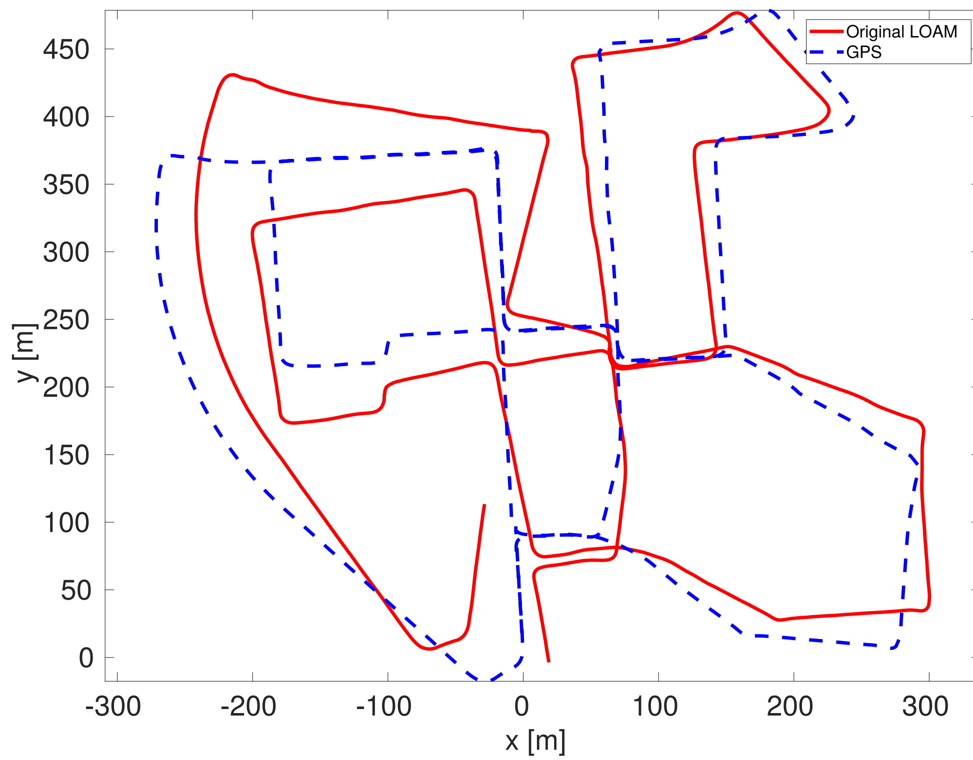
[41] Ji Zhang. *Online Lidar and Vision based Ego-motion Estimation and Mapping*. PhD thesis, 2017.

[42] Ji Zhang and Sanjiv Singh. LOAM : Lidar Odometry and Mapping in Real - time. *Robotics: Science and Systems*, 2014.

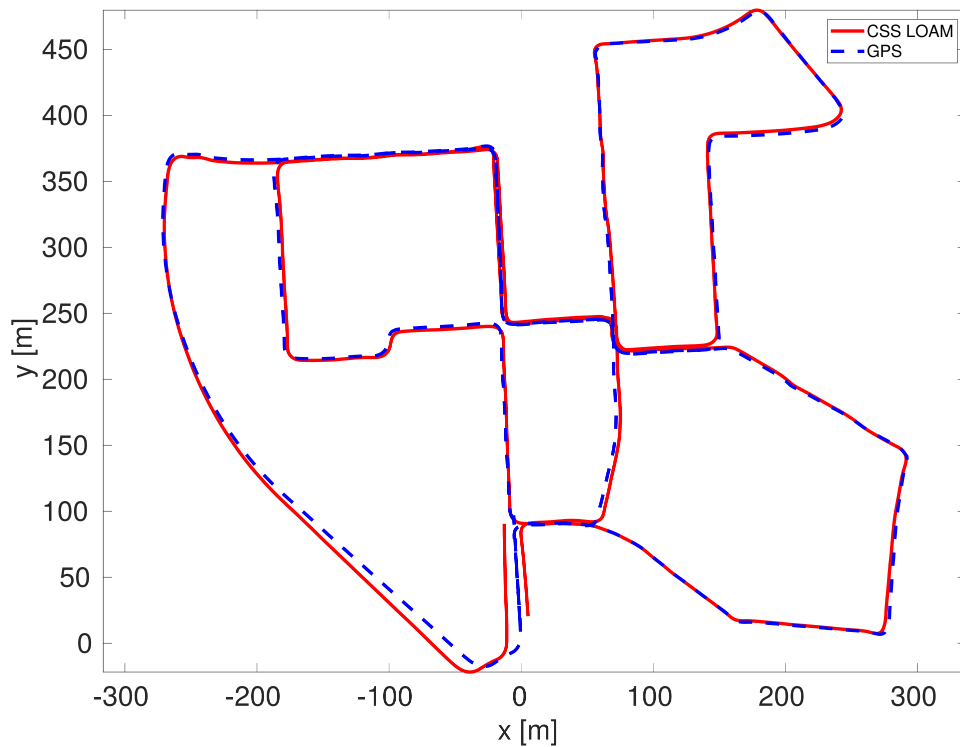
Appendix A

KITTI Dataset Experiments

In this appendix all the sequences from the KITTI dataset that were tested for this thesis are shown, the most representative sequences are shown in chapter 4.

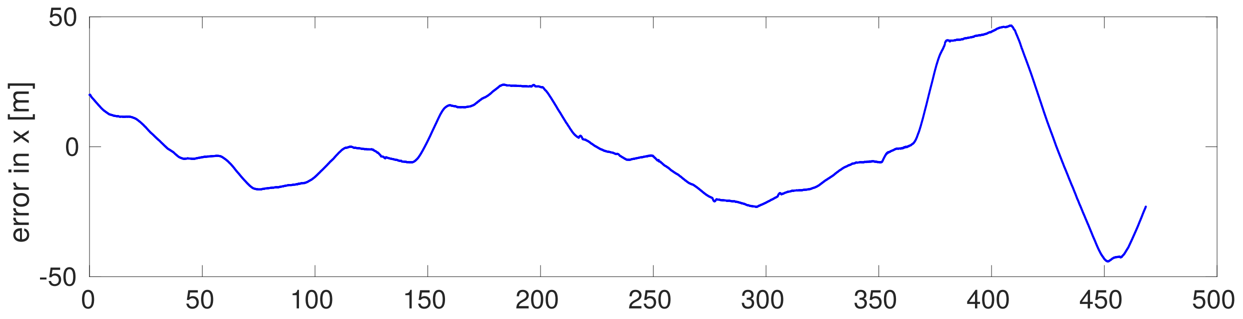


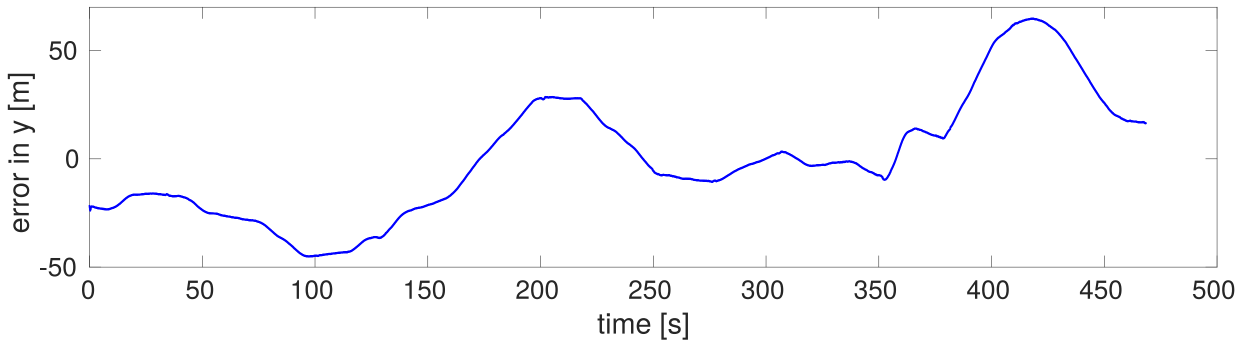
(a) Estimated trajectory using the original LOAM algorithm.



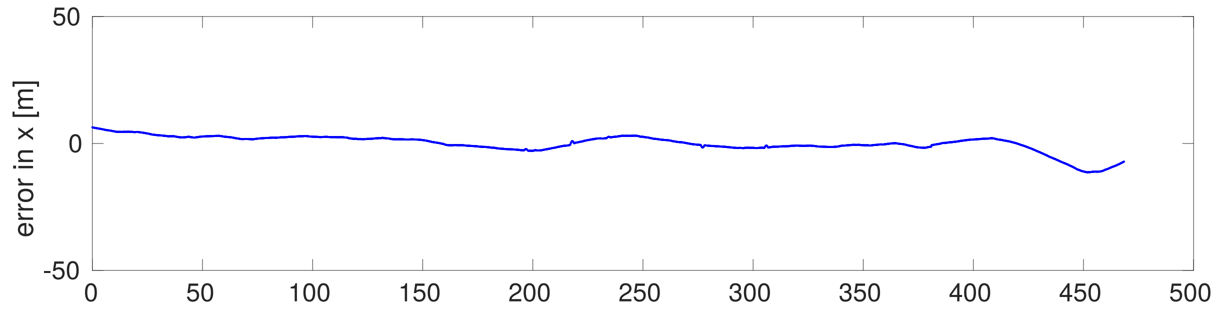
(b) Estimated trajectory using the CSS LOAM algorithm.

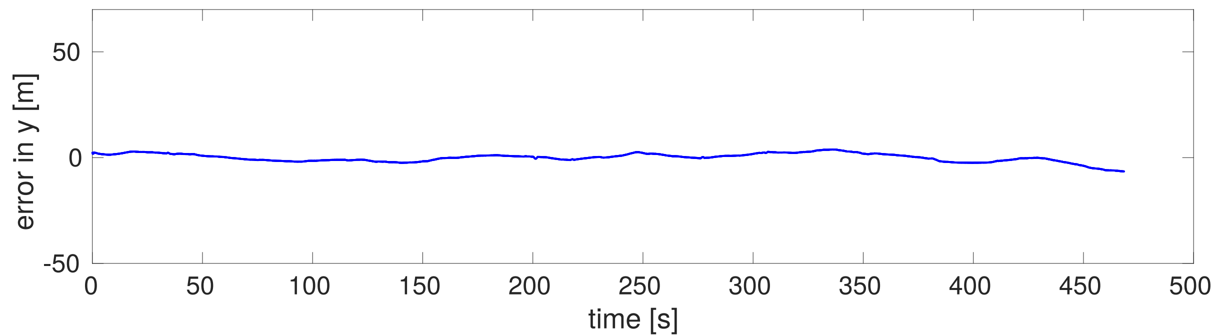
Figure A.1: Estimated trajectories compared to ground-truth in the sequence 00.





(a) Errors in x, y produced by the Original LOAM algorithm.





(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.2: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 00.

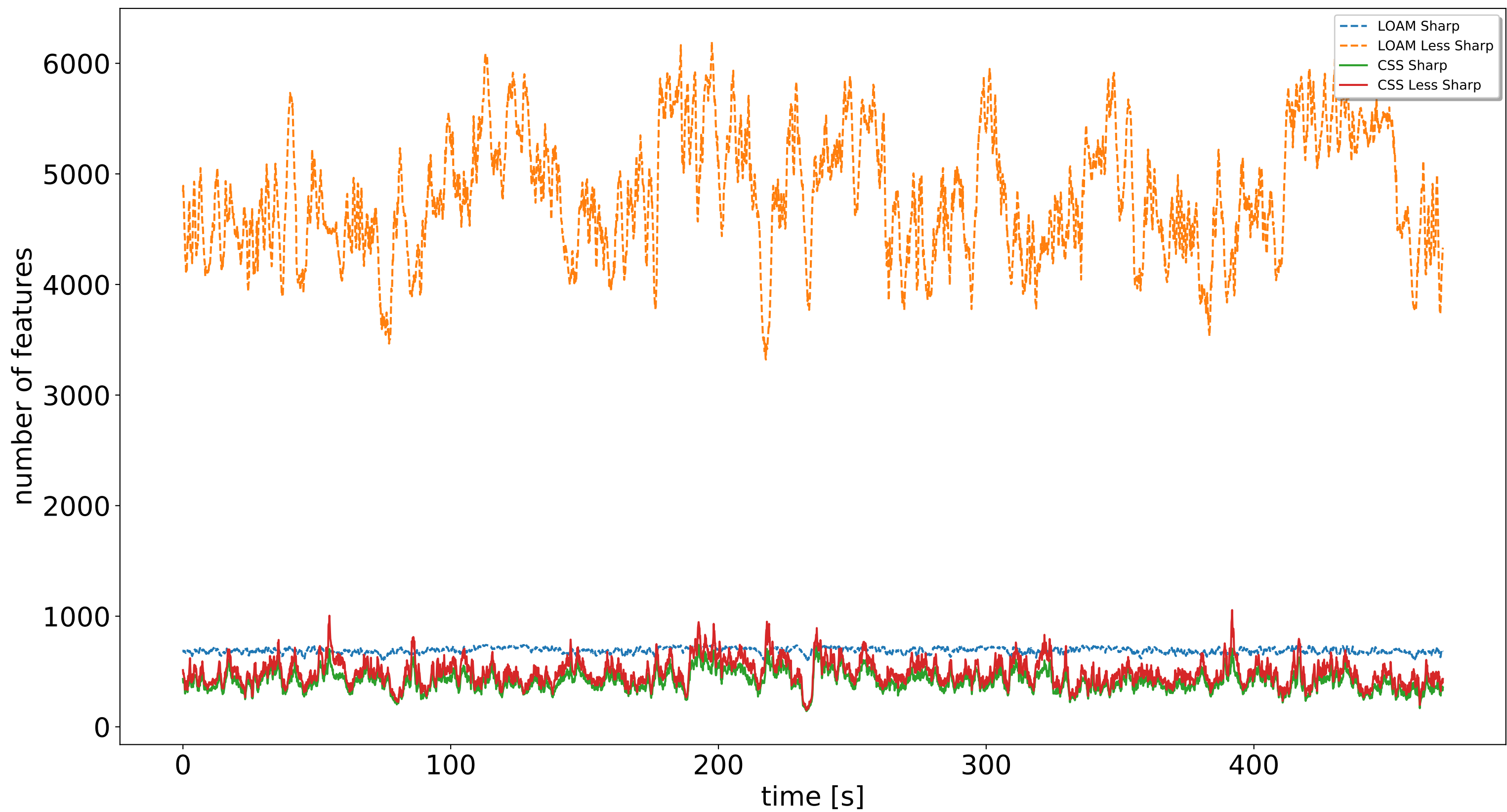
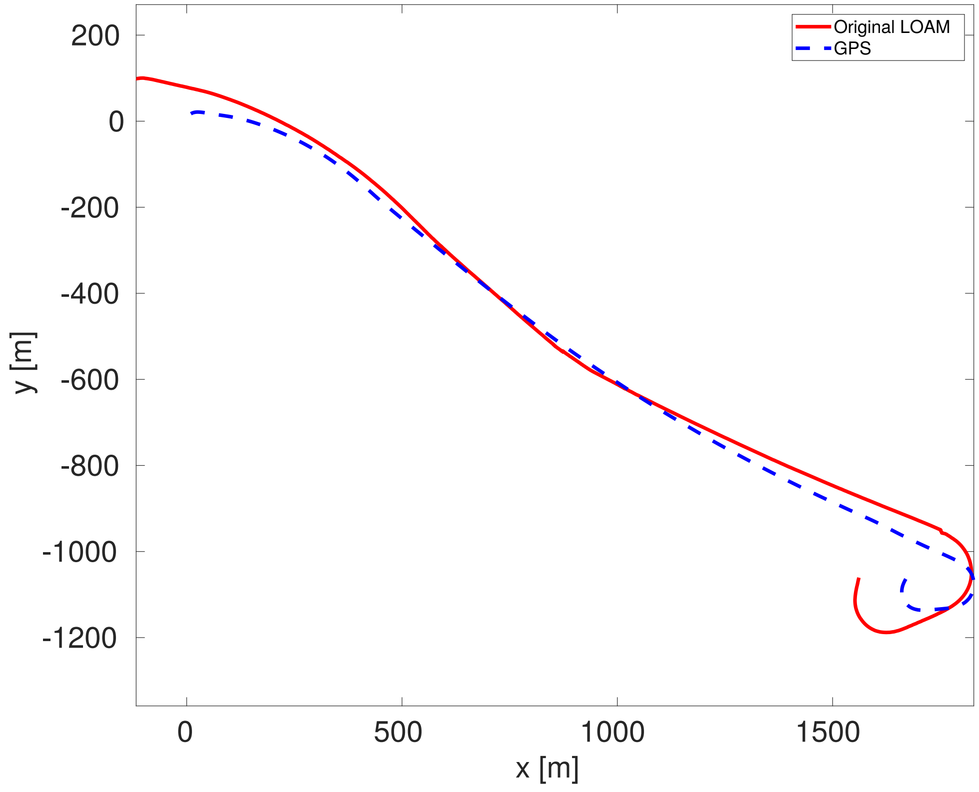
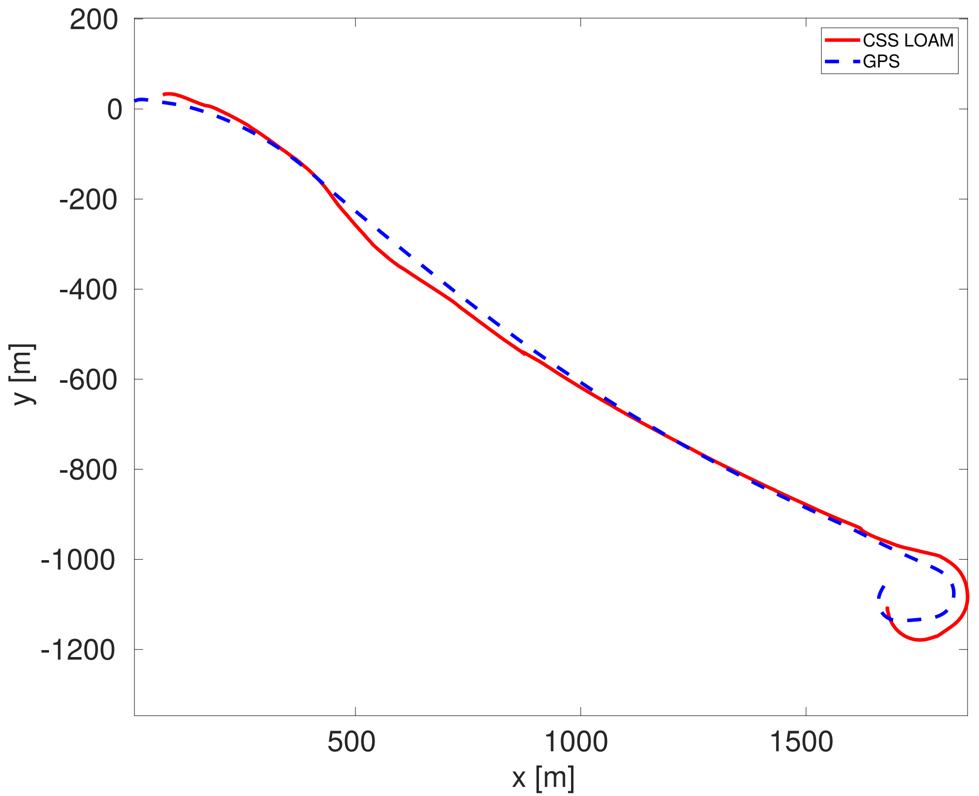


Figure A.3: Number of features in the sequence 00.

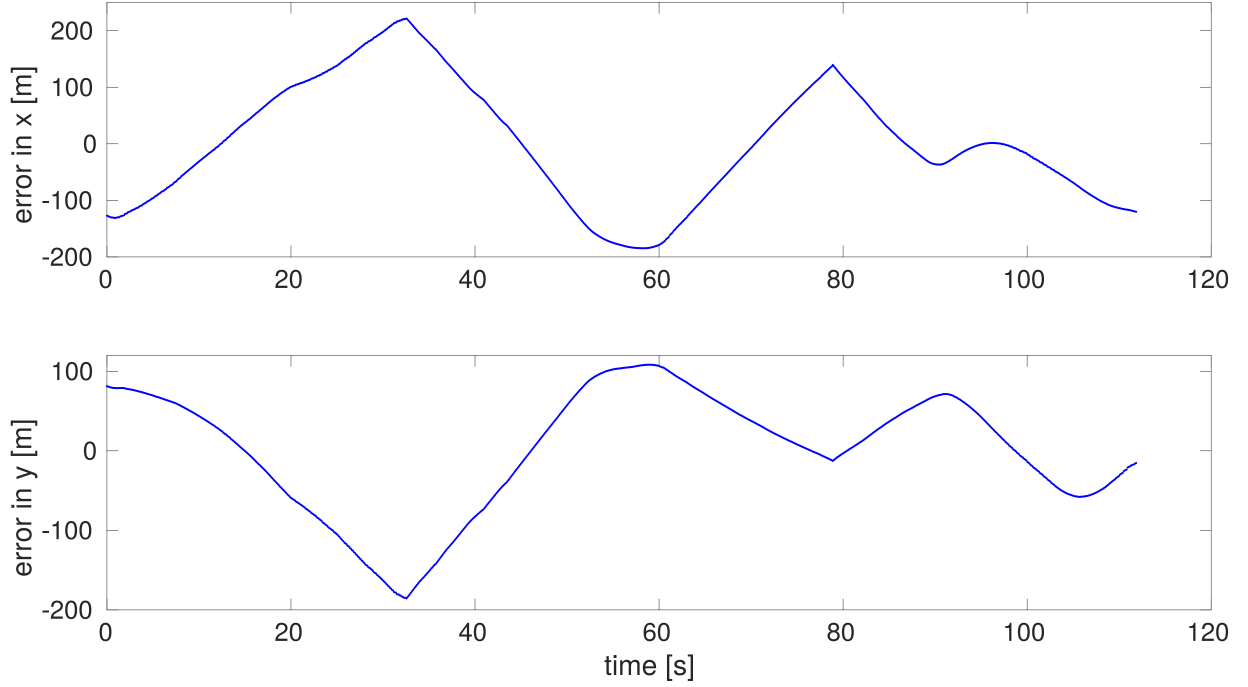


(a) Estimated trajectory using the original LOAM algorithm.

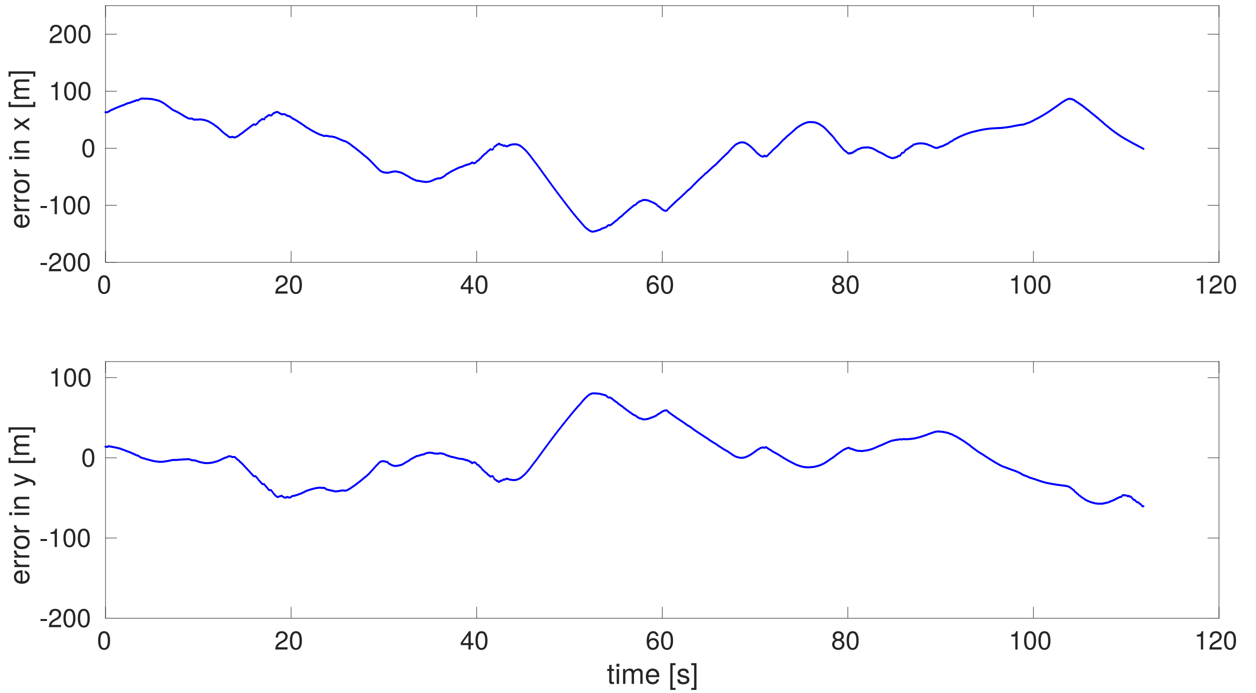


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.4: Estimated trajectories compared to ground-truth in the sequence 01.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.5: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 01.

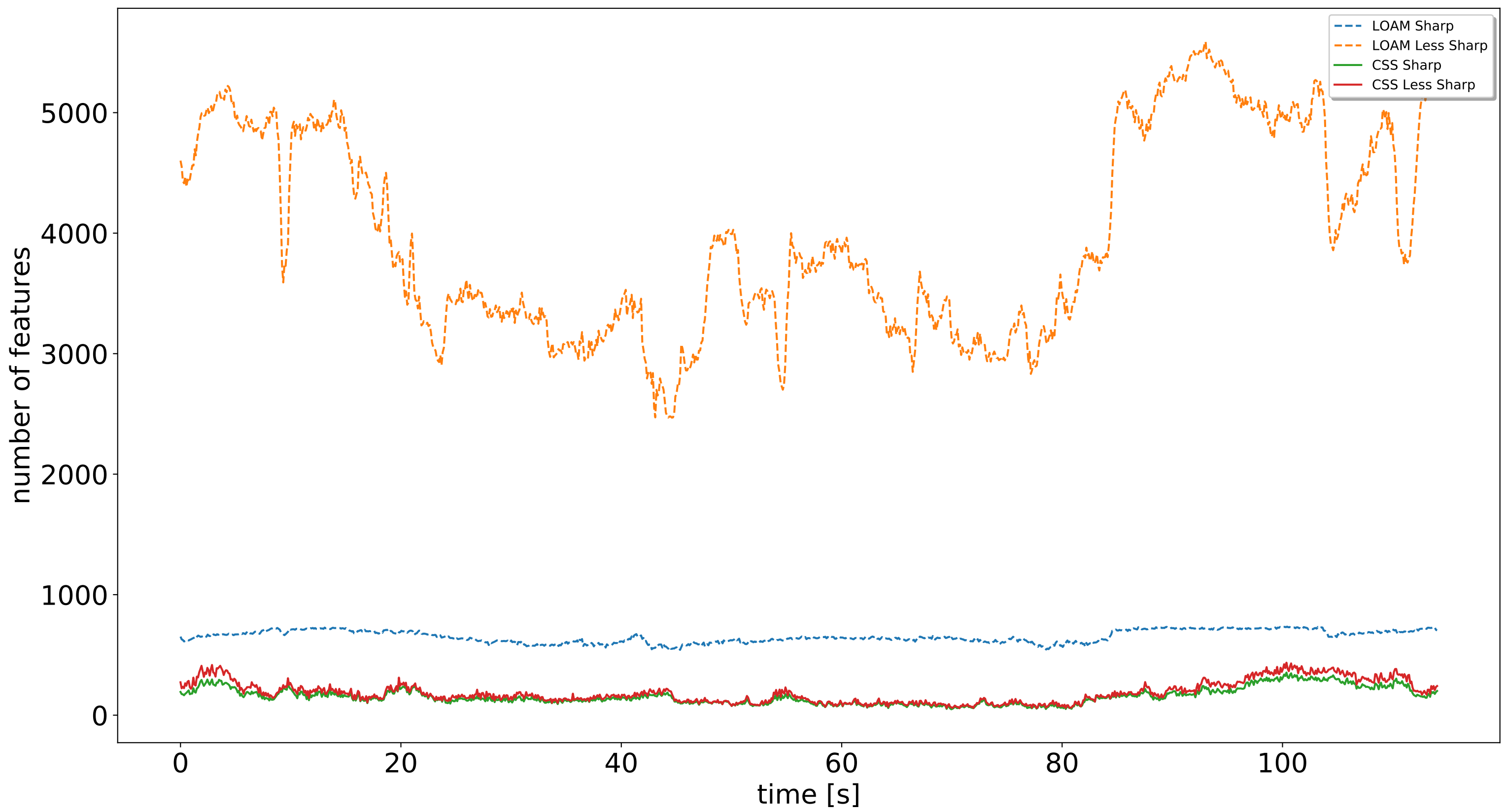
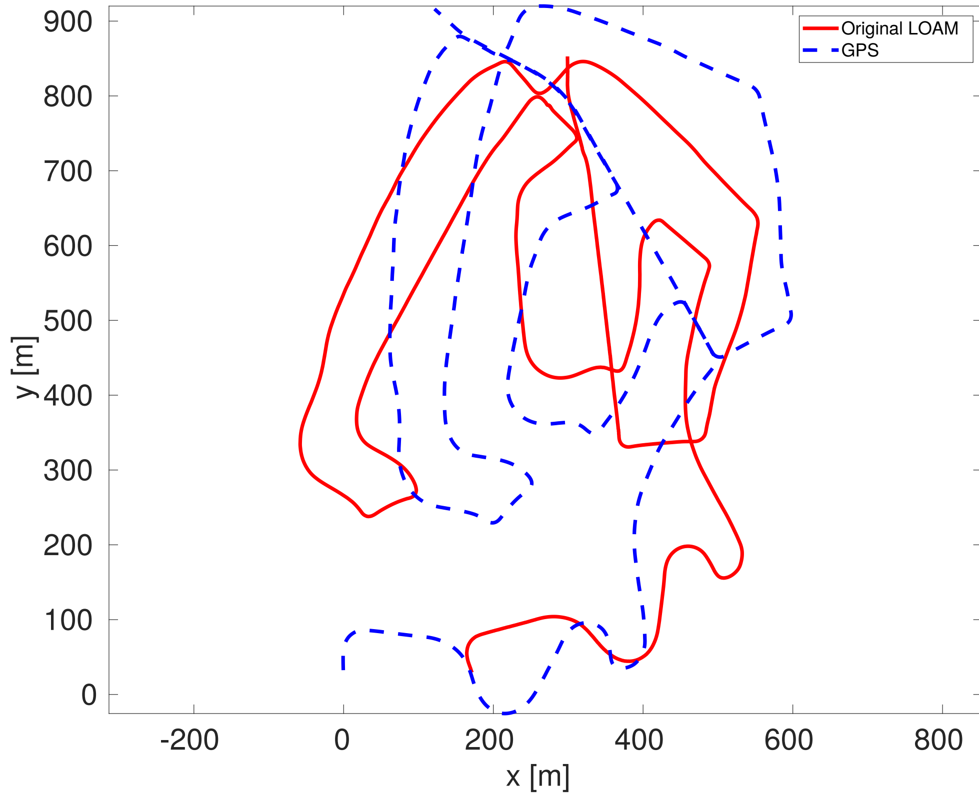
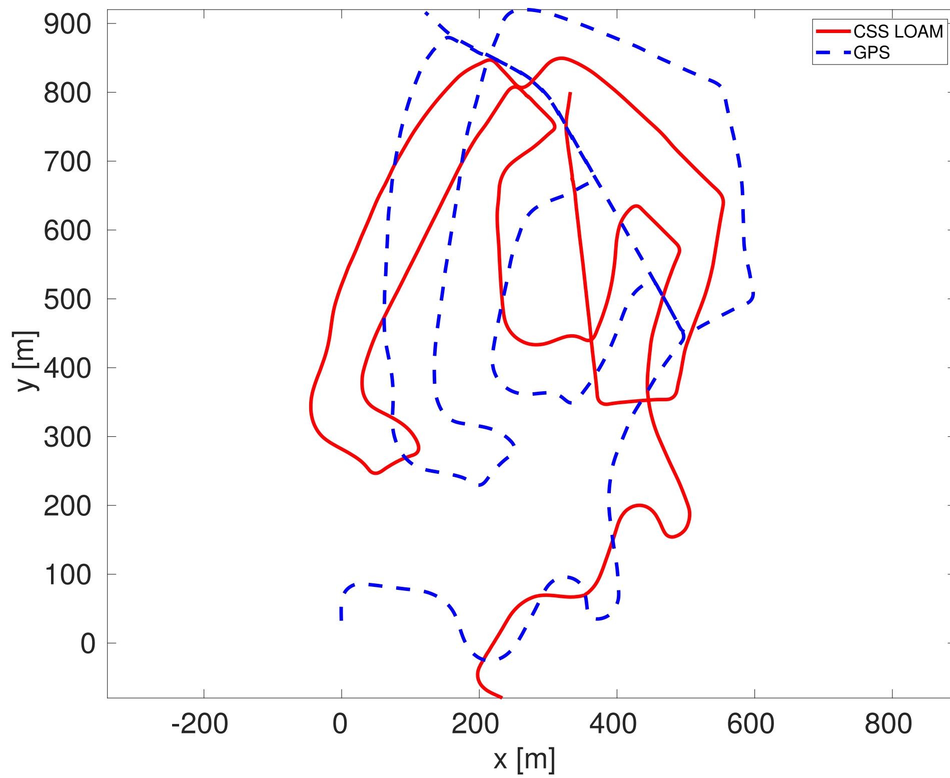


Figure A.6: Number of features in the sequence 01.

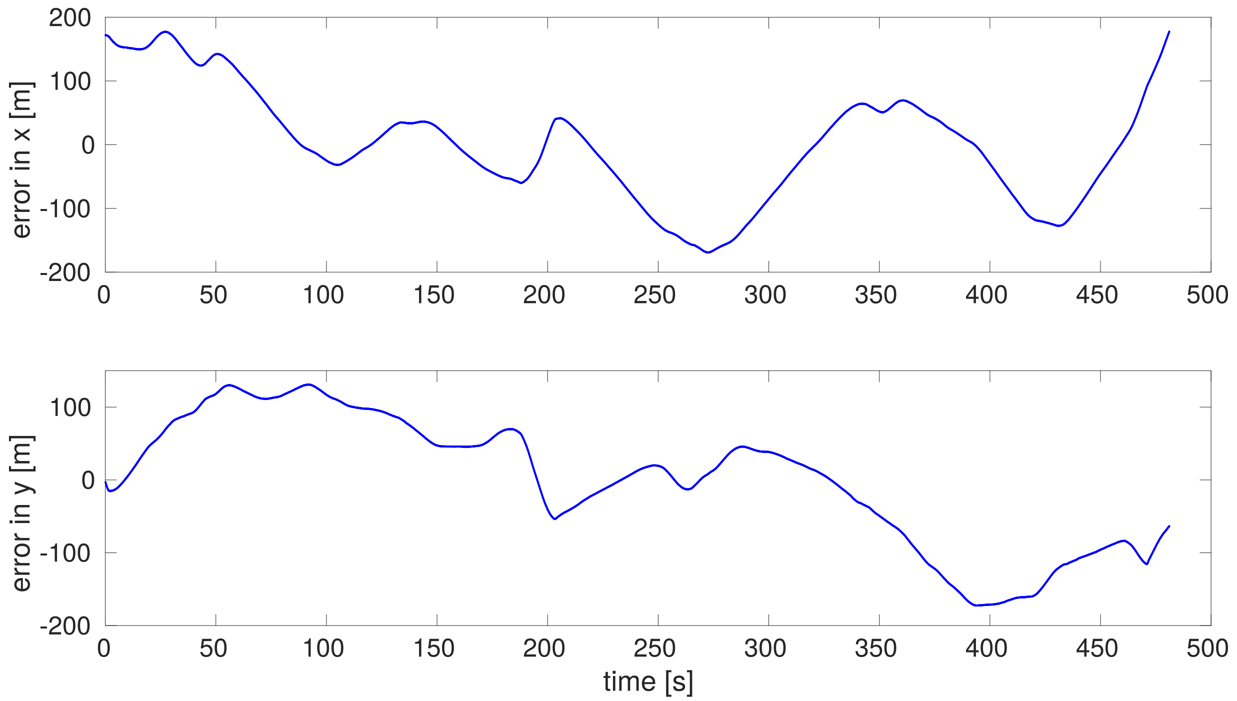


(a) Estimated trajectory using the original LOAM algorithm.

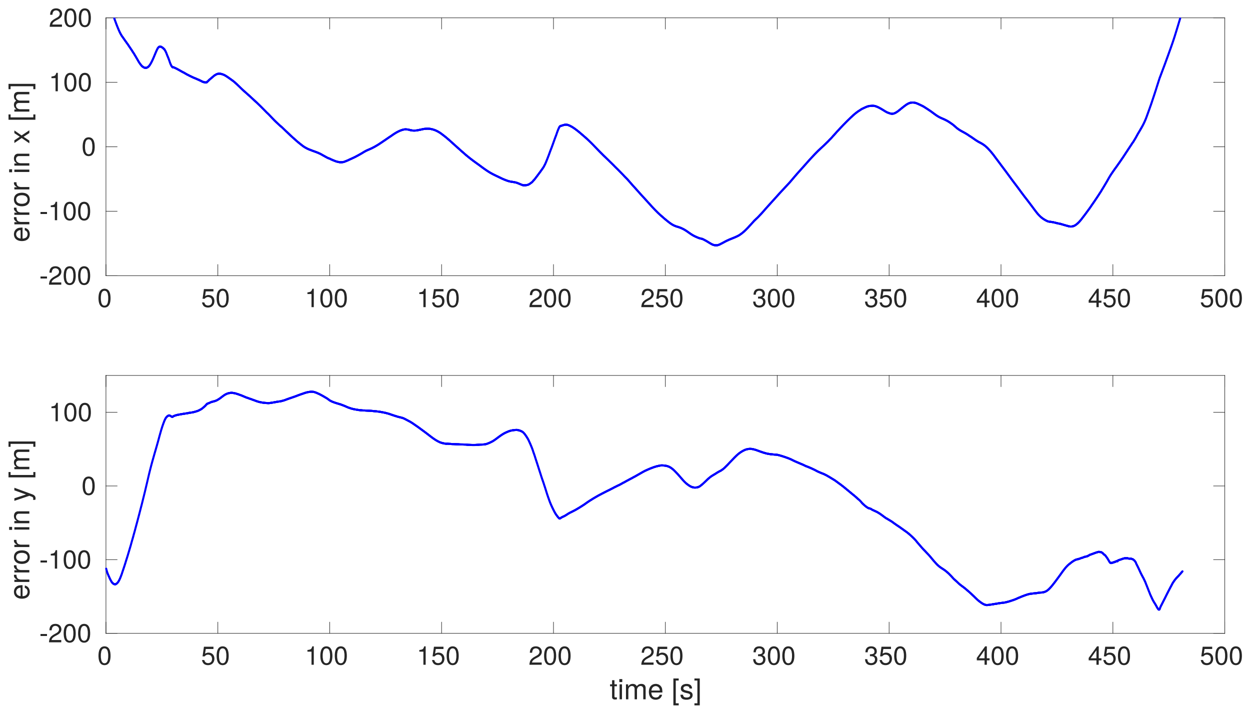


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.7: Estimated trajectories compared to ground-truth in the sequence 02.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.8: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 02.

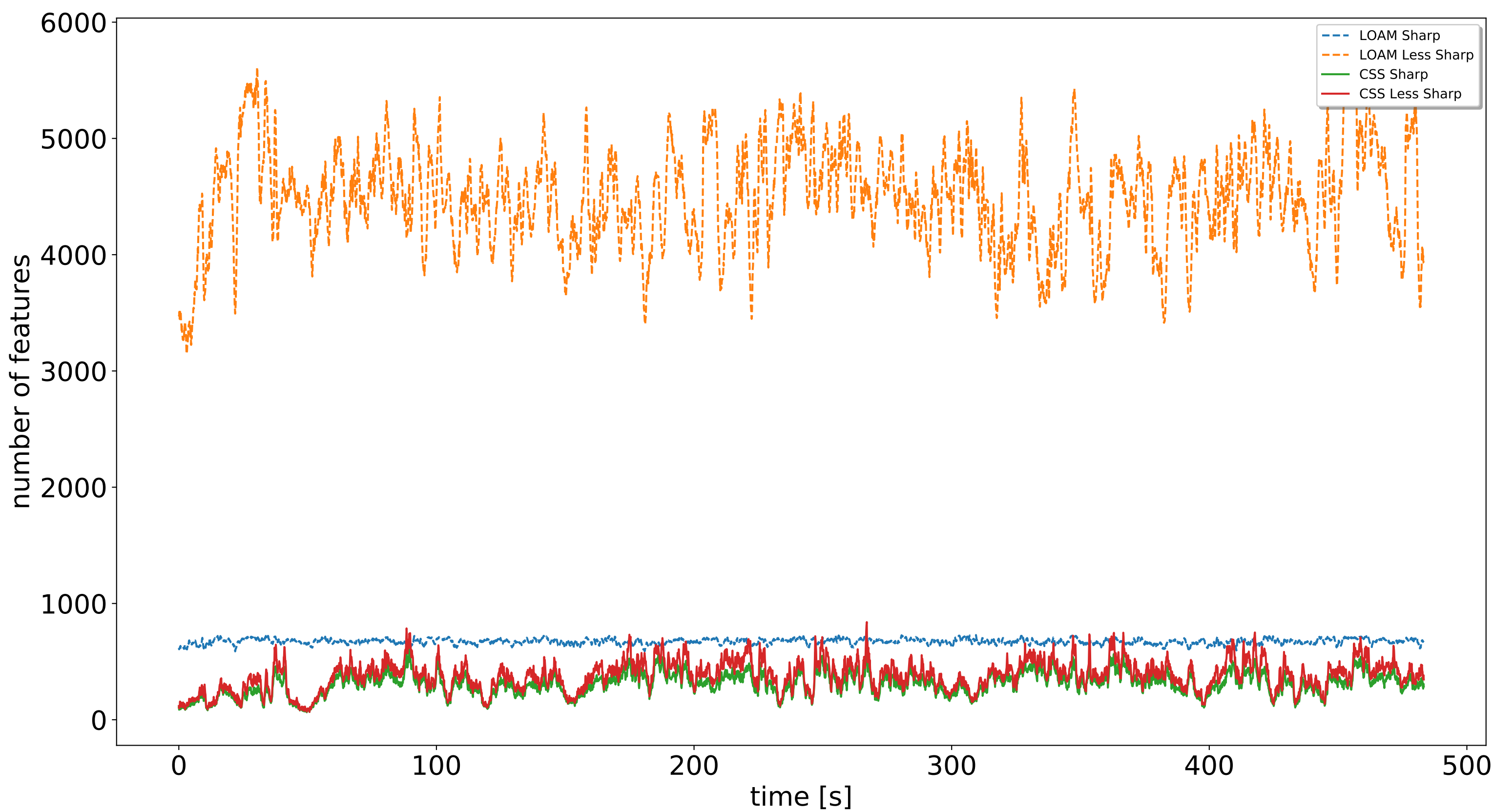
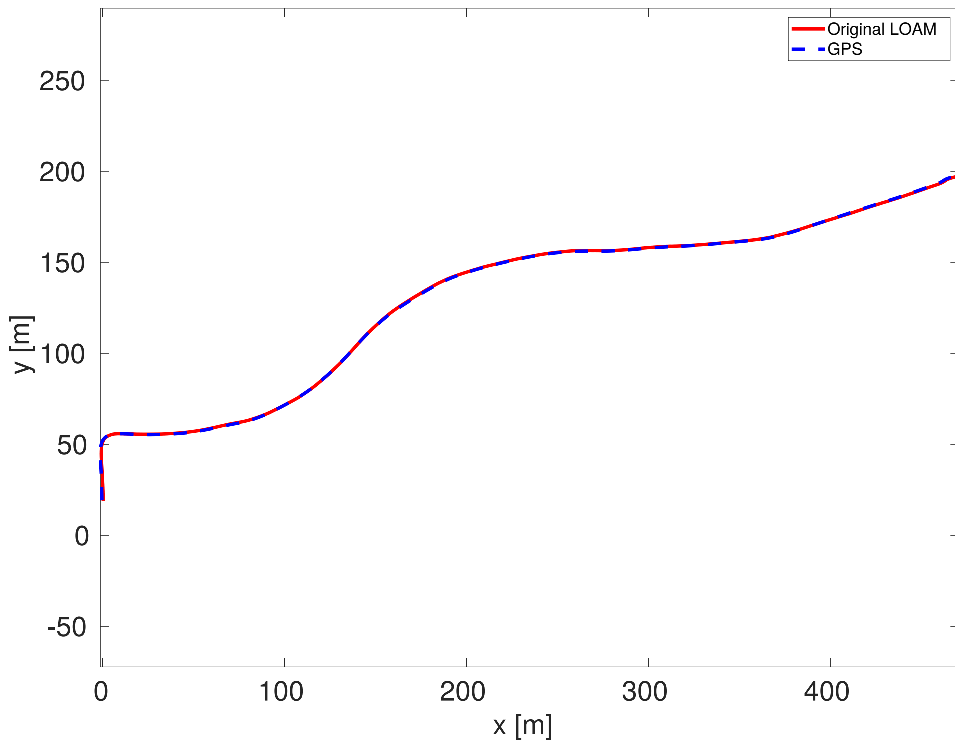
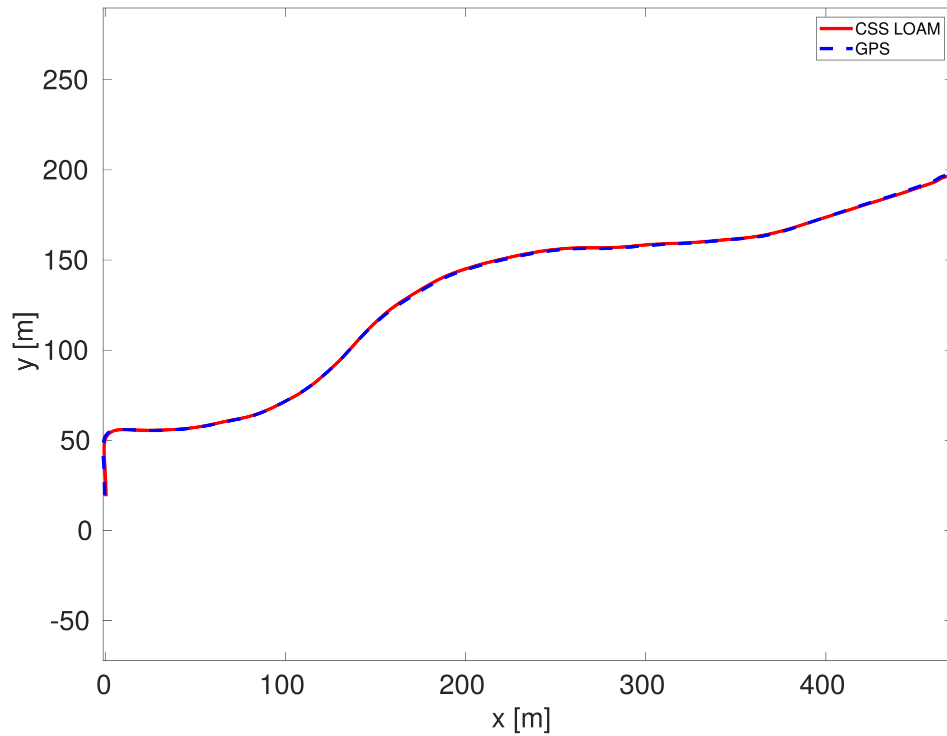


Figure A.9: Number of features in the sequence 02.

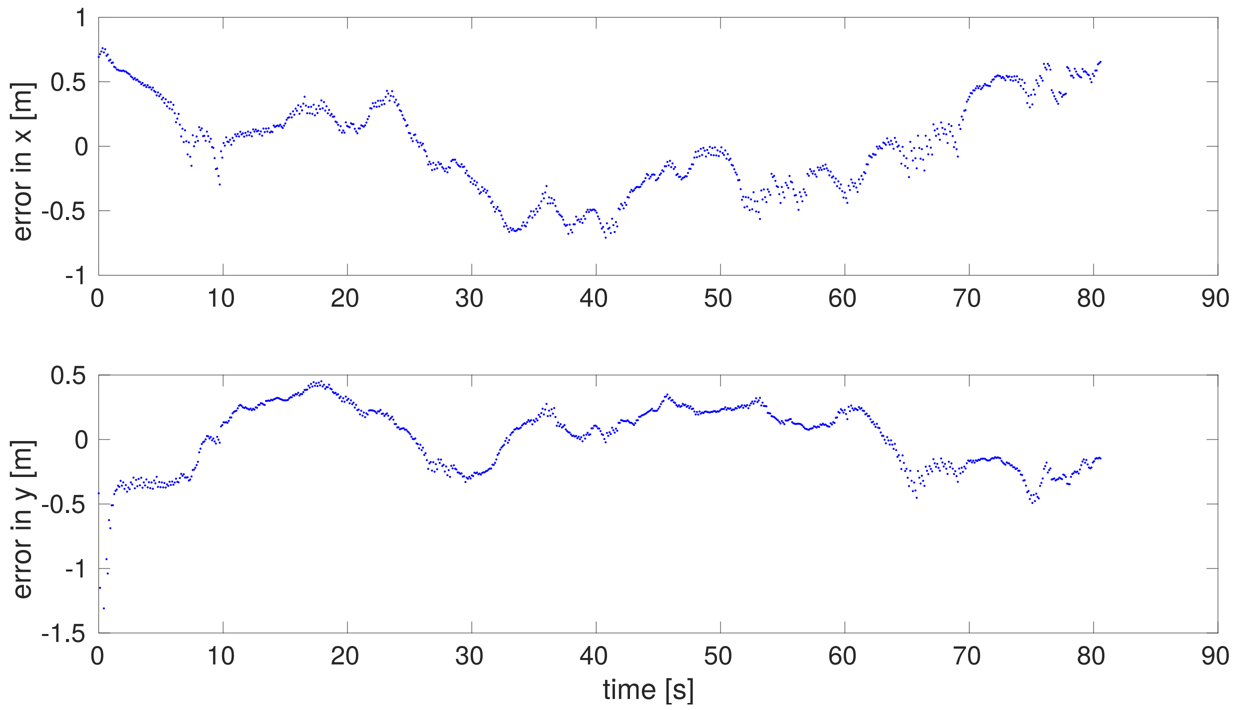


(a) Estimated trajectory using the original LOAM algorithm.

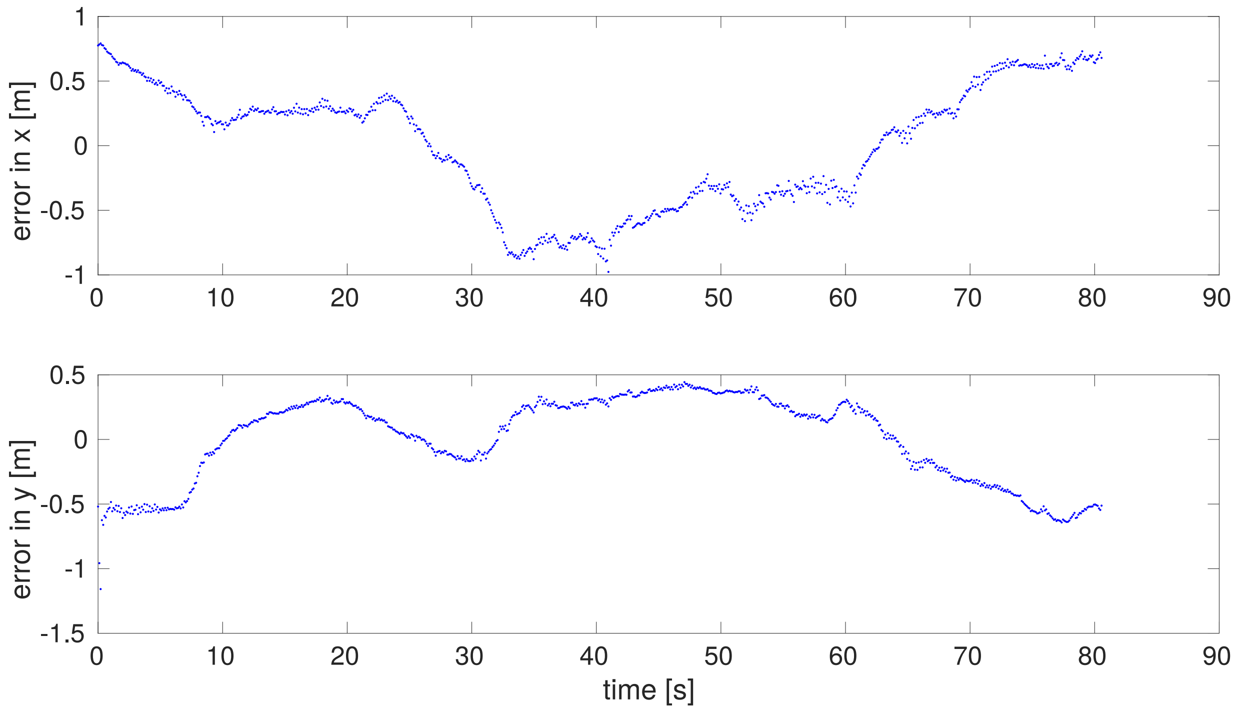


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.10: Estimated trajectories compared to ground-truth in the sequence 03.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.11: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 03.

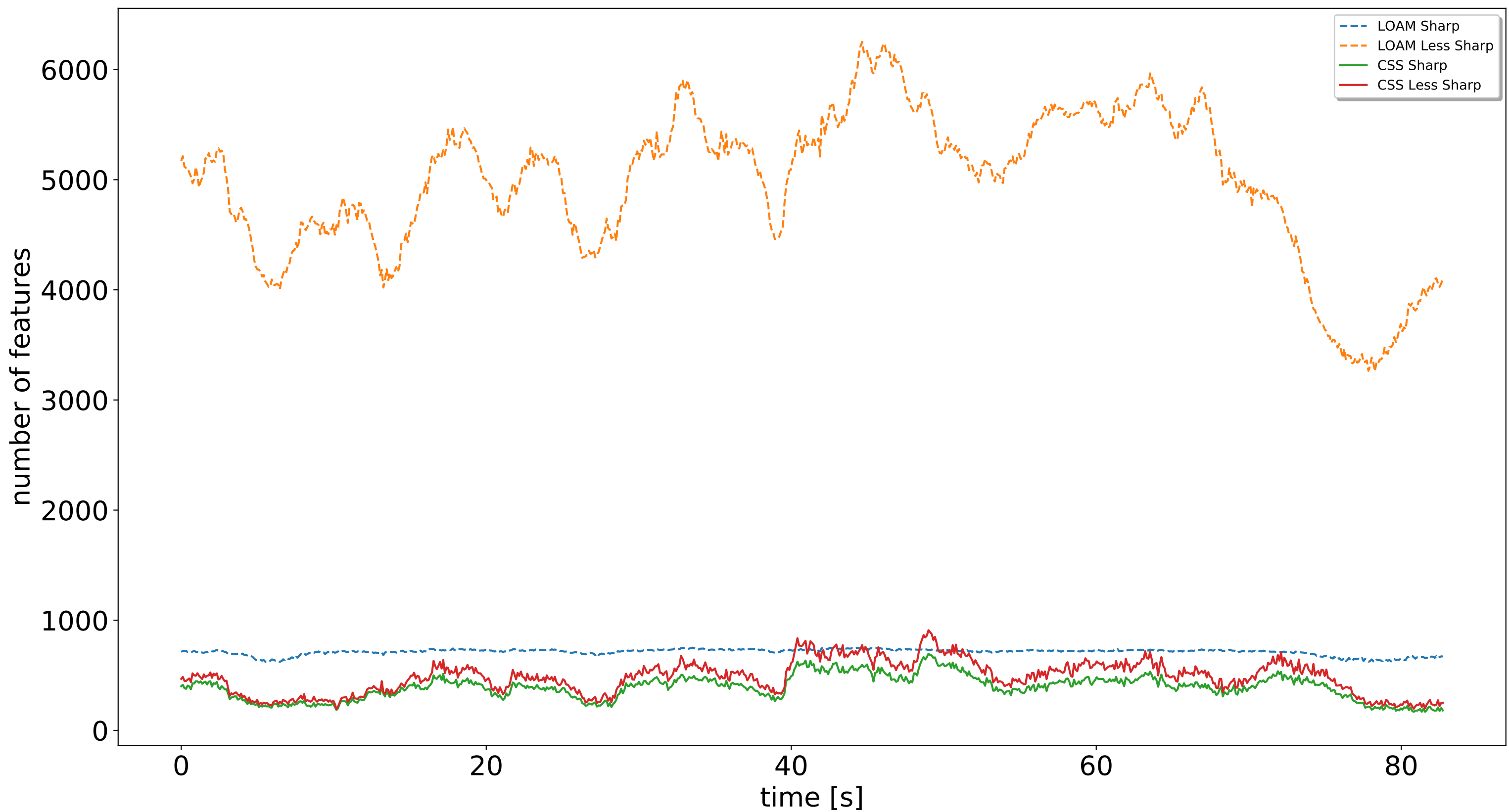
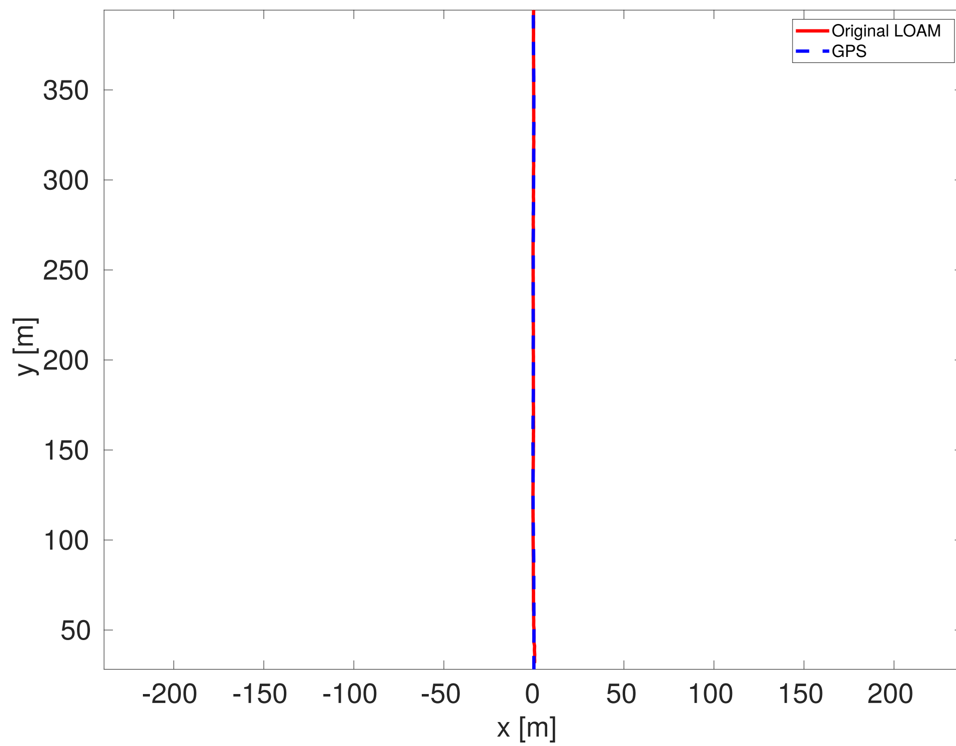
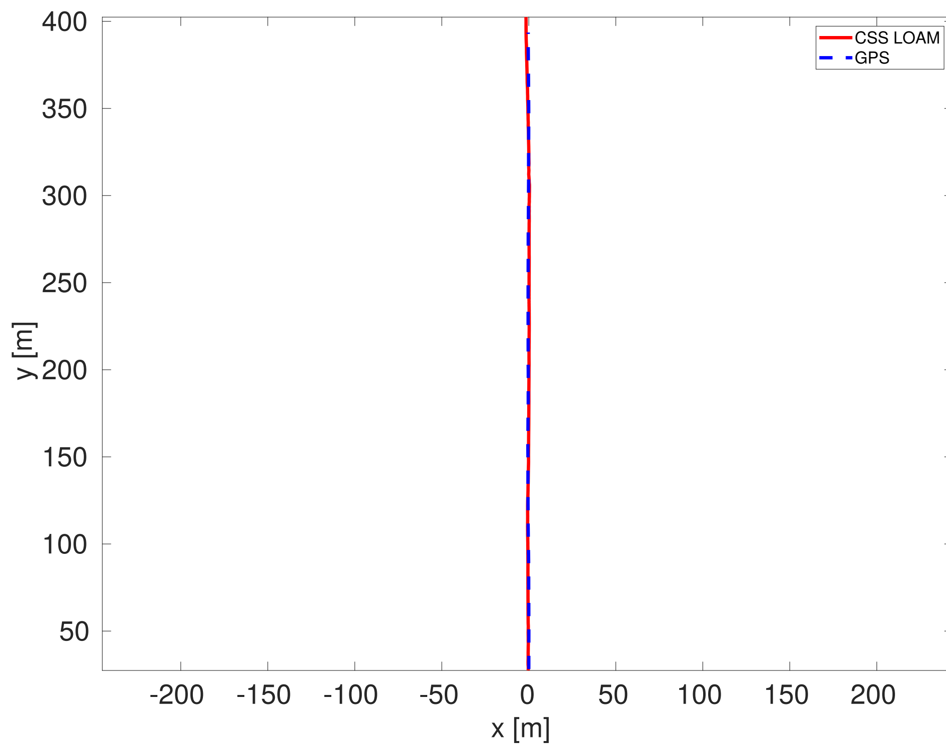


Figure A.12: Number of features in the sequence 03.

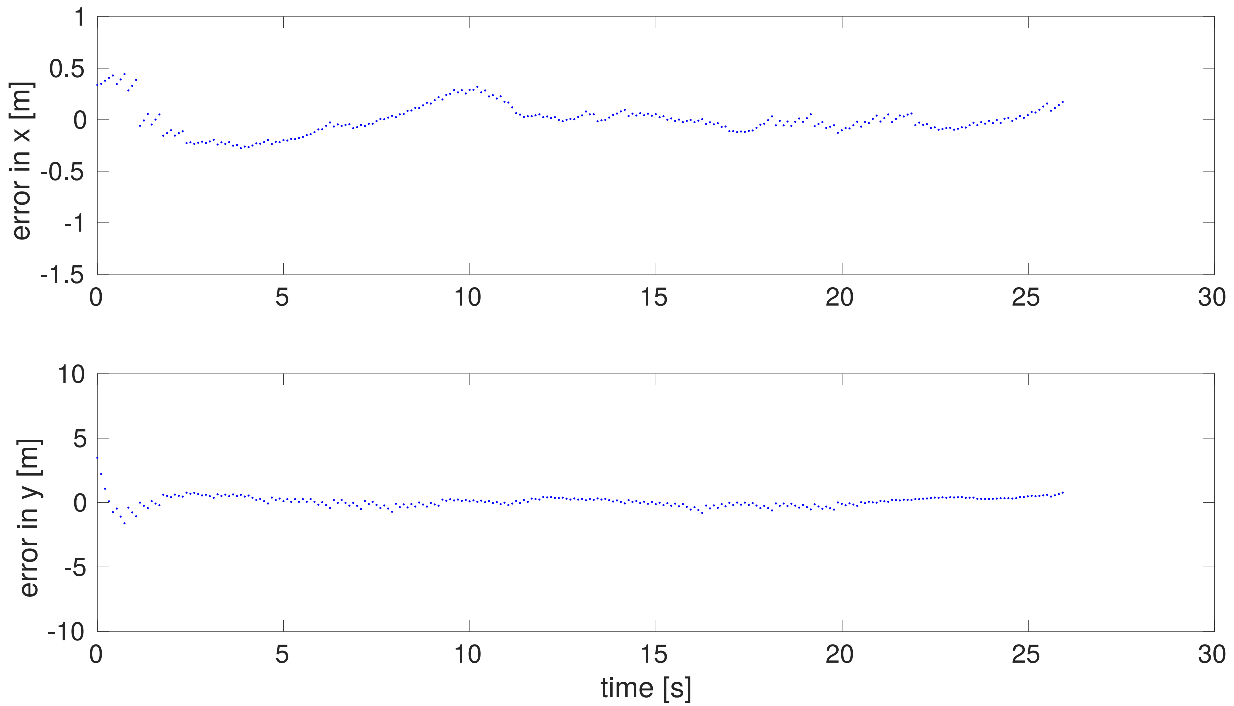


(a) Estimated trajectory using the original LOAM algorithm.

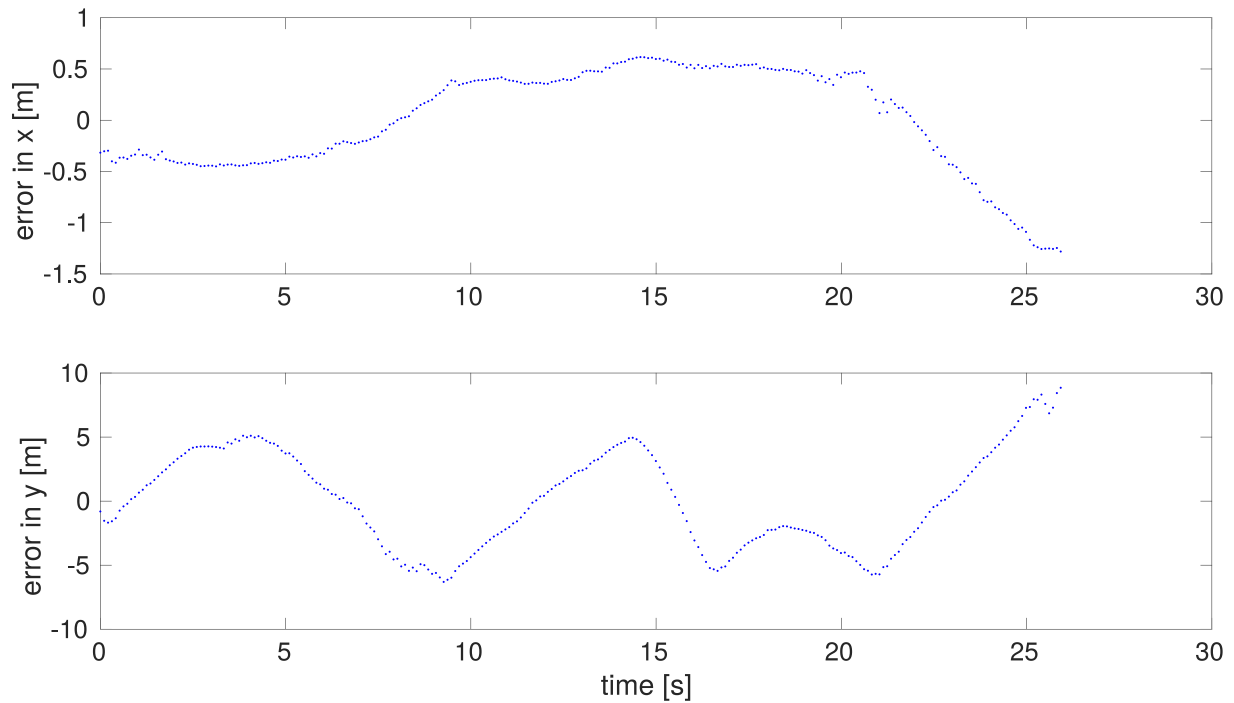


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.13: Estimated trajectories compared to ground-truth in the sequence 04.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.14: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 04.

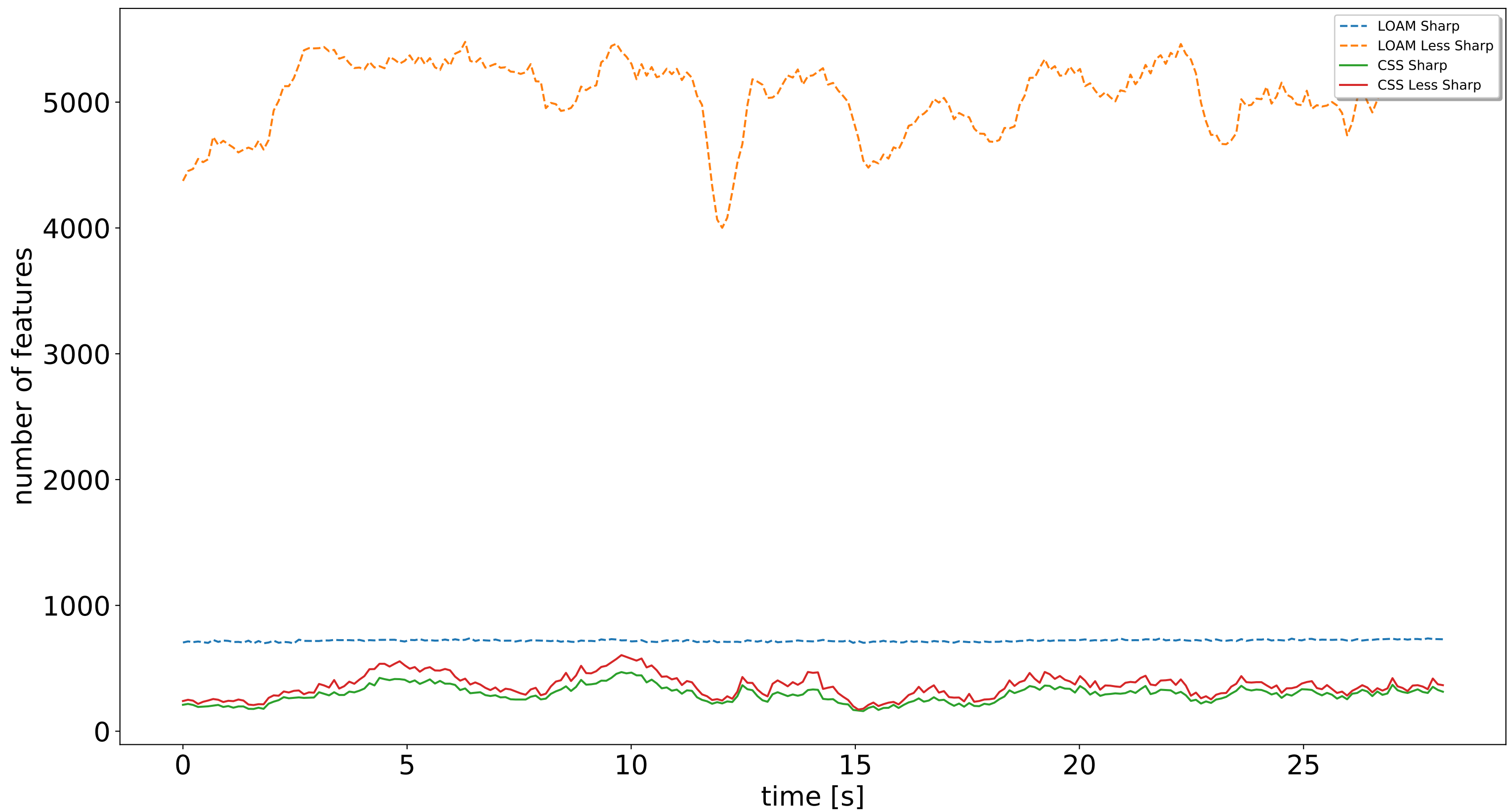
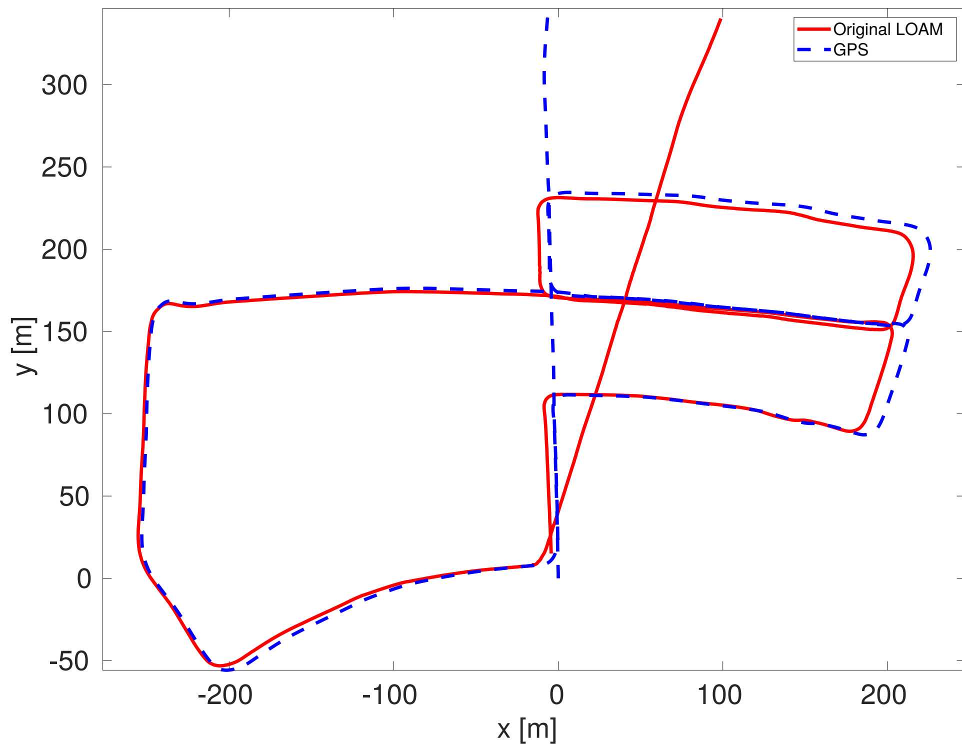
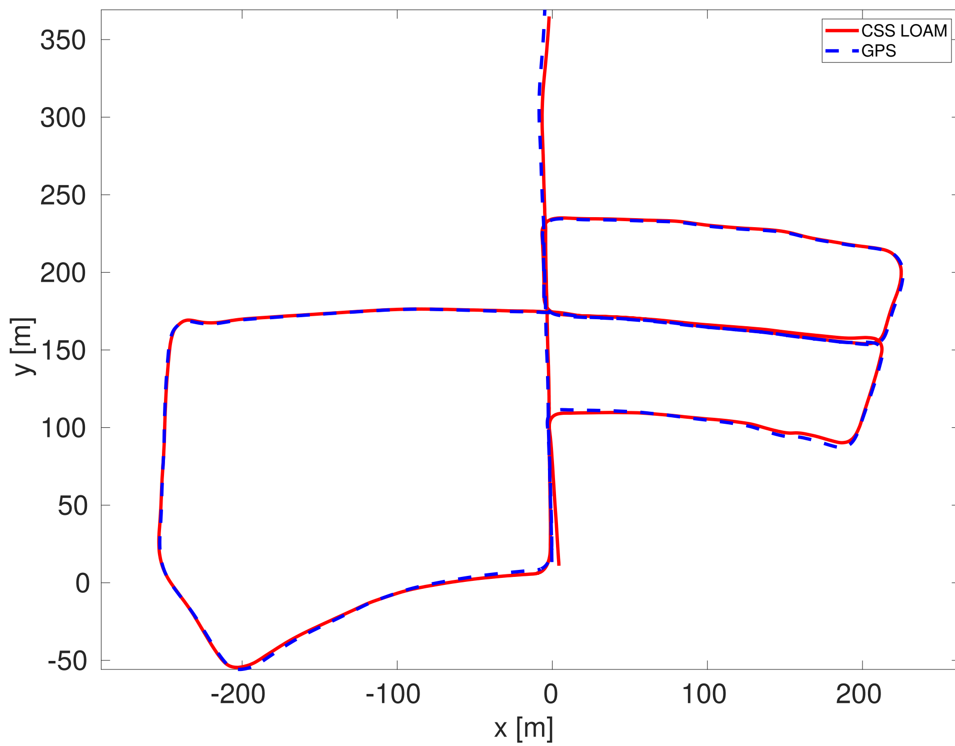


Figure A.15: Number of features in the sequence 04.

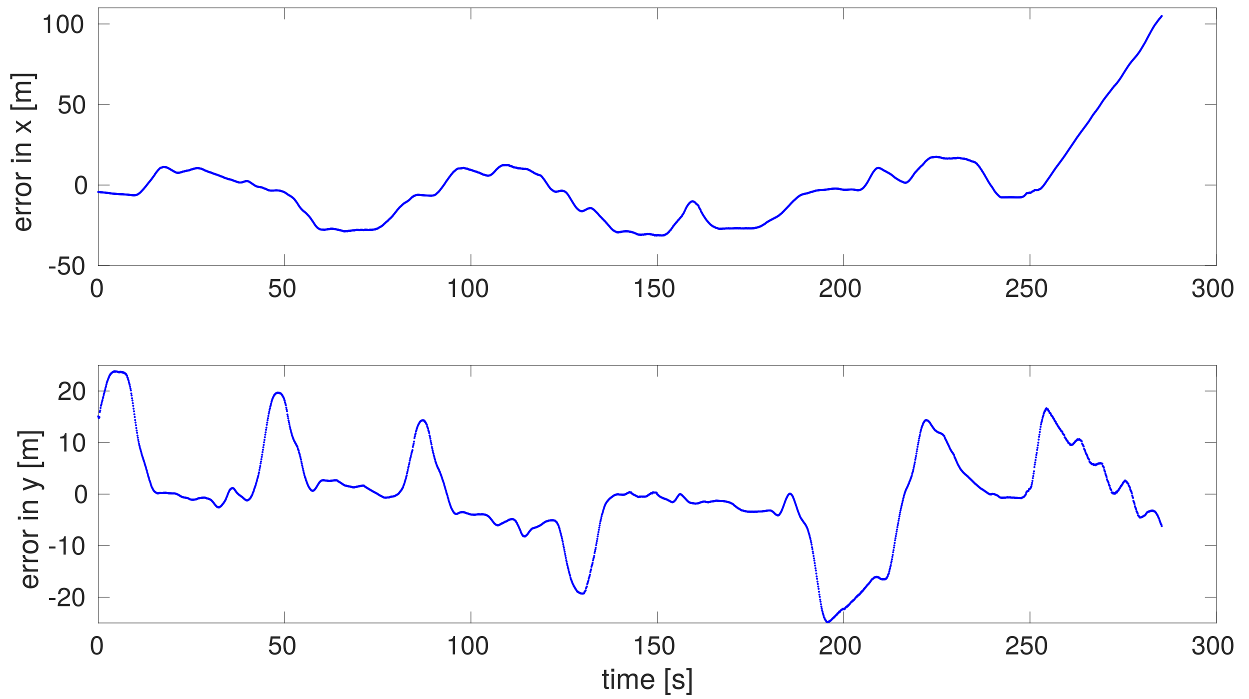


(a) Estimated trajectory using the original LOAM algorithm.

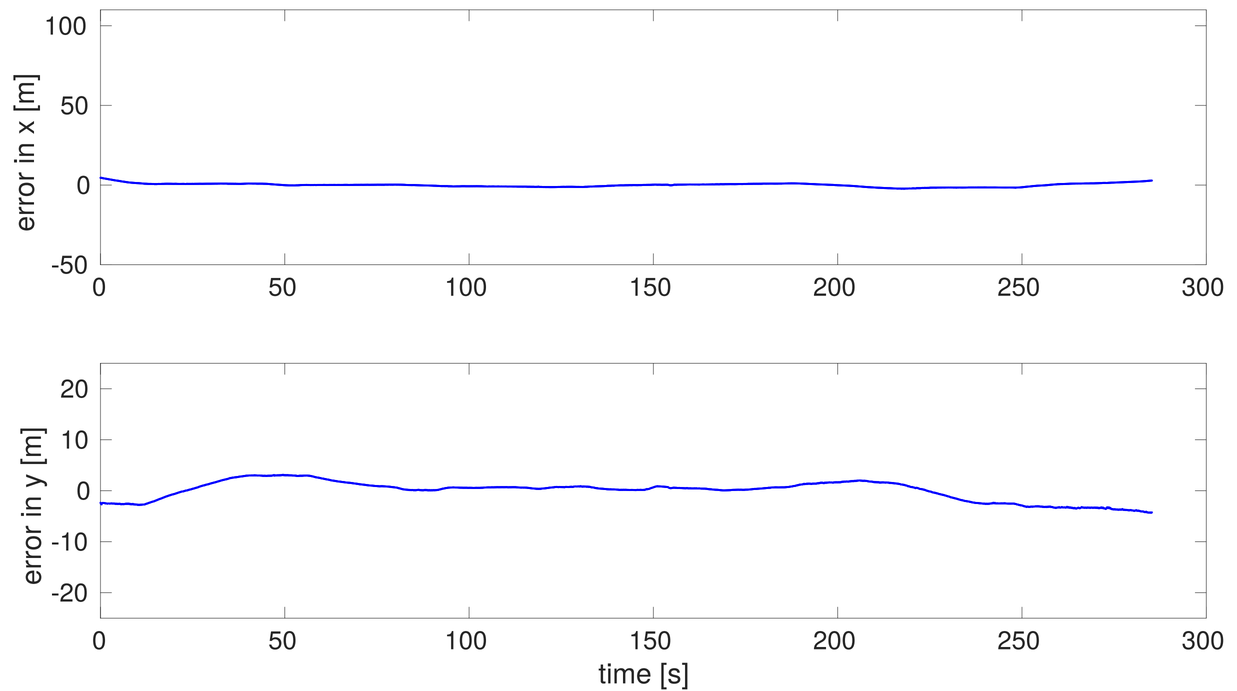


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.16: Estimated trajectories compared to ground-truth in the sequence 05.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.17: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 05.

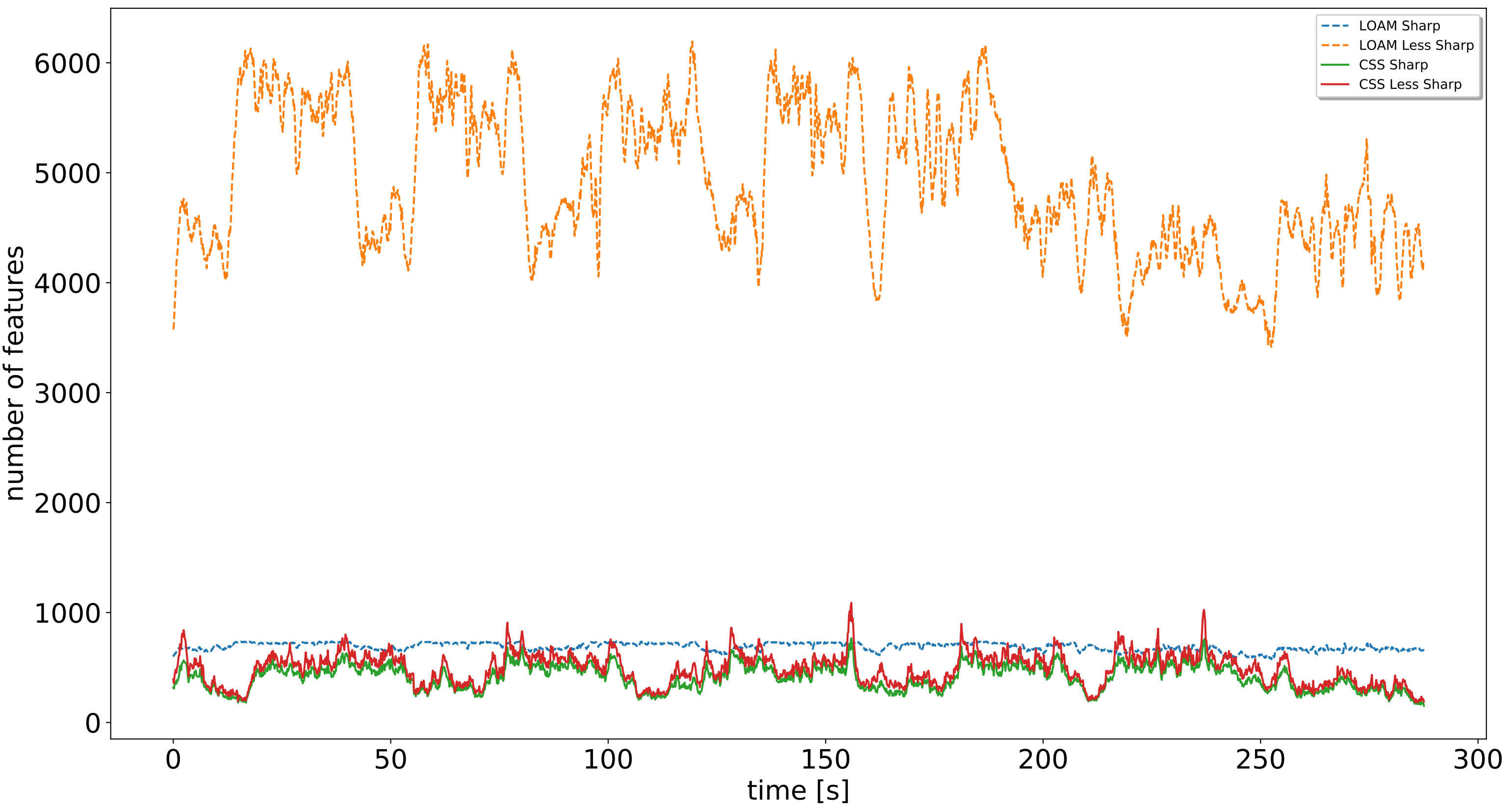
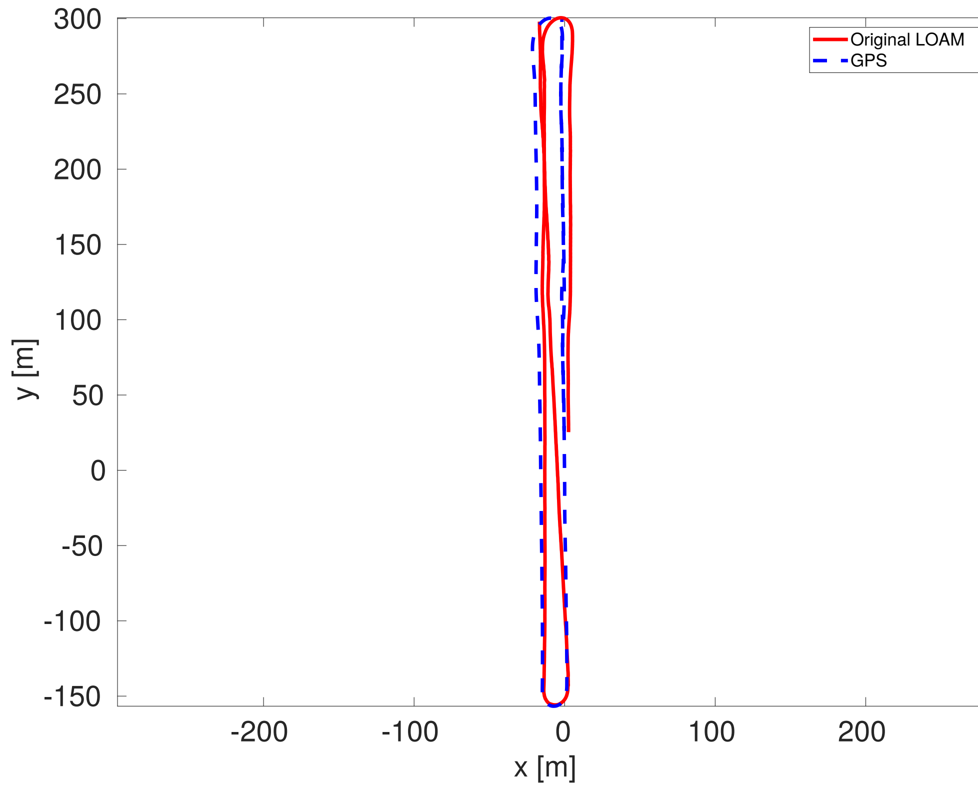
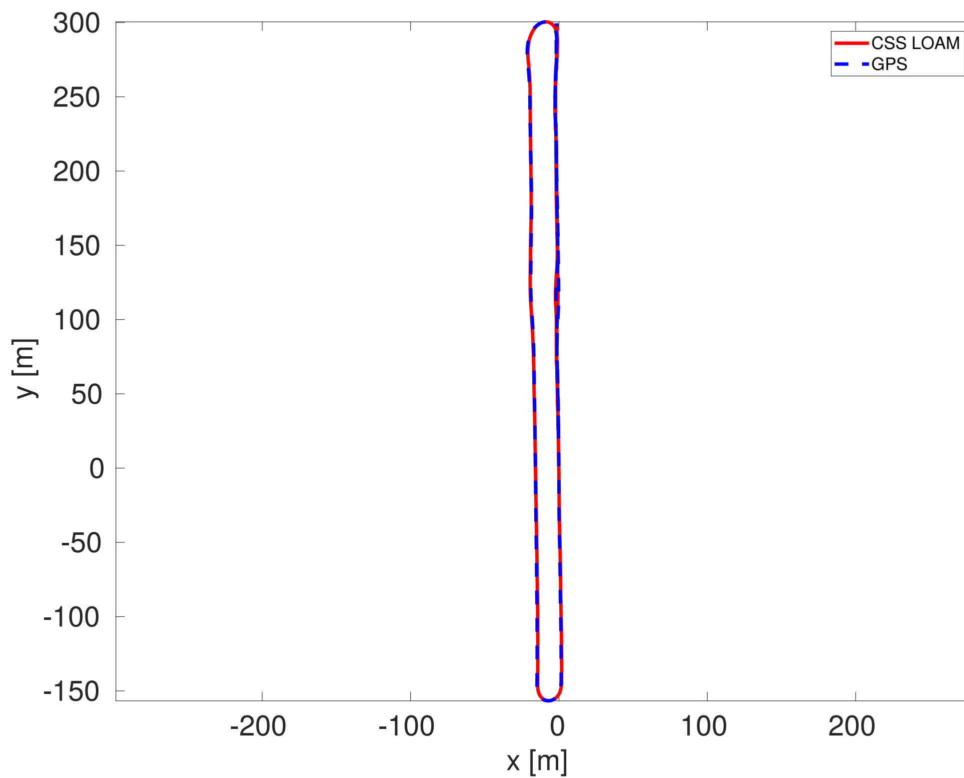


Figure A.18: Number of features in the sequence 05.

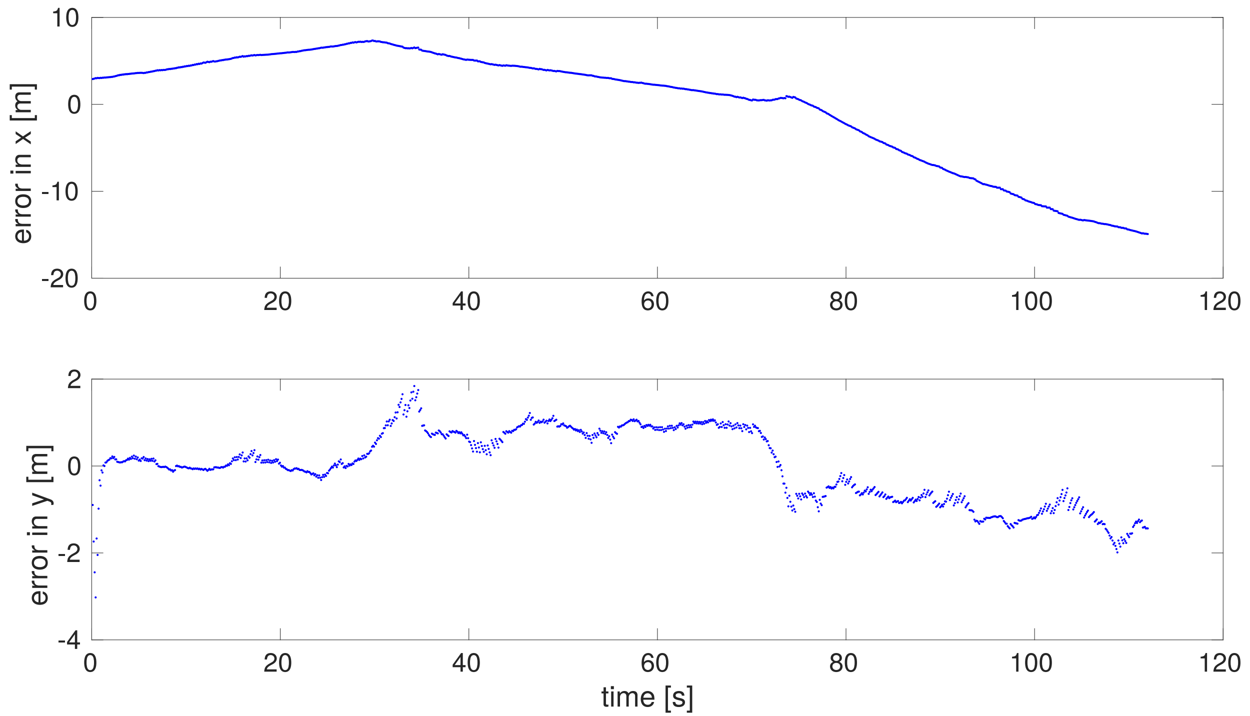


(a) Estimated trajectory using the original LOAM algorithm.

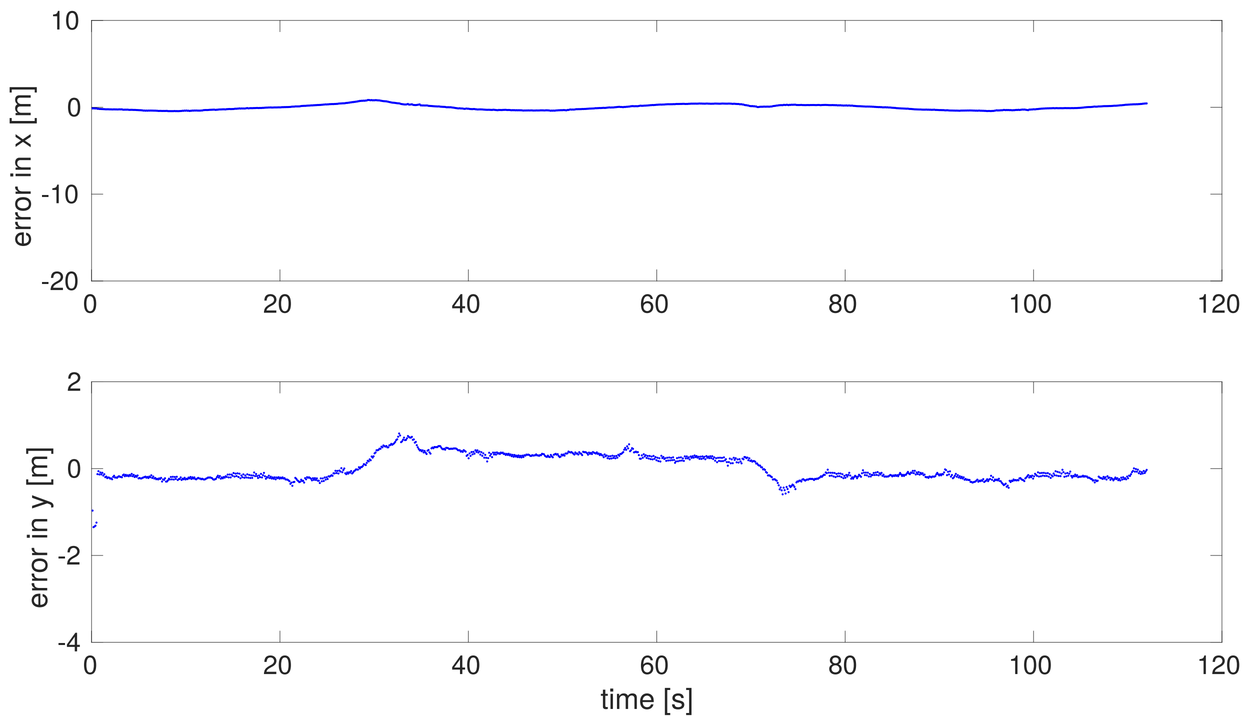


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.19: Estimated trajectories compared to ground-truth in the sequence 06.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.20: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 06.

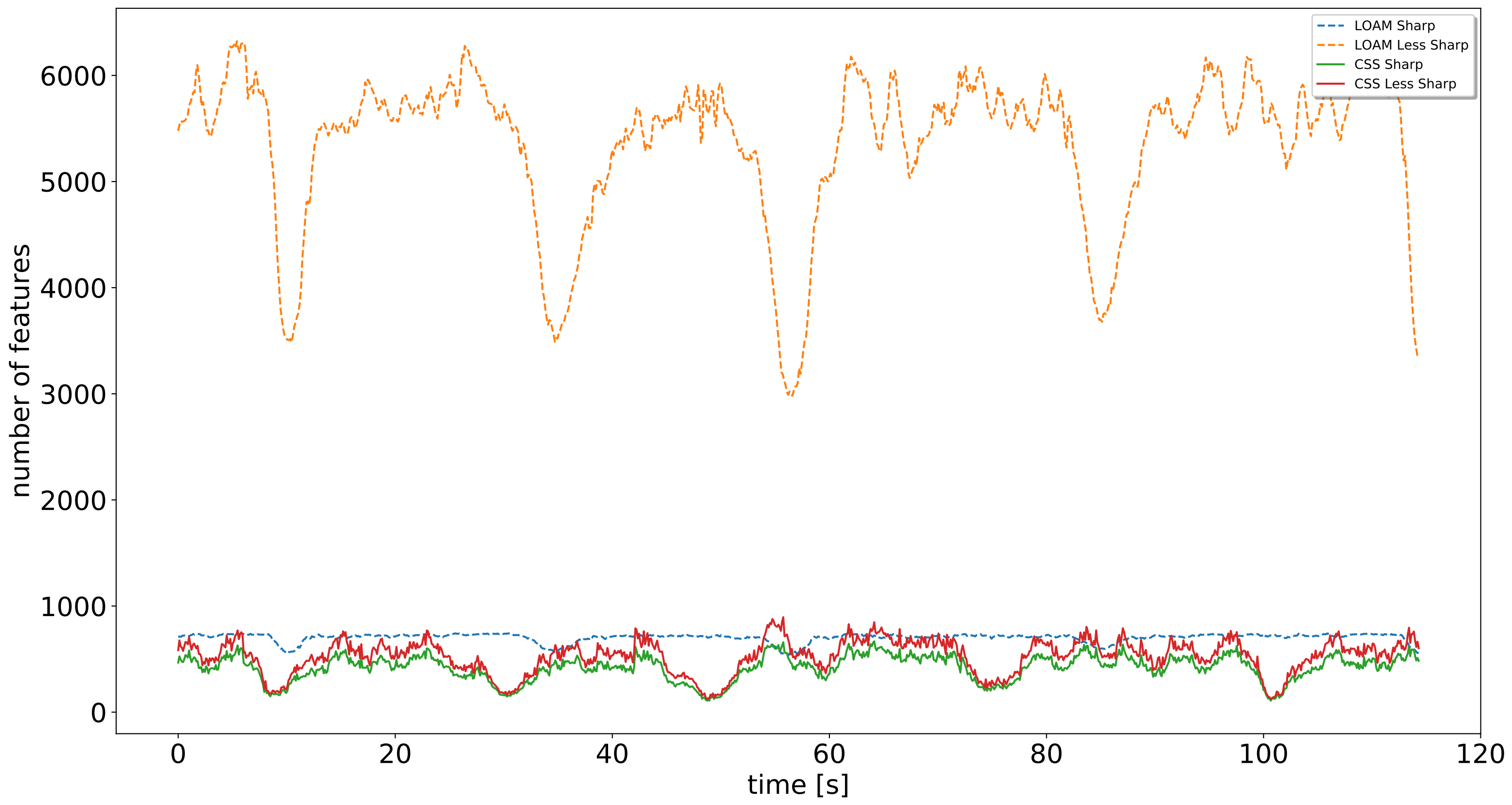
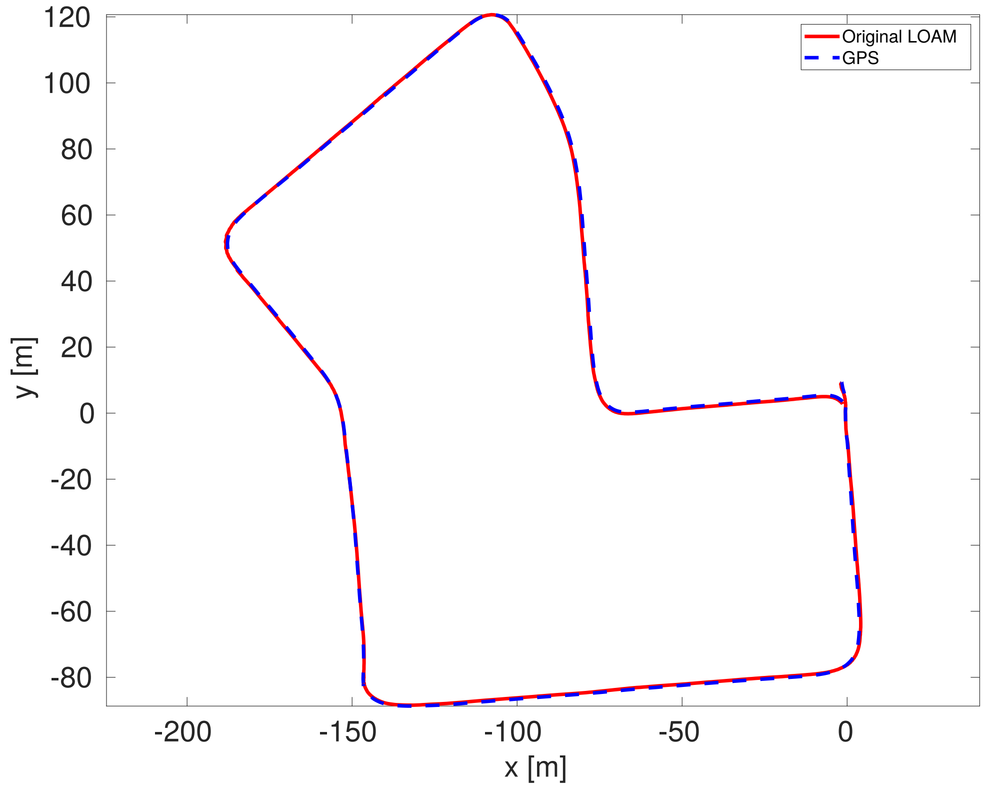
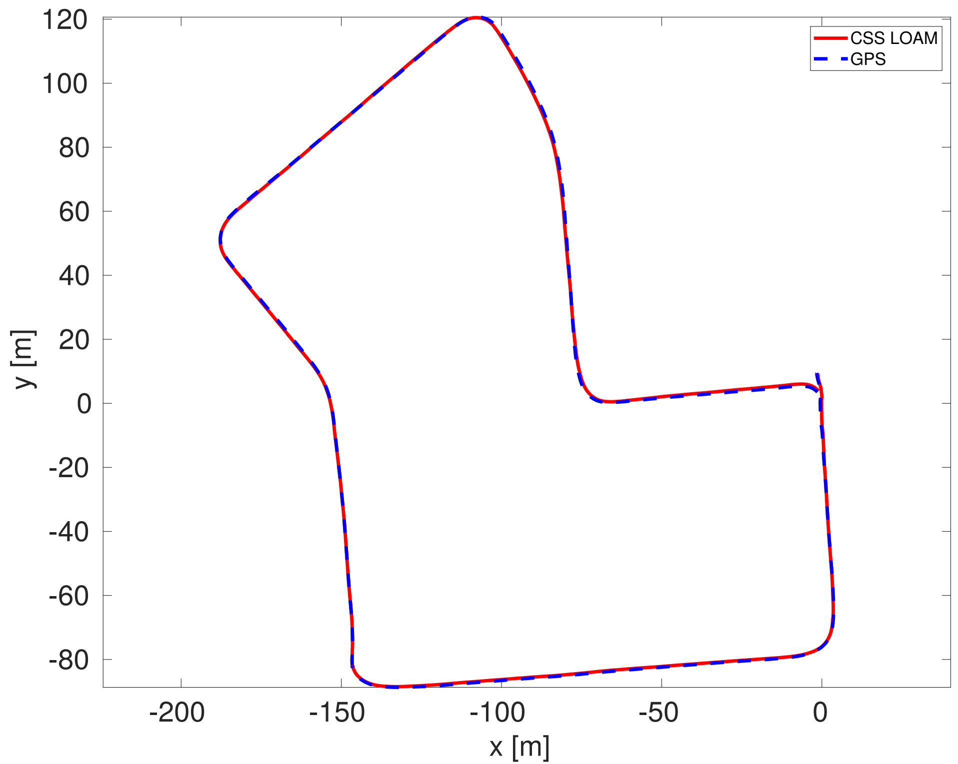


Figure A.21: Number of features in the sequence 06.

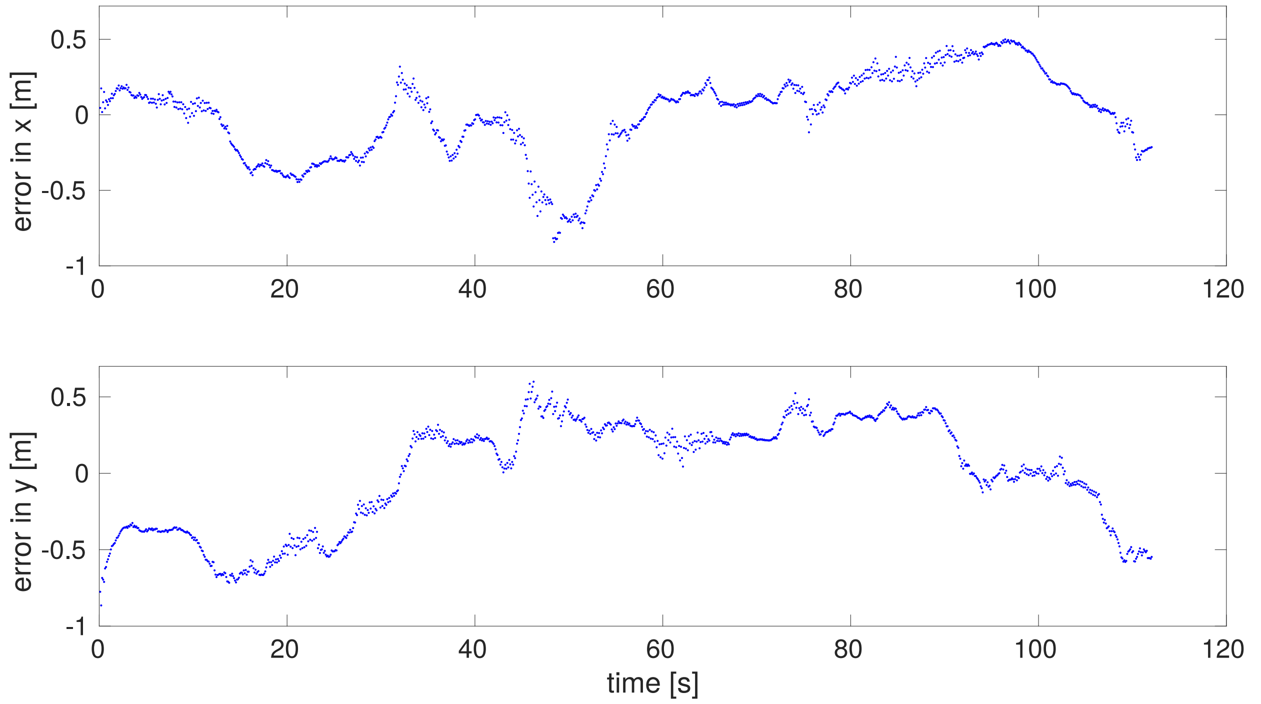


(a) Estimated trajectory using the original LOAM algorithm.

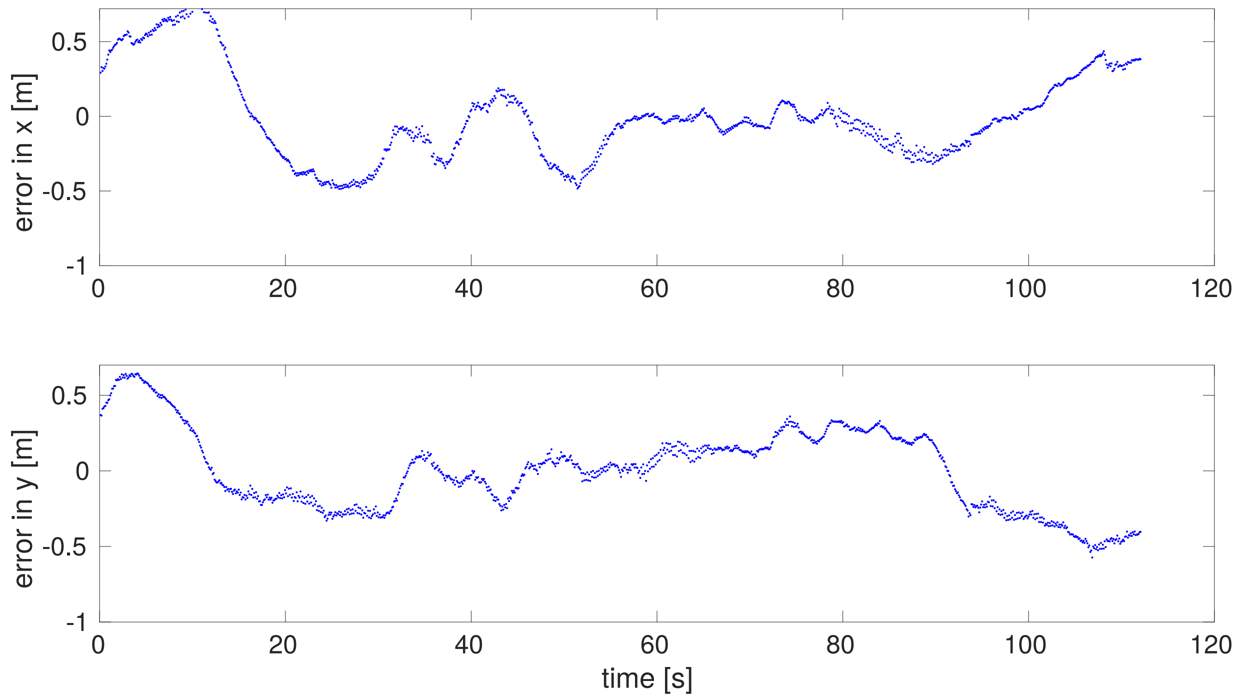


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.22: Estimated trajectories compared to ground-truth in the sequence 07.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.23: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 07.

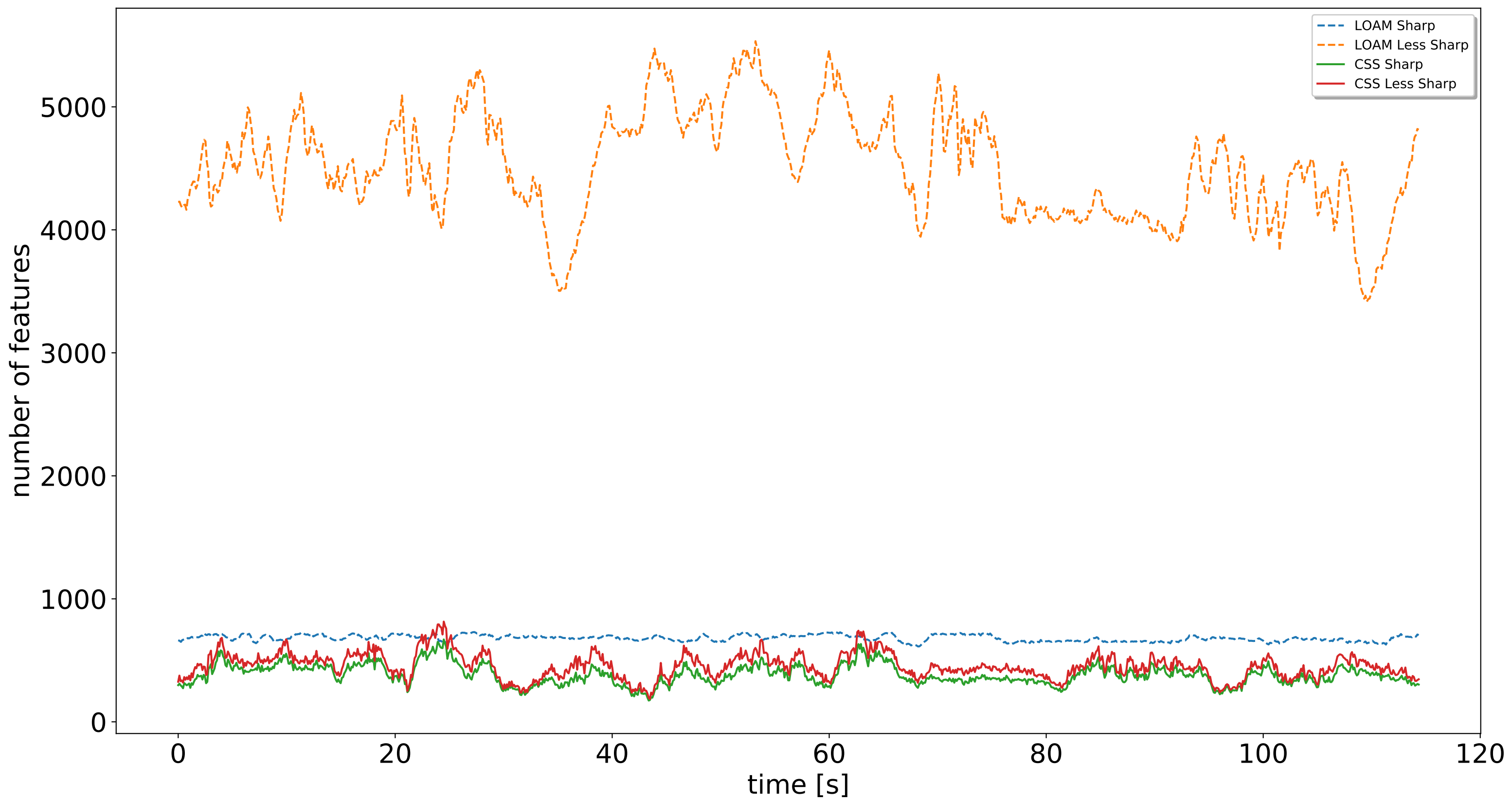
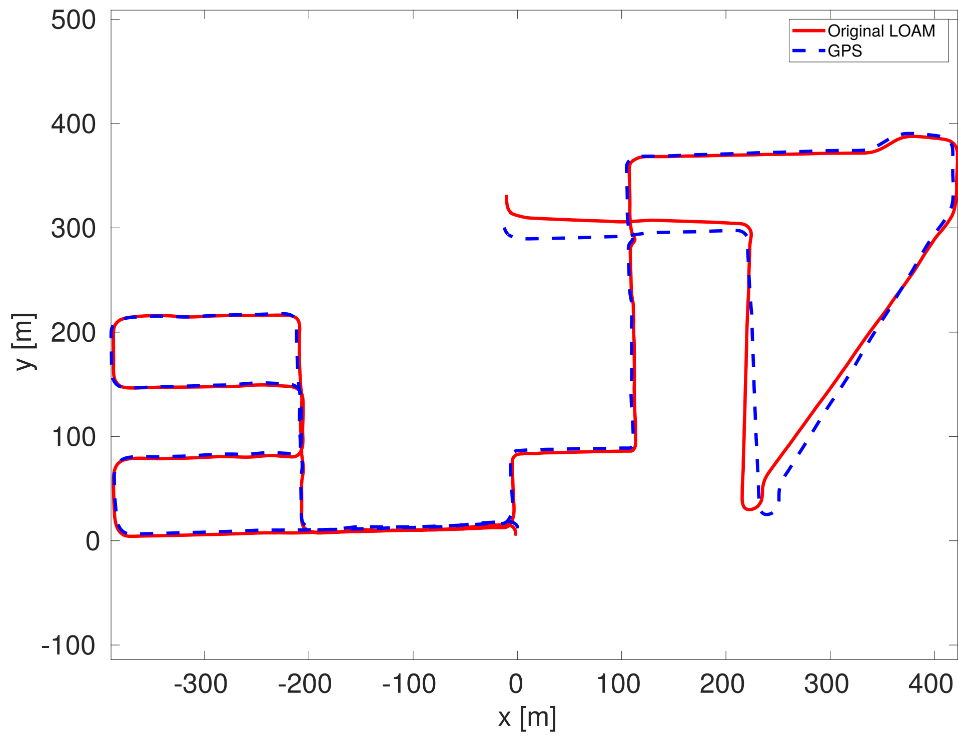
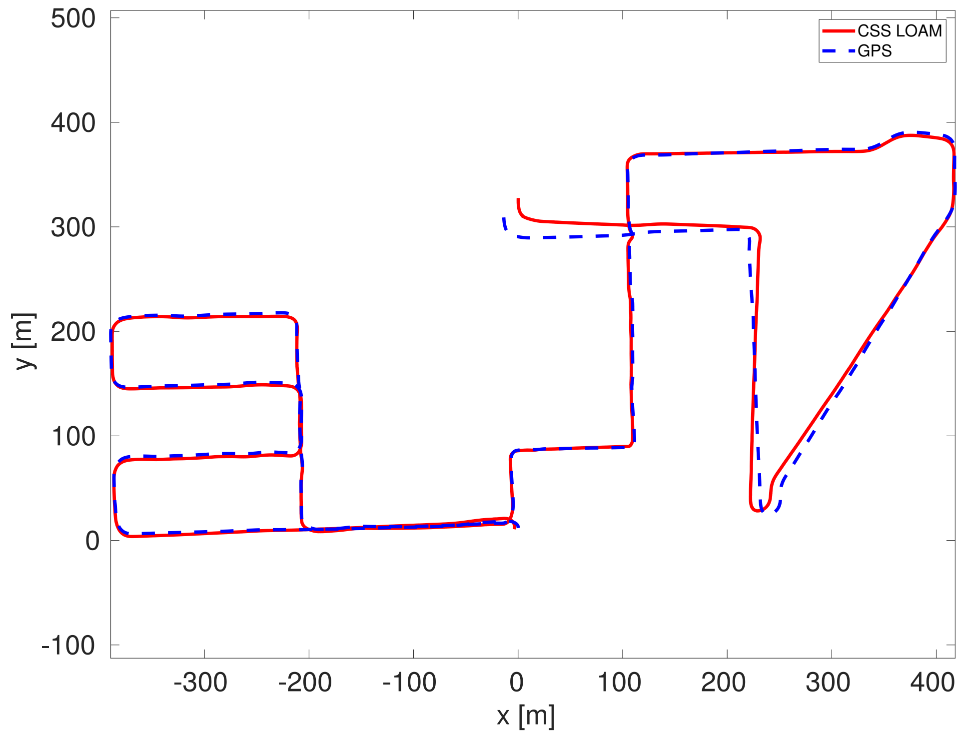


Figure A.24: Number of features in the sequence 07.

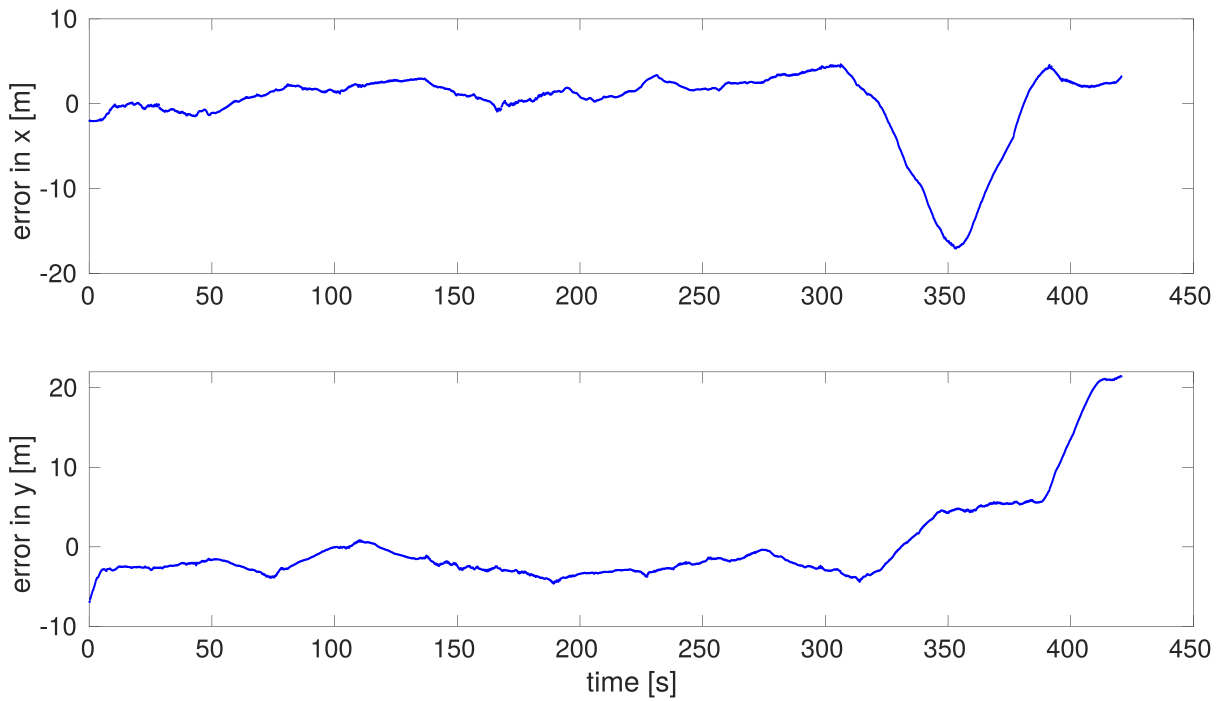


(a) Estimated trajectory using the original LOAM algorithm.

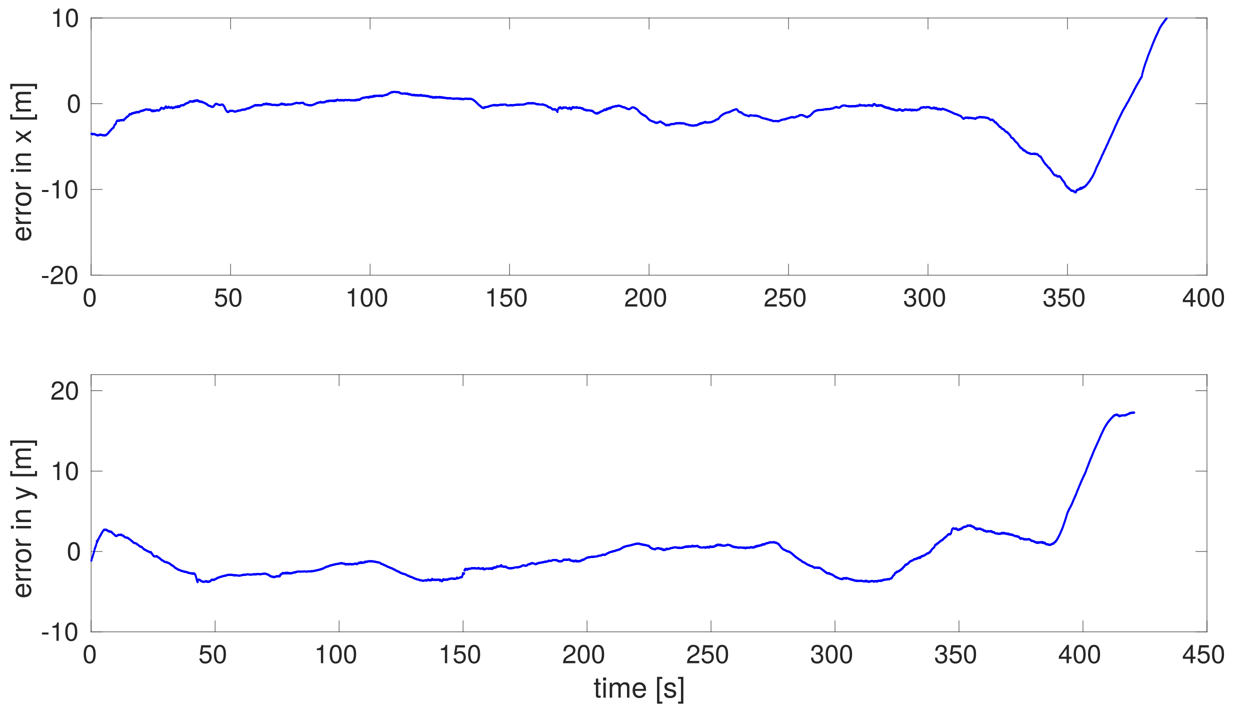


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.25: Estimated trajectories compared to ground-truth in the sequence 08.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.26: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 08.

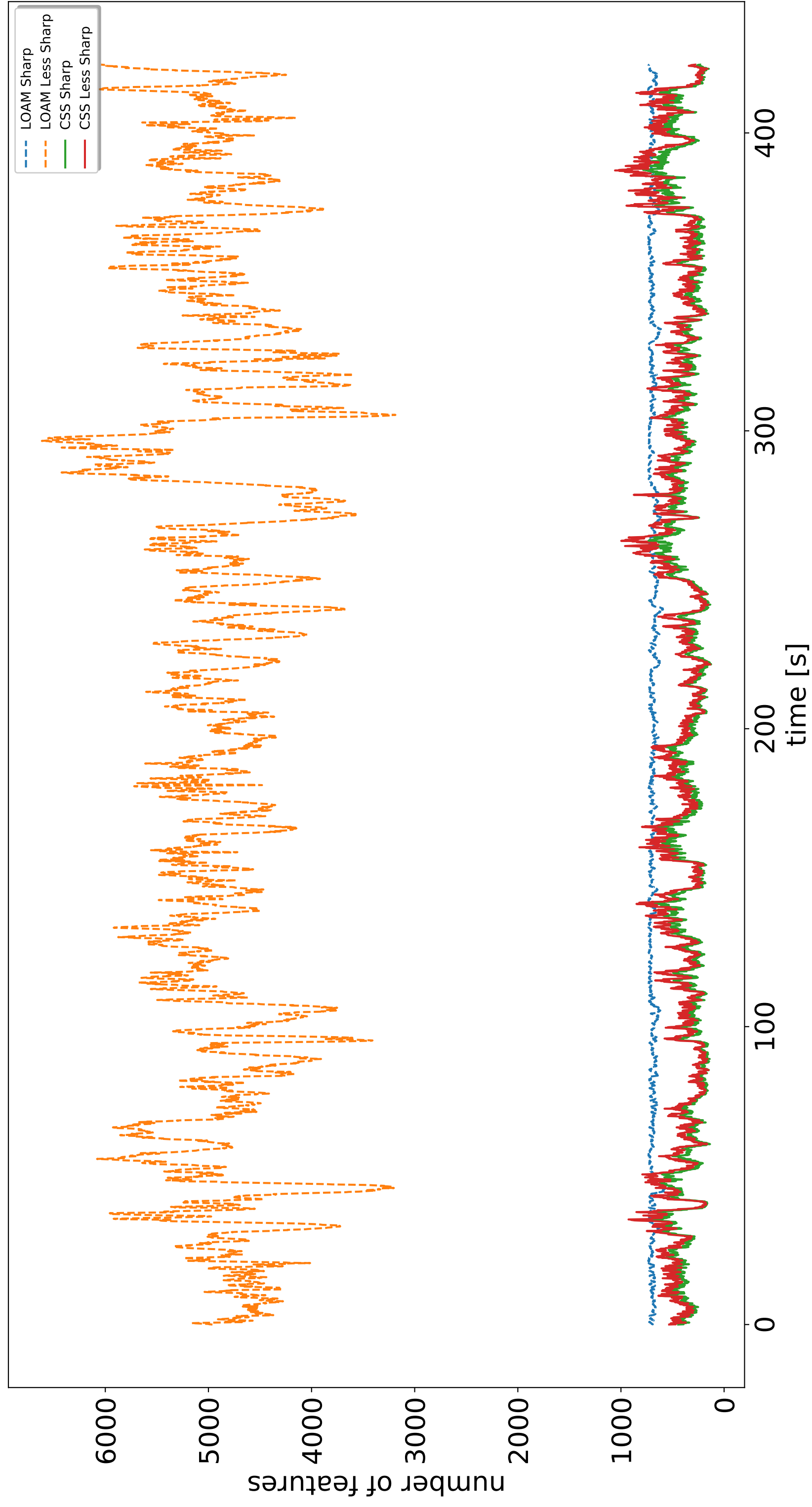
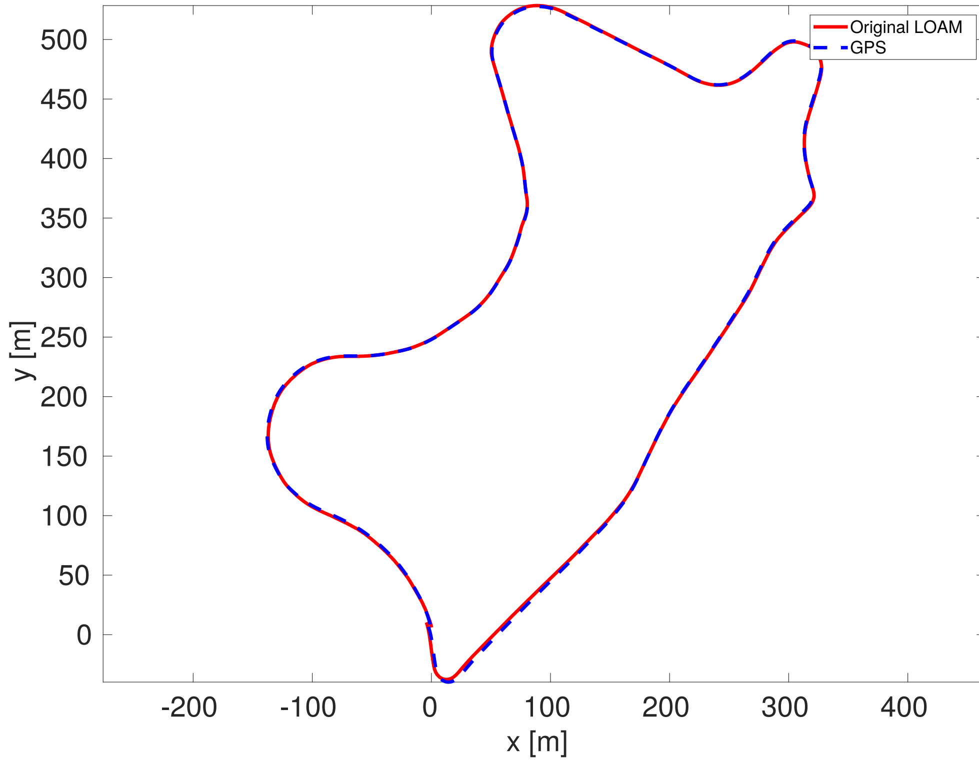
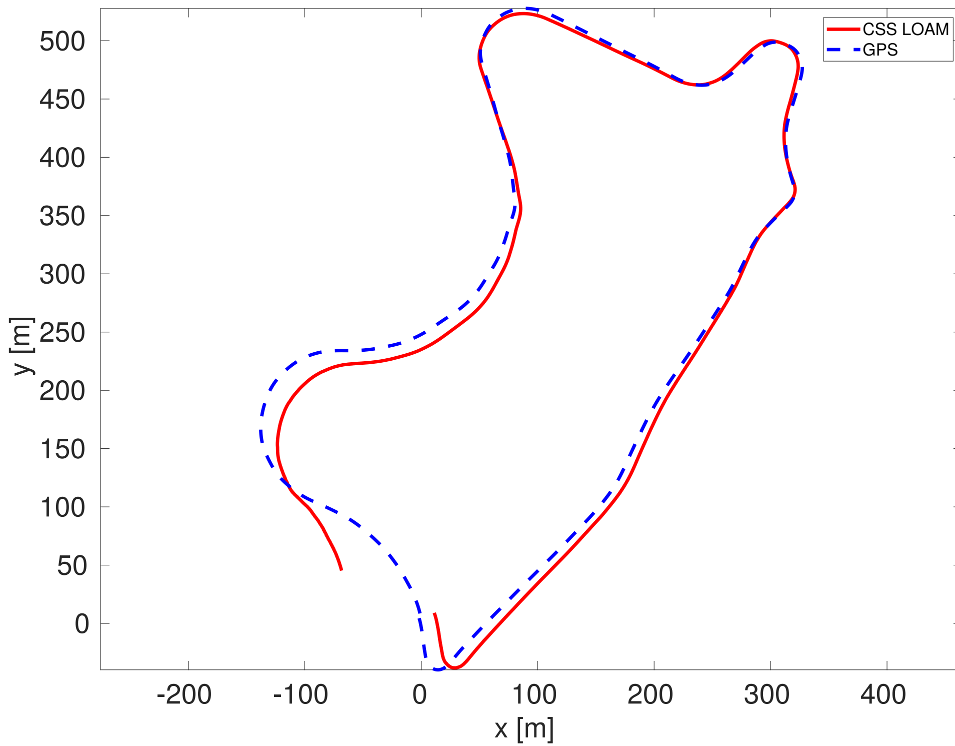


Figure A.27: Number of features in the sequence 08.

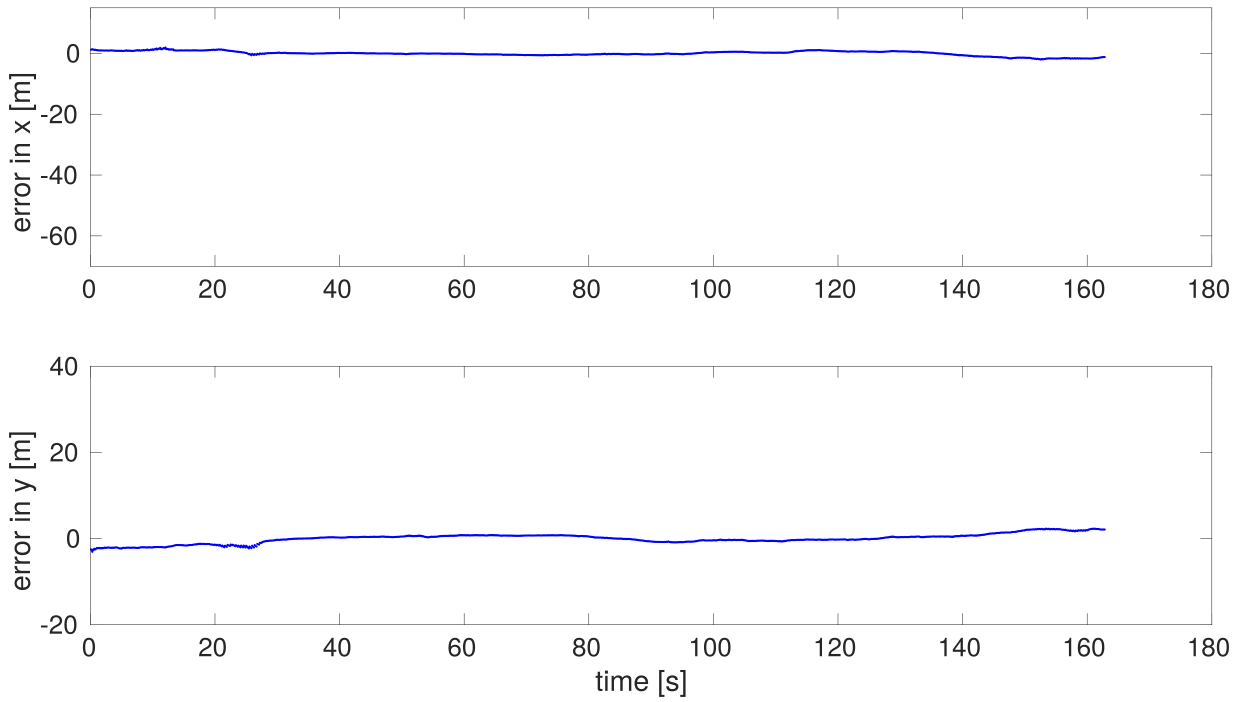


(a) Estimated trajectory using the original LOAM algorithm.

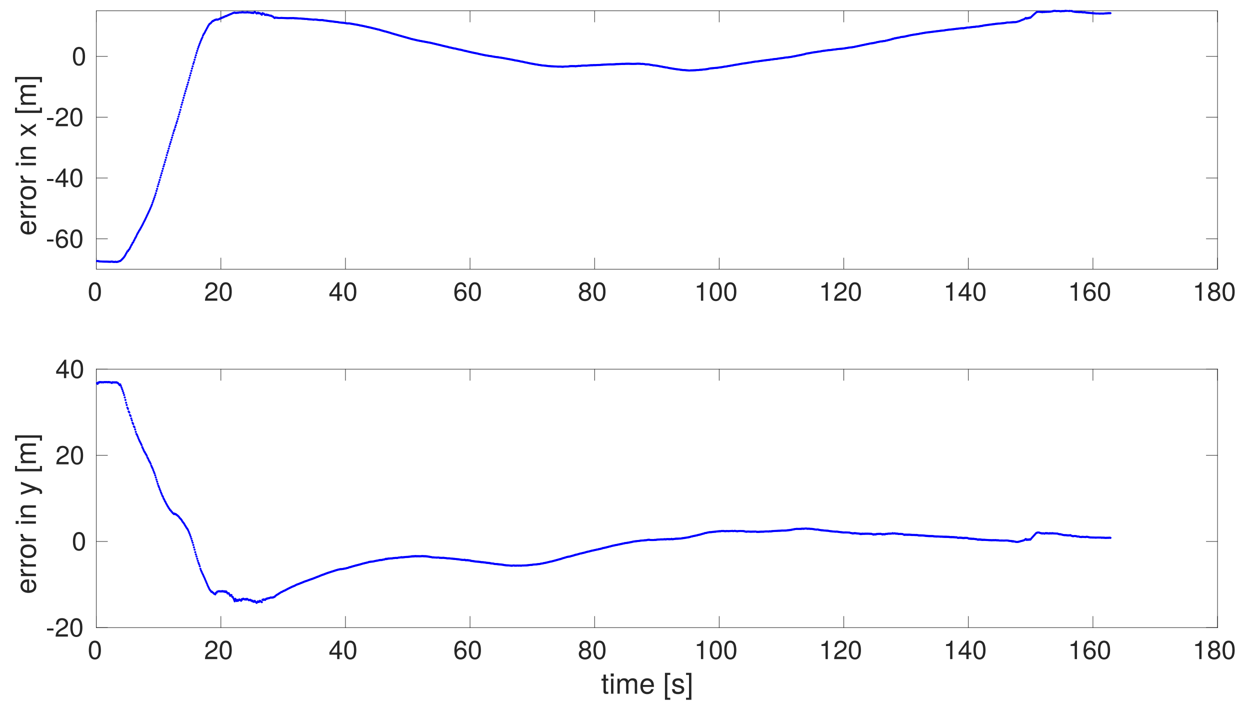


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.28: Estimated trajectories compared to ground-truth in the sequence 09.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.29: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 09.

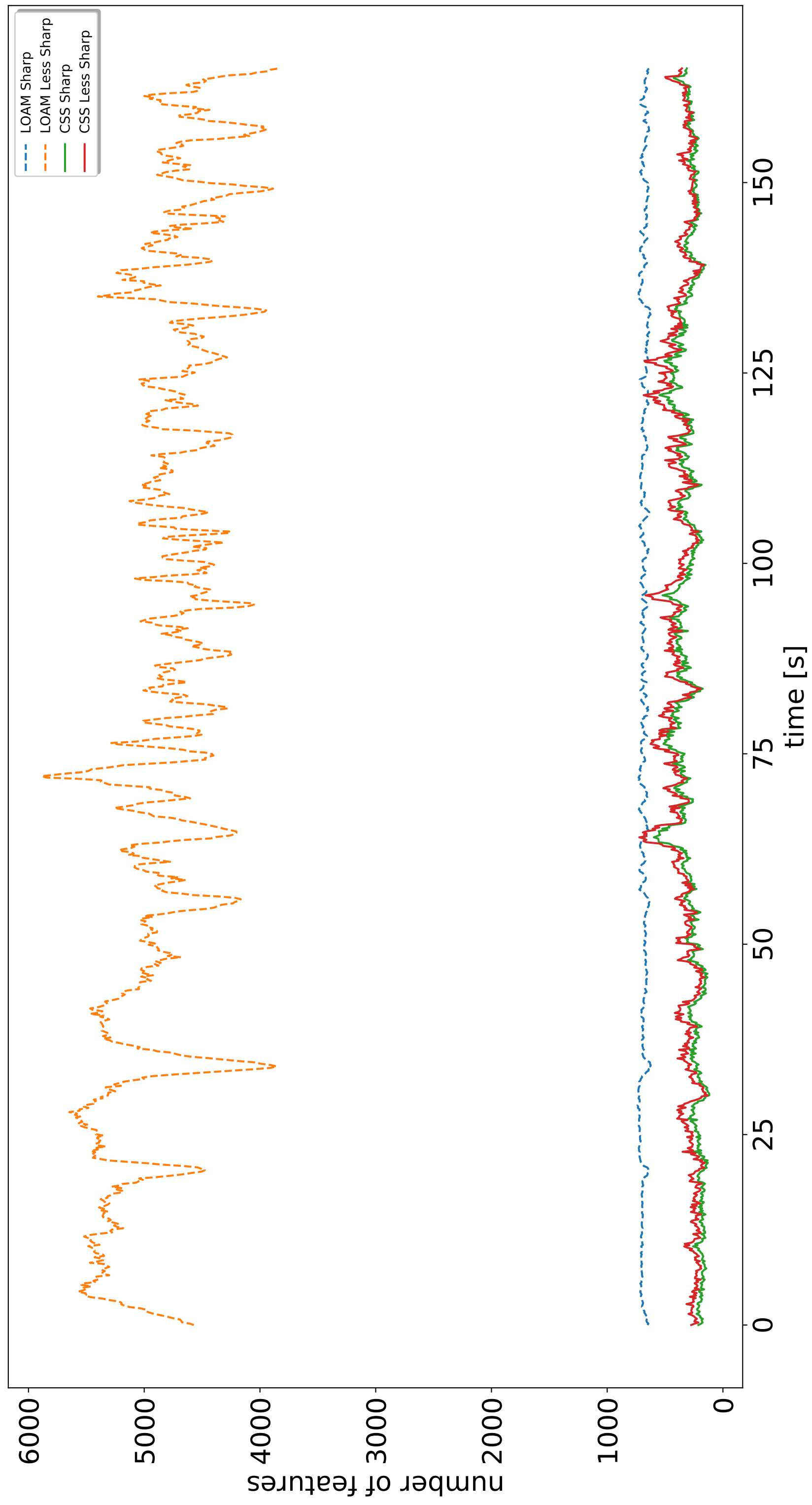
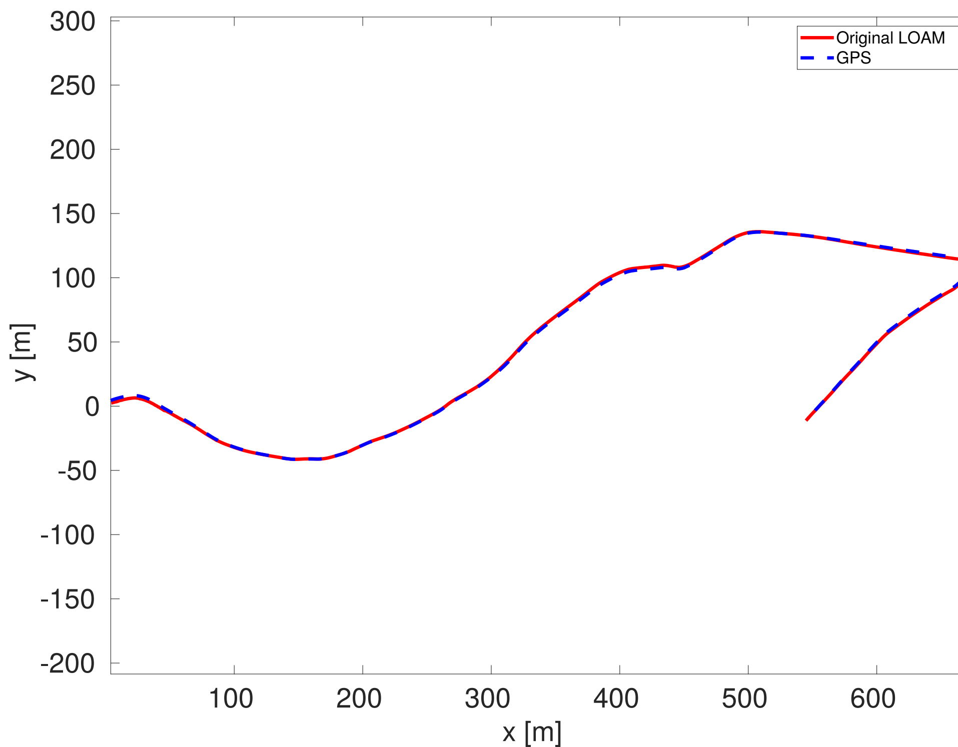
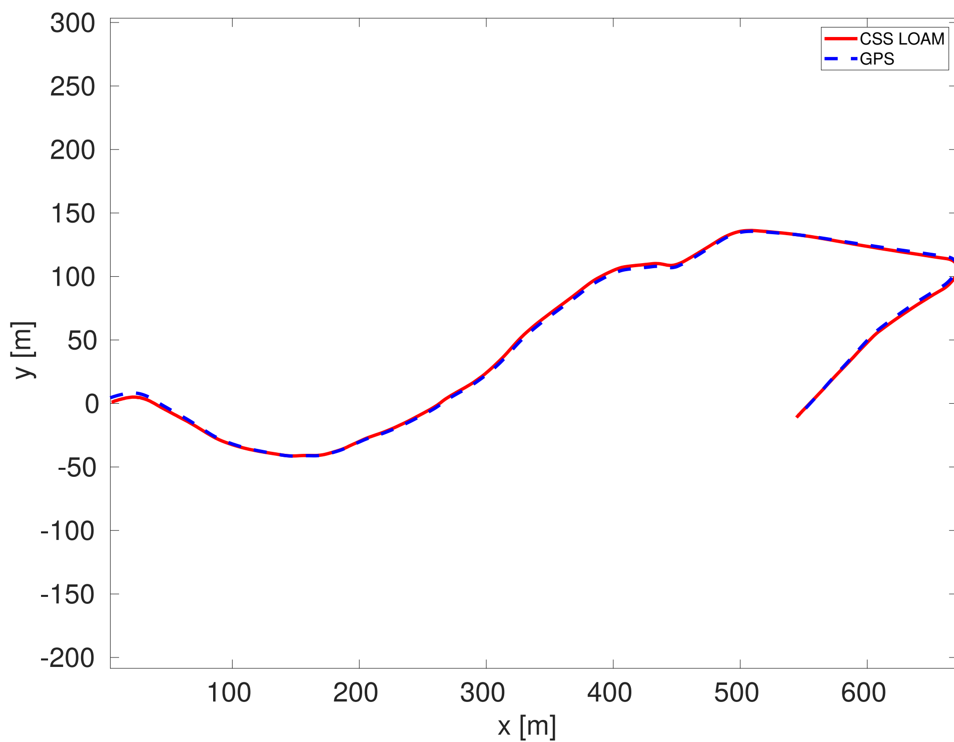


Figure A.30: Number of features in the sequence 09.

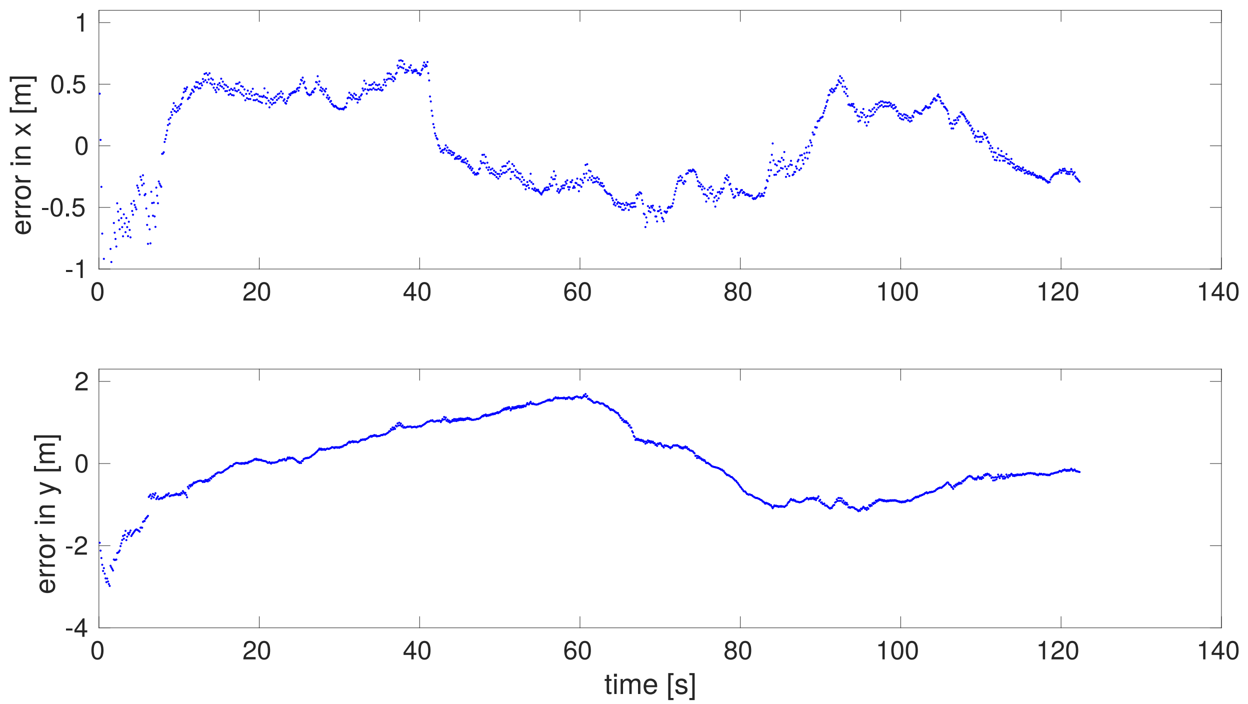


(a) Estimated trajectory using the original LOAM algorithm.

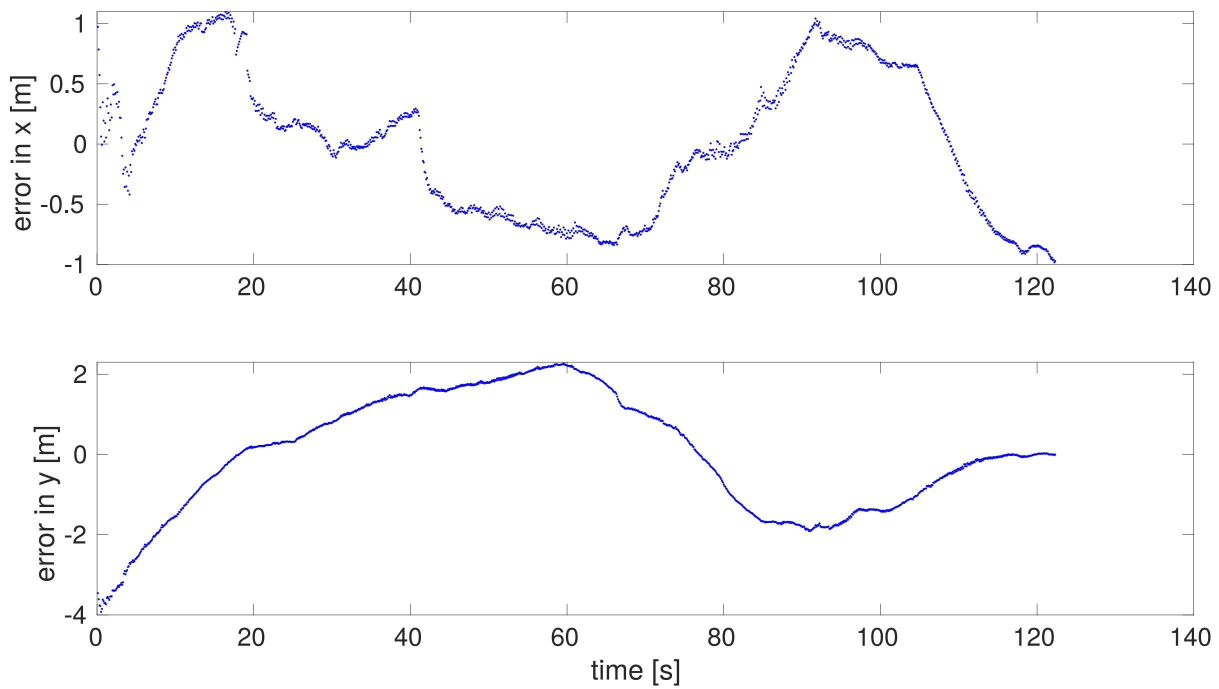


(b) Estimated trajectory using the CSS LOAM algorithm.

Figure A.31: Estimated trajectories compared to ground-truth in the sequence 10.



(a) Errors in x, y produced by the Original LOAM algorithm.



(b) Errors in x, y produced by the CSS LOAM algorithm.

Figure A.32: Errors in x, y of the original LOAM algorithm and the CSS LOAM algorithm in the sequence 10.

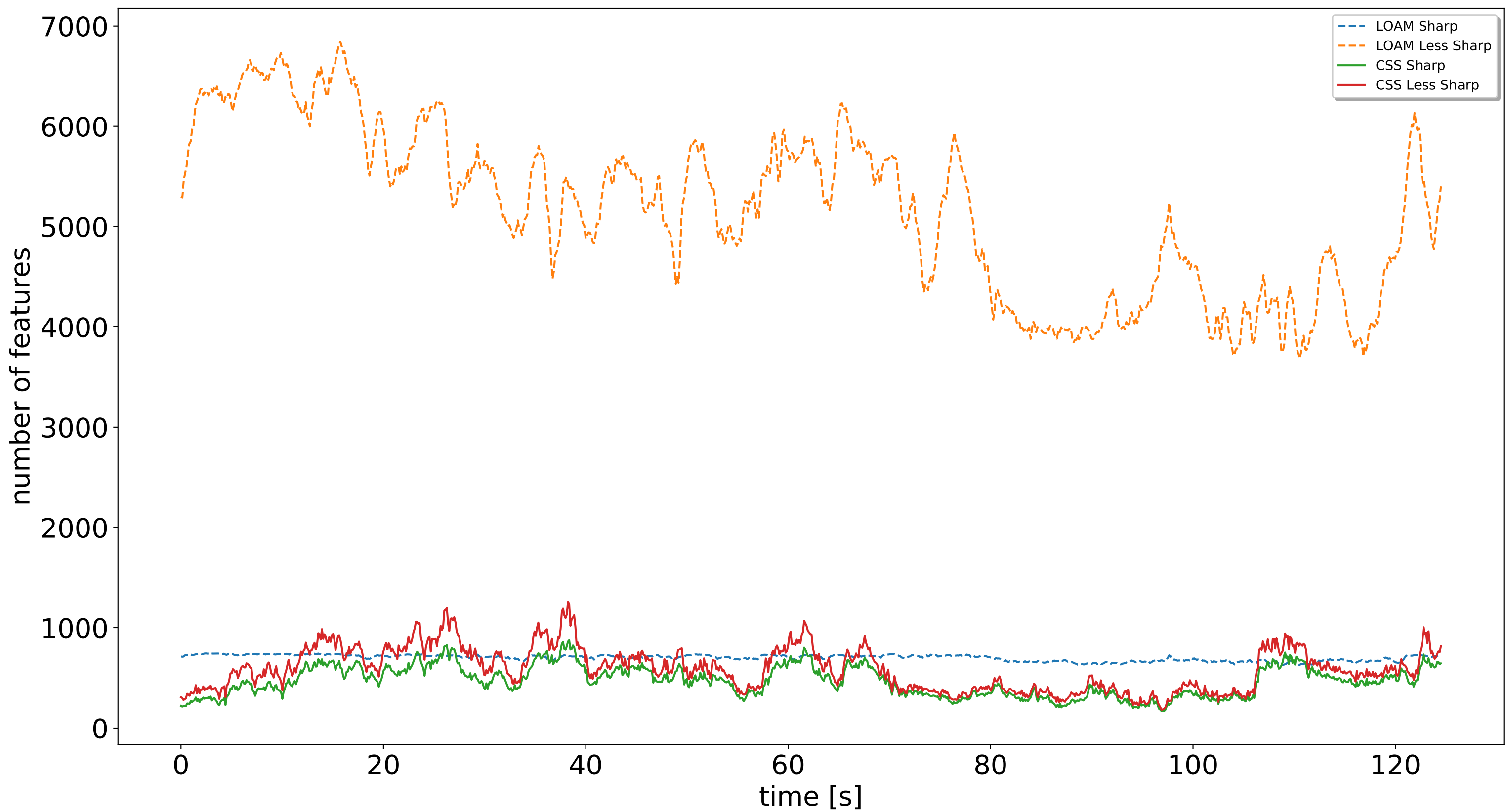


Figure A.33: Number of features in the sequence 10.