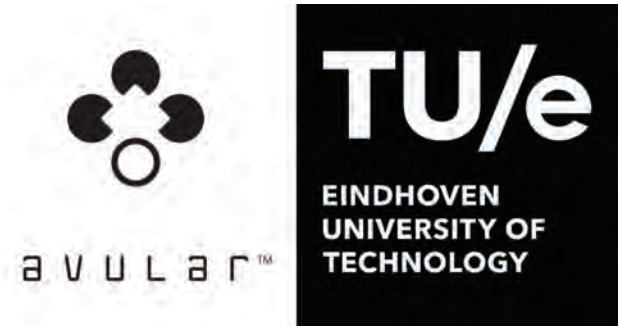
A Factor Graph Based SLAM Back-End Using RGB-D Sensors

Confidential Report: DC 2019.087

September 5, 2019



Eindhoven University of Technology
Department of Mechanical Engineering
Dynamics and Control Group

Author:
B.Song 1283987

Supervisors:
dr.ir.A.A.J.Lefeber (TU/e)
dr.D.J.Antunes (TU/e)
ir.Y.Steinbuch (Avular)

Technische Universiteit
Eindhoven
University of Technology

Declaration concerning the TU/e Code of Scientific Conduct for the Master's thesis

I have read the TU/e Code of Scientific Conduct[i].

I hereby declare that my Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct

Date

05-09-2019

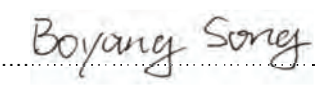
Name

Boyang Song

ID-number

1283987

Signature



Submit the signed declaration to the student administration of your department.

[i] See: http://www.tue.nl/en/university/about-the-university/integrity/scientific-integrity/
The Netherlands Code of Conduct for Academic Practice of the VSNU can be found here also.
More information about scientific integrity is published on the websites of TU/e and VSNU

Acknowledgement

With this opportunity I would like to express my gratitude to all who helped me during my graduation project. I would like to thank all my supervisors, Erjen Lefeber, Yuri Steinbuch, and Duarte Antunes. I appreciate your feedback, guidance, and patience which were of great value to me throughout the project. Also thank you for the many helpful suggestions on my writing skills. Also, I would like to express my gratitude to Henk Nijmeijer for the time you made to provide my work with rigorous feedback and the patience you had throughout all our meetings. A word of appreciation also goes out to Arpit Aggarwal. Thank you for sharing your knowledge and taking the time to improve the readability of my work. Finally, I would like to thank everyone at Avular B.V. for giving me the opportunity to work on this project. It has been a very pleasant and fun working environment for the past months.

Abstract

Many recent mobile robotic applications require the mobile agent to perceive the environment and locate itself within the area at (approximately) the same time. Such an agent could be a self-driving car, an unmanned aerial vehicle or an other mobile agent. This motivates the concept of Simultaneous Localization And Mapping (SLAM). To apply SLAM algorithms, the mobile agent is usually equipped with sensors, such as laser scanners, stereo cameras, and RGB-D sensors.

A SLAM system can be divided into a front-end and a back-end. The front-end is sensor dependent, while the back-end usually forms an optimization problem using the data given by the front-end. When solving the optimization problem, most of the back-ends in current SLAM systems use batch optimization algorithms, such as bundle adjustment. These algorithms do not fit the incremental property of the SLAM problem, where the size of the optimization problem in the back-end increases incrementally over time.

This work proposes a SLAM back-end based on a factor graph formulation and takes RGB-D sensor data as input. The proposed system consists of three parallel processes running on a CPU. The processes are respectively used for tasks of estimating the trajectory, detecting possible loop closures, and creating a map of the environment. The three processes take the sensor data and communicate with each other to output an estimated trajectory and a dense map with occupancy information.

The main difference of this work from the other SLAM frameworks is the usage of an incremental optimization algorithm, instead of batch optimization algorithms. To implement this algorithm, a factor graph is created and maintained during the system running. Based on the factor graph, a node culling mechanism is designed and implemented for loop closure correction purpose.

Finally, several experiments using online real-world datasets are carried out for evaluation. The results show that the proposed system can give a trajectory estimate as well as a dense occupancy map of the working area, that agree with the ground truth. Thereafter, the experimental results are analyzed and several causes for the estimation error are pinpointed.

List of Symbols

δ	The tuning constant in the Huber function.
$\mathbf{0}_{m\times n}$	The zero matrix of dimension $m \times n$.
$\mathcal{F}_i$	The body-fixed frame at time step i.
$\mathcal{F}_w$	The world frame.
$e_{i,j}$	The error between the prediction and measurement of the j-th landmark at time step i.
K	Camera intrinsic matrix.
l_j	The j-th landmark position in the world frame.
$m_{i,j}$	The measurement to the j-th landmark at time step i.
R_i	The rotation matrix from the world frame to the body-fixed frame at time step i, expressed in the world frame.
T_i	The transformation from the world frame to the body-fixed frame at time step i, expressed in the world frame.
t_i	The three dimensional vector translating the origin of the world frame to the origin of the body-fixed frame at time step i, expressed in the world frame.
$T_{i,k}$	The transformation from the body-fixed frame at time step i to the body-fixed frame at time step k, expressed in the body-fixed frame at time step i.
V	The diagonal covariance matrix of the zero-mean Gaussian noise.
v	The zero-mean Gaussian noise.

Contents

Chapter 1

Introduction

1.1 Background

In the robotics community, different types of mobile robots have been developed for various indoor and outdoor applications in the past years. Some examples are given in Figure 1.1, including robot vacuum cleaners at home, unmanned aerial vehicles (UAVs) in agricultural industry, rovers for space exploration, and autonomous cars.

Figure 1.1: Examples of robotic applications. Left-top: robot vacuum cleaner at home [1]. Right-top: unmanned aerial vehicle in agriculture [2]. Left-bottom: rover for space exploration [3]. Right-bottom: autonomous car [4].

All these applications require the robots to work autonomously or partial-autonomously in unknown or even changing environments. For example, when the robot vacuum cleaner starts working in a new household environment, it needs to build a map of the working area, and based on this map, plan an efficient cleaning path; while performing the cleaning job, it should be able to locate itself such that it does not clean the same place several times; in addition, when the environment changes, e.g., some furniture being moved, it should also update the map accordingly. Similar requirements are also found in many other scenarios. It can be concluded that a high level of autonomy of the mobile robots is required in more and more applications.

To achieve autonomy, the robots should be capable of perceiving the surroundings and locating itself within the area. To this end, the robot is usually equipped with sensors, such as laser scanners, stereo cameras, and RGB-D sensors (sensors that give both color image and depth image). These sensors are able to detect features, which are distinctive static structures in the environment, and regard these features as landmarks. For example, when using vision systems, the special pixels are considered. Thereafter, the landmarks are used for determining the motion of the robot, by calculating the relative poses to these landmarks. The pose includes

both the position and orientation information. Using the motion information, combined with the initial condition of the robot, the robot is able to locate itself over time. Conversely, when the robot's location is obtained, the locations of the landmarks can also be determined. All the landmarks together can be seen as a rough map of the environment, since the map is sparse.

It can be seen that, in almost all the situations, the localization and the mapping processes happen at (approximately) the same time, because they are dependent on each other. Intuitively, to locate the robot in the map, the map should be known, and to map the environment, the localization information is needed. This motivates the concept of Simultaneous Localization And Mapping (SLAM). Precisely, SLAM is the technology by which an agent can incrementally construct and update a consistent map of an unknown environment, while simultaneously locating itself within the map. Such an agent could be a self-driving car, an unmanned aerial vehicle or an other mobile agent.

In the next three sections, we give a brief introduction to the general structure of SLAM systems, and address two commonly encountered problems in SLAM.

1.1.1 SLAM System Architecture

A typical SLAM system contains two components: the front-end and the back-end [5]. The front-end processes raw sensor data and transforms it to measurements that are amenable for the back-end estimation, while the back-end performs inference on the abstracted data produced by the front-end [5]. This structure is summarized in Figure 1.2.

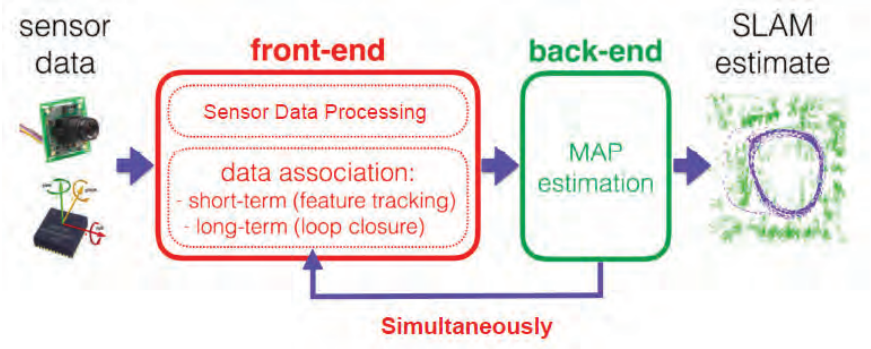


Figure 1.2: A typical SLAM system architecture containing front-end and back-end [5]. Note that the MAP estimation in the back-end stands for Maximum-A-Posteriori estimation.

The front-end of a SLAM system is application dependent. Usually, different choices of sensors lead to different data processing methods in the front-end. For examples, sonar sensors in underwater applications, downward monocular cameras in drone applications, and laser scanners in many indoor applications. All these applications apply different SLAM front-ends. On the other hand, the approaches used in the back-end are similar because they are independent of the sensor choices. In the back-end of the SLAM system, a Maximum-A-Posteriori (MAP) estimation problem is solved. Most of the back-end in recent SLAM systems formulate the problem of MAP estimation as an optimization problem, where an optimal solution is obtained by minimizing the errors between the measurements and the predictions. The predictions are computed by the states to be determined, namely the robot poses and the landmark locations. By solving the optimization problem, the robot poses and landmark locations, that fit the measurements best, can be obtained.

1.1.2 Loop Closure

A commonly encountered problem in SLAM is drift in pose estimation over time. This problem comes from the noise in the measurements, which leads to an error in pose estimation. As time goes along, the error accumulates and becomes visible in the estimated robot trajectory. This also causes a drift in the created map, since the map construction depends on pose estimation. This

phenomenon is visualized in the left part of Figure 1.3. To compensate the drift, the robot has to return to some pre-visited place and successfully detect this fact. Next, the newly created map is fused with the previous one to generate a consistent map. This procedure is called loop closure in SLAM and is visualized in the right part of Figure 1.3.

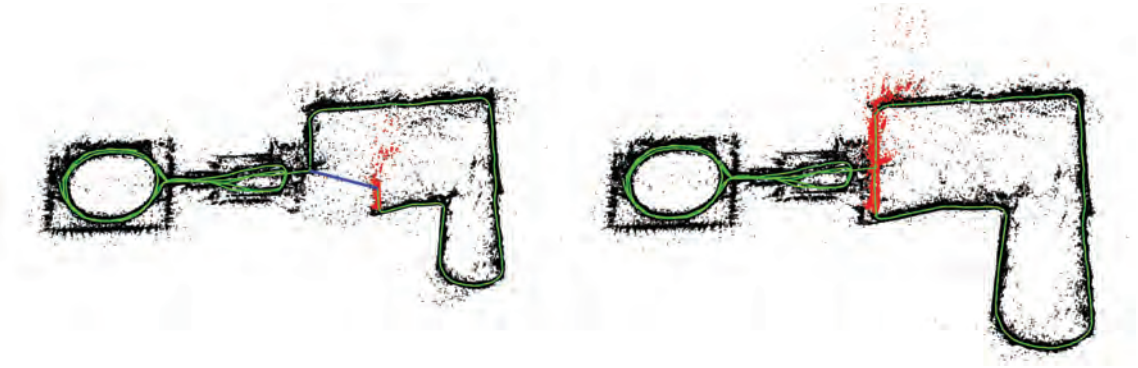


Figure 1.3: Illustration of the influence of drift and loop closure [6]. Left: without loop closure, both the estimated robot trajectory (the green lines) and the created map (the black dots) drift. The blue line shows the drift before loop closure. Right: after loop closure, the drift is compensated, which also results in a consistent map. The red dots are the fused part of the map.

The loop closure function is essential for SLAM systems. It not only compensates the drift in trajectory and map, but also guarantees that the map does not grow boundlessly, which may cause memory shortage in real robotic systems.

1.1.3 Dense Mapping

Another problem often considered by SLAM system developers is dense mapping. Dense mapping is opposite to sparse mapping. The difference between these two is shown in Figure 1.4. It can be observed from the figure that the sparse map only contains landmarks but not the occupancy information of the space.

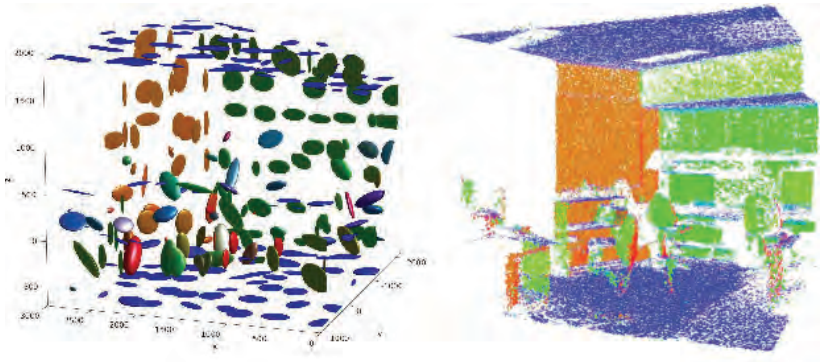


Figure 1.4: The examples of sparse map (left) and dense map (right) [7].

The landmarks in a sparse map is useful for localizing the mobile agent, with a reasonable computational cost. However, a sparse map is not suitable for tasks like path planning, since the occupancy information is dropped by it. On the other hand, a dense map is capable of such tasks, but the downside is it requires more computational resources.

To show that a dense map is essential for robot autonomy, we take the robot vacuum cleaner as an example. While cleaning the room, the robot vacuum cleaner has to move back to the charging station when the battery is low, and this can happen at any point in the map. As a result, the robot has to find a route from the current position to the charging station. The route in this

situation is determined by solving a path planning problem in robotics, and it requires a dense map.

This work aims to develop a back-end of a SLAM system. In addition, other key components in SLAM, such as loop closure and dense mapping, are also considered.

1.2 Literature Review

In this section, several state-of-the-art SLAM frameworks are first reviewed. In the study of these frameworks, we try to identify the front-end and back-end of these systems, and study how they perform loop closure and dense mapping. At the end, an optimization algorithm is reviewed and compared with the current optimization algorithms used in SLAM.

1.2.1 ORB-SLAM and ORB-SLAM2

ORB-SLAM [6] and its upgraded version ORB-SLAM2 [8] are feature-based SLAM frameworks. Specifically, all the tasks in these SLAM frameworks depend on the same feature, namely the ORB feature. Recall that features in computer vision are special pixel structures on the image. ORB is short for Oriented FAST and Rotated BRIEF [9], which builds on the FAST keypoint detector and the BRIEF descriptor. They are further explained in the following.

1. FAST is short for Features from Accelerated Segment Test, which is a high-speed corner detection method used in computer vision to extract feature points [10].

2. BRIEF is short for Binary Robust Independent Elementary Features, which is a feature descriptor that is very efficient to build and match [11].

The keypoint detector locates the feature's position on the image, while the descriptor represents the unique structure of the feature. ORB features are extremely fast to compute and match, while they have good invariance to viewpoints [6]. The invariant property to viewpoints guarantees the features are detected correctly despite of the orientation and the scale of the features on the image.

ORB-SLAM adopts a multi-threaded architecture, which allows three threads that run in parallel. The three threads are tracking, local mapping, and loop closing. The tracking thread can be seen as the front-end of the system, while the local mapping and loop closing threads cover the back-end and part of the front-end. A brief introduction of these threads is given in the following:

- The tracking thread is in charge of localizing the camera with every frame and deciding when to insert a new keyframe. In ORB-SLAM, a frame is defined as the image captured by the camera at each time step. A keyframe is a certain frame that has at least 50 features observed in several different camera views and more than 20 frames have passed by from the last keyframe [6].

- The local mapping thread processes new keyframes and performs an optimization algorithm named bundle adjustment (BA) to achieve an optimal reconstruction in the local surroundings of the camera pose. Bundle adjustment is an optimization algorithm, which tries to find the optimal solution for both robot poses and landmark locations.

- The loop closing thread detects large loops and corrects the accumulated drift by performing a pose graph optimization, which only updates the robot poses instead of poses and landmark locations.

One drawback of ORB-SLAM is that it is designed for monocular cameras only, thus it suffers from the problem of scale drifting. This is because the depth information is missing when using a single camera, so the scale of the observed features is also a parameter that needs to be determined. The estimation of scale introduces an extra error, resulting in the drifting problem. To solve this problem, the ORB-SLAM2 system has been carried out to support stereo and RGB-D cameras. The stereo and RGB-D system estimates the map and trajectory with a metric scale so it does not

suffer from scale drift [8]. The scale drifting problem is visualized in Figure 1.5 using a popular online dataset [12].

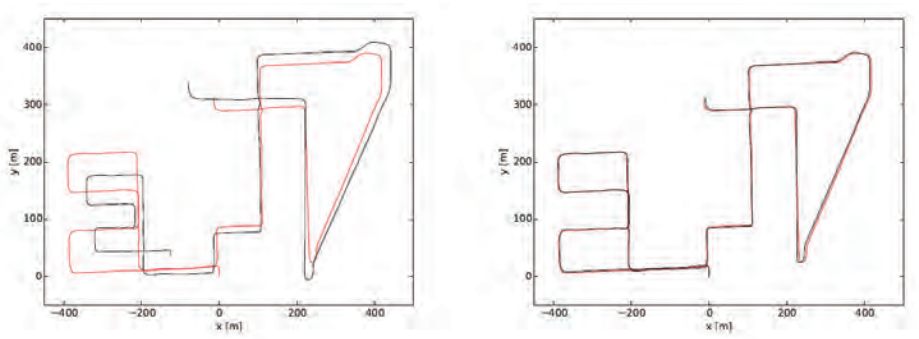


Figure 1.5: Estimated trajectory (black) and ground-truth (red) in KITTI 08 [12]. Left: monocular ORB-SLAM, right: ORB-SLAM2 (stereo). Monocular ORB-SLAM suffers from severe scale drift in this sequence, especially at the turns. In contrast, the stereo version using ORB-SLAM2 is able to estimate the true scale of the trajectory and map without scale drift [8].

For loop closure, ORB-SLAM and ORB-SLAM2 depend on a pre-trained dataset containing thousands of features. When performing loop closure detection, the extracted features in the current color image are searched inside the dataset. The appearance information of the features in the dataset are stored. When most of the appeared features in two images overlap with each other, a loop closure happens. To correct the estimation of robot poses and landmark locations using the loop closure information, ORB-SLAM and ORB-SLAM2 compute a similarity transformation between these two images, and add this transformation as a new condition in the optimization problem. By resolving the optimization problem, the updated estimates can be obtained.

Both ORB-SLAM and ORB-SLAM2 cannot generate a dense map in real-time. Although [8] presents several dense reconstructions of different environments, the reconstruction procedure does not happen while tracking the motion of the camera. In other words, ORB-SLAM and ORB-SLAM2 first estimate the camera trajectory based on the features observed in different camera views, and then use the estimated trajectory, combined with the sensor depth information, to reconstruct a dense map. Furthermore, the dense map generated afterwards adopts a point cloud format, which uses a huge number of 3D points to represent the structure of the environment. Therefore, it can be inferred that although the point cloud map representation is visually pleasant, it does not provide occupancy information which is necessary for path planning. Furthermore, to guarantee the density, a large number of points are stored, which indicates that point cloud maps are generally memory-inefficient.

In conclusion, ORB-SLAM and ORB-SLAM2 are state-of-the-art SLAM frameworks with multi-threaded system architecture, which enables real-time processing using only a CPU. In addition, ORB-SLAM2 supports all the mostly used visual sensors, including monocular cameras, stereo cameras, and RGB-D sensors. However, both of them are not suitable for applications requiring active path planning, because they cannot provide a dense map in real-time. Furthermore, these systems can not incorporate other sensor data, such as inertial measurement unit (IMU) data.

1.2.2 RGBD-SLAM

RGBD-SLAM [13] is the first popular open-source SLAM system [6]. RGBD-SLAM is also a feature-based SLAM framework, which can be broken up into three modules: the front-end, the back-end, and a separate mapping module:

- The front-end computes frame-to-frame motion by feature matching and using the iterative closest point (ICP) algorithm. ICP is an algorithm employed to minimize the difference between two sets of points [14].

- The back-end performs a pose graph optimization with loop closure constraints. In contrast to ORB-SLAM, RGBD-SLAM does not generate a sparse landmark-based map. Thus in the back-end, it only performs the pose graph optimization instead of bundle adjustment. Recall that the pose graph optimization algorithm only updates robot poses, while bundle adjustment optimizes both robot poses and landmark locations.

- The mapping module creates the dense map using sensor data and the trajectory estimated from the back-end.

For loop closure detection, RGBD-SLAM does not require a pre-trained dataset. Instead, it employs an efficient and straightforward-to-implement strategy by randomly selecting several keyframes and checking if any two of them match to each other. A keyframe in RGBD-SLAM is different from that in ORB-SLAM, as it is defined as a frame that cannot be matched to any previous keyframes. This strategy is simple but highly dependent on other system mechanisms to minimize the drift while tracking. Thus in general, the performance of loop closure cannot be guaranteed.

To build a dense map, the raw RGB-D sensor data are projected into a common coordinate frame, based on the information about the estimated trajectory. However, simply projecting the raw data is highly inefficient. Therefore, in that system, a 3-dimensional probabilistic occupancy map is created from the raw data.

1.2.3 Batch Optimization and iSAM

The SLAM process is inherently incremental. Intuitively, at every time step, the states to be determined in the SLAM problem are recomputed according to the newly observed data as well as the old data. In addition, the states to be determined also expand at each time step, due to newly observed landmarks and a new robot pose at the current time step. It is also known that the newly measured data often only influences the poses and observed landmarks at the previous few time steps, instead of the states long ago (if no loop closures happen).

On the other hand, the commonly used optimization methods in most of the current SLAM systems, including ORB-SLAM/ORB-SLAM2, are batch optimization algorithms, such as bundle adjustment. In other words, batch optimization algorithms compute all the states to be determined in the problem using all the data obtained from the sensor. Thus, when using these algorithms in SLAM, at each time step, all the states are recomputed considering the new and old measurement data, although most of them change little. It can also be seen that the complexity for solving such an optimization problem increases as time goes along. As a result, these algorithms are applied subject to some fixed horizon. Taking ORB-SLAM as an example, to make it run in real-time, the local mapping thread performs bundle adjustment with a horizon of 12 frames. This is also the reason why the BA algorithm in local mapping thread is called LOCAL BA in ORB-SLAM.

Apart from setting a horizon for batch optimization algorithms, other solutions exist, such as the incremental smoothing and mapping (iSAM) algorithm. iSAM, proposed by [15], is a novel approach to the SLAM problem that is based on fast incremental matrix factorization. This algorithm exploits the naturally sparse information matrix, which represents the correlations between states. When new measurements and states occur, the information matrix is also updated, but only those matrix entries that actually change are recalculated. Later, iSAM has been upgraded to iSAM2 [16], which uses a Bayes tree data structure and enables fully incremental operation of the optimization problem. The iSAM/iSAM2 algorithms are not as popular as batch optimization algorithms in SLAM, since most of the current SLAM systems including ORB-SLAM/ORB-SLAM2 and RGBD-SLAM use batch optimization algorithms. However, the incremental property of iSAM/iSAM2 is more suitable to SLAM problems.

1.3 Problem Definition and Objective

Based on the knowledge of the current SLAM systems, the proposed system should fulfill the following requirements. Note that the requirements are mainly for the back-end of a complete SLAM system, since this work lays its focus on that.

1. The system should output the position and orientation of the robot at each time step, and consequently, give **an estimated trajectory**;
2. The system should give **a dense map** of the working area, which incorporates the occupancy information;
3. The system should be able to handle **loop closures** such that the trajectory and the map are consistent;
4. The system should be easy for usage on different software platforms;
5. The system should be able to run on a CPU in real-time.

The current SLAM systems can partially meet the above requirements, because normally the dense mapping process is performed by GPUs (the second and the fifth requirements can not be met at the same time).

The SLAM problem is incremental with its size growing over time. However, most of the current SLAM systems apply batch optimization algorithms in the back-end, which inherently do not suit the incremental property. The problem is caused by the fact that the batch optimization algorithms function on a manually selected interval of the previous frames and update the states only inside the interval. As a result, we set the primary **objectives** of this work as follows:

1. Investigate an incremental optimization method in the back-end that solves the problem mentioned above.
2. Develop a SLAM back-end that fulfills all the requirements above.

The main **contributions** or the key differences from other SLAM systems are presented in the following.

1. Instead of using the batch optimization algorithms as most of the other SLAM systems do, we propose a factor graph based optimization method that uses an incremental algorithm.
2. Based on the factor graph formulation, a culling method is proposed and implemented to incrementally create a pose graph from a full graph (a graph containing both poses and landmarks). This makes the loop closure correction feasible in real-time.
3. A bag-of-words based loop closure detection algorithm is tested in combination with other SLAM modules in real-time.

1.4 Outline of the Thesis

This thesis is organized in the following structure:

- Chapter 2: Landmark Decision
  This chapter reviews the camera model used in computer vision. Based on this knowledge, the method of choosing landmarks from raw pairs of RGB-D images is elaborated on. Practical issues such as distortion of the color images, mismatching of feature points, are also discussed.
- Chapter 3: Factor Graph-Based SLAM
  In this chapter, the mathematical formulation of the observation model is derived. Using this model, the errors between prediction and measurements are obtained. Then we present the

expression of the cost function for the optimization problem to minimize. Since tons of data exist in this problem, a factor graphical model is introduced to efficiently organize these data, and we propose the construction procedure. Finally, the iSAM algorithm proposed by [16] is considered to numerically compute a solution to the optimization problem and experimental results are also presented.

- Chapter 4: Loop Closure
  This chapter reviews the bag-of-word model, which is widely used for loop closure detection algorithms. Later the motivation of a pose graph is addressed and the construction method based on the graph created following Chapter 3 is proposed.

- Chapter 5: Dense Mapping
  This chapter introduces a dense volumetric map representation named Octomap, and then compares it with point cloud map representation. At last, the approach of creating a dense map using pose estimates and point clouds is presented.

- Chapter 6: System Integration and Evaluation
  This chapter discusses the integration of the three components presented in Chapter 3, Chapter 4, and Chapter 5. The communication delays between processes are also discussed. Then an evaluation method and evaluation quantities are introduced. Finally, the evaluation results are presented and analyzed.

- Chapter 7: Conclusions and Recommendations
  In the final chapter, the objectives of this work are first reviewed. Based on this, the conclusions are drawn. At the end, several recommendations for future work are mentioned.

Chapter 2

Landmark Decision

In this chapter, we describe the approach of choosing landmarks and obtaining measurements to these landmarks based only on the raw RGB-D images. Practical issues, such as the influence and the correction of distortion, are also discussed.

2.1 RGB-D Sensor

RGB-D sensors, such as the Microsoft Kinect, are cost-efficient and light-weight sensors compared to 3D laser scanners which are widely used in SLAM. Usually, a RGB-D sensor consists of a color image sensor, a depth image sensor, and an Infra-Red (IR) projector (see Figure 2.1). Using these sensors, a RGB-D sensor can provide color information as well as the estimated depth for each pixel, based on either the structured light (e.g. Kinect v1) or the time of flight (e.g. Kinect v2) principle. An example of a pair of RGB-D images can be found in Figure 2.2, where in the depth image the grey scale is used to represent distance information. The darker the pixel is in a depth image, the shorter the distance. If no measurements are obtained for some pixels, the depth image sets a black color for those pixels.

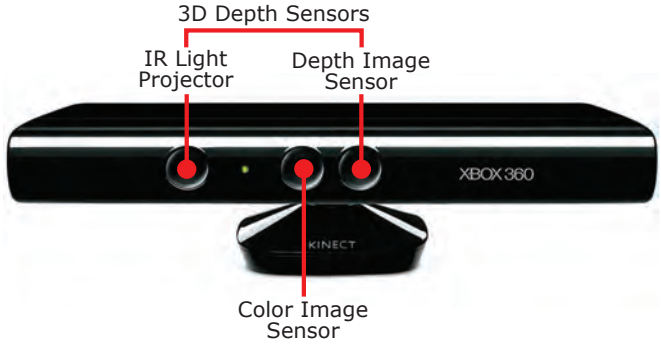


Figure 2.1: An introduction of components of the Microsoft Kinect v1.

Figure 2.2: An example of RGB-D images: RGB (color) image (left) and depth image (right)

2.2 Landmark Candidates

In RGB-D based SLAM, the features extracted from the color images are usually considered as landmark candidates. Recall that features are the pixels on the images that have specific structures. In this work, the ORB features are chosen, because of their operation efficiency and invariant property to the rotation of the features. ORB features can provide two kinds of information: the positions and the descriptors of the features. They are introduced as follows:

- The positions of the features on the color images are 2-dimensional coordinates, denoted as (u, v). The coordinates (u, v) are usually called pixel coordinates. The pixel coordinate system is defined with the origin located at the left-top corner of the image, x axis pointing to the right horizontally, and y axis pointing down vertically (see Figure 2.3a for reference).
- The descriptors of the features are binary vectors. When matching ORB features, the Hamming distance between these binary vectors is computed. The Hamming distance is computed by counting the number of the different bits in the two binary vectors. If the distance is small, then the features are matched.

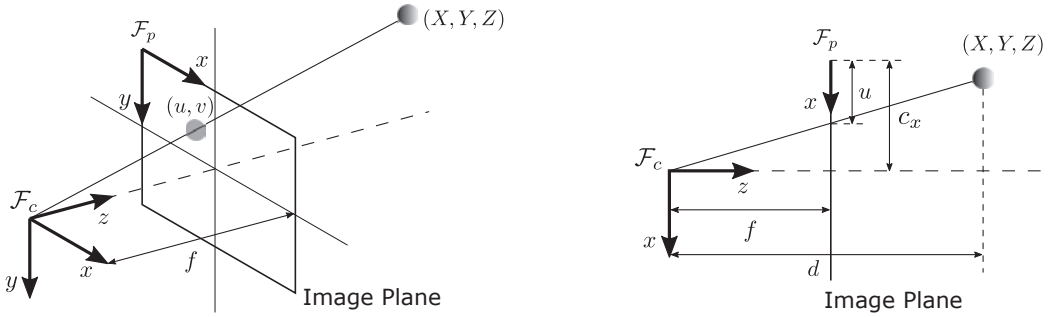


Figure 2.3: (a) Illustration of pixel coordinates and sensor coordinates, where $\mathcal{F}_p$ and $\mathcal{F}_c$ stand for the pixel coordinate system and sensor coordinate system, respectively. (b) Top view of coordinate systems.

The pixel coordinates are transformed into sensor coordinates associated with depth information. The sensor coordinate system of RGB-D sensors are defined as the coordinates with the origin at the optical center of the camera, x, y axes parallel to the pixel coordinate system, and z axis pointing outward of the camera (see Figure 2.3a for reference). To find the transformation between the pixel coordinates and the sensor coordinates, the top view of the coordinate systems in Figure 2.3a is presented in Figure 2.3b, and the transformation is given by

$$\begin{aligned} X &= \frac{u - c_x}{f_x} d, \\ Y &= \frac{v - c_y}{f_y} d, \\ Z &= d, \end{aligned} \tag{2.1}$$

or in matrix form

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} d, \tag{2.2}$$

where $\begin{bmatrix} X & Y & Z \end{bmatrix}^\top$ are the sensor coordinates, $\begin{bmatrix} u & v & 1 \end{bmatrix}^\top$ are the pixel coordinates with an extra constant 1, d is the depth for the operating pixel, and f_x, f_y, c_x, c_y are pre-estimated parameters, representing the focal length in x and y directions, and the optical center's x and y positions, respectively. Also note that all the above parameters are in pixels. The matrix consisting of these

parameters is the camera intrinsic matrix, which is assumed to be known in this work, and denoted as

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.3}$$

Features without depth information are neglected. The code for aligning the pixel coordinates with the depth information to obtain the sensor coordinates are given below, which is the implementation of (2.2). If no depth information is available, the function would not be called.

```
cv::Point3d Core::ComputePos(const cv::Point2d &p, const double &d)
{
    cv::Point2d pp((p.x-_K.at<double>(0,2))/_K.at<double>(0,0),
        (p.y-_K.at<double>(1,2))/_K.at<double>(1,1));
    double dd = double(d)/_scale;
    return cv::Point3d(pp.x*dd, pp.y*dd, dd);
}
```

After the above operation, the 3D measurements of the landmarks are obtained. However, due to distortion, the computed measurements are not equally accurate. This phenomenon is discussed in the next section.

2.3 Distortion

The distortion in vision systems is caused by the camera lens. In general, distortion can be divided into three classes: barrel distortion, pincushion distortion, and the mixture of both types. The first two types of distortion are illustrated in Figure 2.4.

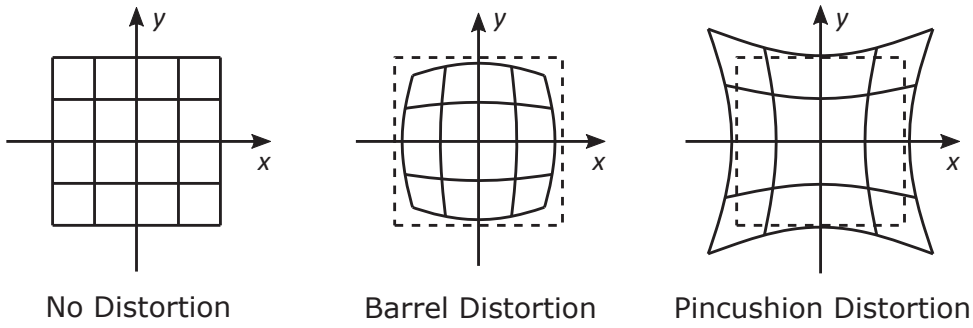


Figure 2.4: An illustration of the influence of distortion in vision systems

To compensate the influence of distortion, an undistorting operation is needed. This operation is applied on the normalized coordinates (x, y), which are given by setting $d = 1$ in (2.2):

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}. \tag{2.4}$$

Then the undistorted normalized coordinates $(\tilde{x}, \tilde{y})$ are computed by [17]

$$\begin{aligned} \tilde{x} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2), \\ \tilde{y} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^2) + 2p_2 xy, \end{aligned} \tag{2.5}$$

where k_1, k_2, k_3, p_1, p_2 are pre-estimated distortion parameters, and $r = \sqrt{x^2 + y^2}$. The distortion parameters as well as the focal length and optical center parameters are determined through camera calibration. Since the calibration of a camera is not the main concern of this

work, we only briefly introduce the general idea. The calibration is carried out by Zhang's method [18], which requires the camera to observe a 2D pattern, e.g. a chessboard, at several different views. The pixel coordinates and the planar positions of the corners on the pattern are related by the intrinsic matrix and the distortion model in (2.5), which is nonlinear. Thus in Zhang's method, the camera parameters and the distortion parameters are estimated by minimizing the fitting error of the pixel coordinates and the planer positions according to their relations.

The C++ code for the feature extraction and undistortion are given by the following list, where the undistortion function is given by OpenCV [17], but the idea behind this function is the same as (2.5).

```
void Core::ExtractFeatures(const cv::Mat& imRGB)
{
    std::vector<cv::KeyPoint> keypoints, undistorted_keypoints;
    cv::Mat descriptors;

    // feature extraction using ORB_SLAM2::ORBExtractor
    (*_extractor)(imRGB, cv::Mat(), keypoints, descriptors);

    // if the first distortion parameter is set to zero,
    // undistortion is disabled.
    if(_D.at<double>(0) == 0.0)
    {
        undistorted_keypoints = keypoints;
    }
    else
    {
        cv::Mat mat(keypoints.size(),2,CV_64F);
        for(unsigned int i=0; i<keypoints.size(); ++i)
        {
            mat.at<double>(i,0) = keypoints[i].pt.x;
            mat.at<double>(i,1) = keypoints[i].pt.y;
        }

        // Undistort points
        mat = mat.reshape(2);
        cv::undistortPoints(mat, mat, _K, _D, cv::Mat(), _K);
        mat = mat.reshape(1);

        // Fill undistorted keypoint vector
        undistorted_keypoints.resize(keypoints.size());
        for(unsigned int i=0; i<keypoints.size(); ++i)
        {
            cv::KeyPoint kp = keypoints[i];
            kp.pt.x = mat.at<double>(i,0);
            kp.pt.y = mat.at<double>(i,1);
            undistorted_keypoints[i] = kp;
        }
    }

    _kp.push_back(undistorted_keypoints);
    _des.push_back(descriptors);
}
```

After the above operation the undistorted normalized coordinates are obtained, and using the

coordinates, the 3D measurements of the landmarks are given by

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ 1 \end{bmatrix} d. \tag{2.6}$$

2.4 Feature Matching and Filtering

When matching the features in different camera views, or finding the same landmarks in different frames, same structures of pixels are searched in different color images by computing the Hamming distances of the descriptors. Recall that the descriptors are vectors with equal length. Thus, the Hamming distance between two descriptors is the number of positions at which the corresponding values are different. For efficiency consideration, only the matched features in two subsequent color images are considered to be landmark candidates. In practice, wrongly matched features are possible, so we perform the Lowe's ratio test [17, 19] to filter incorrect matches.

The Lowe's ratio test works under the assumption that every feature on one image can only match to exactly one feature on the other image. Thus when using Lowe's ratio test, the first two best matches for each feature on the previous image are searched on the subsequent image. Thereafter, the Hamming distances between the feature on the previous image and the two matches are compared. Because of the one-to-one correspondence assumption, the distance between the feature on the previous image and the first best match should be much smaller than the distance of the second best match. In practice, the second distance is multiplied to a ratio, and if the result is greater than the first distance, then the first best match is considered as a correct match. The implementation is given below.

```
// match features
void Core::MatchFeatures(const cv::Mat& des1, const cv::Mat& des2,
        std::vector<cv::DMatch>& good_matches)
{
    cv::Ptr<cv::DescriptorMatcher> matcher =
        cv::DescriptorMatcher::create("BruteForce-Hamming");

    // lowe's ratio test filtering method
    std::vector<std::vector<cv::DMatch>> matches;
    matcher->knnMatch(des1, des2, matches, 2);

    for(size_t i=0; i<matches.size(); ++i)
    {
        if(matches[i][0].distance < _ratio*matches[i][1].distance)
            good_matches.push_back(matches[i][0]);
    }
}
```

The matching results before and after filtering are shown in Figure 2.5. It can be observed that the mismatched features are filtered out, although the number of matches decreases.

After getting all the landmark candidates, the depth information is taken into account. Only the features corresponding to depths within a pre-defined boundary are regarded as landmarks. In addition, the landmarks are associated with indices as well as the 3D positions with respect to the sensor coordinates (the measurements of the landmarks). The whole landmark selection procedure is further explained with codes in Appendix A.1.

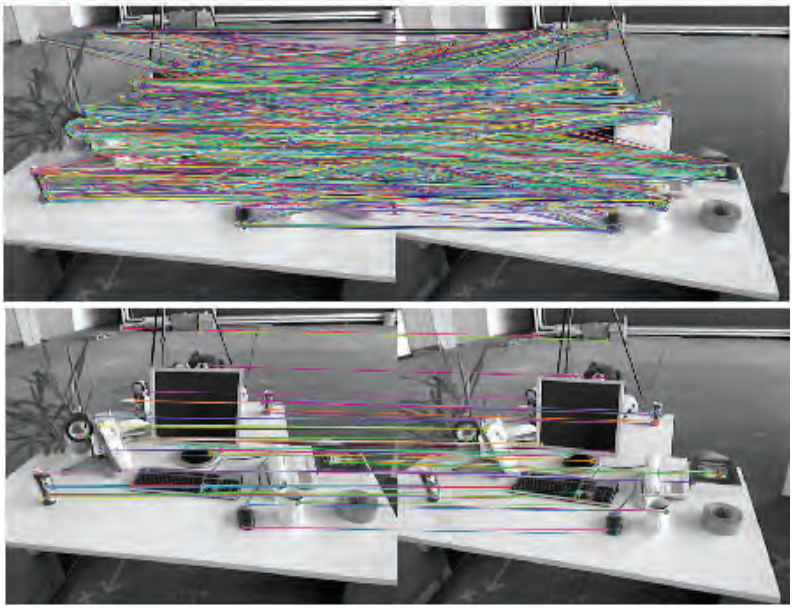


Figure 2.5: The feature matching results before (up) and after (bottom) filtering. The color lines connect the corresponding features.

2.5 Summary

In this chapter, we reviewed the commonly used camera model and the feature matching method. Based on the review, we discussed how to formulate the required landmark measurement data from raw RGB-D images. It can be seen that the number of the measurements is large, since normally hundreds of features are extracted from each image. As a result, in the next chapter, we introduce a data structure, namely a graph, to organize the sensor data, and to perform an optimization algorithm to reason about the optimal solution of the SLAM problem.

Chapter 3

Factor Graph-Based SLAM

This chapter focuses on the functionality requirement of the SLAM core module. In this work, the poses at each time step and the landmark positions are regarded as the states of our system, and the measurements are obtained as described in Chapter 2. By defining the states and the measurements, the SLAM problem is seen as an estimation problem of the states given the measurements. To this end, to estimate the states, the observation model, which links the system states and the measurements, is first studied. Next, the graph data structure is introduced to mathematically form the optimization problem, depending on the large number of measurements. Thereafter, we show that the outliers in the measurements can be rejected by modifying the cost function of the optimization problem. Finally, experimental results using an online real-world RGB-D dataset are shown and analyzed.

3.1 Formulation of the Observation Model

To formulate the observation model, a typical SLAM problem for a planar mobile robot in Figure 3.1 is considered. Although it is a 2D problem, we consider it as a 3D problem with three degrees of freedom fixed. The mobile robot (gray triangle) moves along some trajectory (solid line), and the landmarks (circles) are located in the working area of the robot. In addition, the landmarks are assumed to be static in this area. In this scenario, both the trajectory and the locations of the landmarks are not given to the robot. To enable the robot to locate itself and perceive the environment, it is equipped with a sensor, e.g., an RGB-D sensor, which can measure the x, y, z coordinates of the landmarks with respect to the sensor. Using these measurements, the following two outputs should be delivered by the SLAM algorithm:

1. the position and orientation of the robot at each time step, and consequently, the estimated trajectory;

2. the locations of the landmarks in the working area coordinates, and all the landmarks together are regarded as a sparse map of the working area.

In the following sections, we first introduce the precise definitions of the notations needed for the formulation. Using the defined notations, we describe the general principle for searching solutions of the SLAM problem, and show that if there is sufficient measurement data, the SLAM problem is solvable. Finally, we give the expression of the observation model using the defined notations.

3.1.1 Notations

To mathematically formulate the required outputs listed above, such that math tools can be applied to solve the SLAM problem, we define the following:

- several coordinate frames, including the world frame $\mathcal{F}_w$ and the body-fixed frames $\mathcal{F}_i$ with i representing the time step.

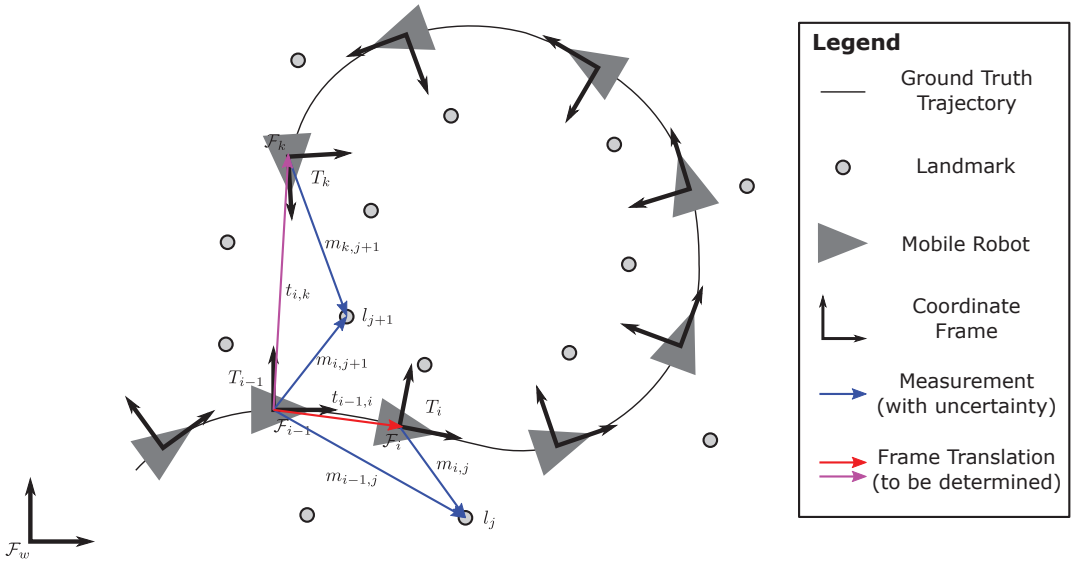


Figure 3.1: Illustration of the SLAM problem.

- the robot poses in world frame $\mathcal{F}_w$, which are expressed as the coordinate transformations from the world frame $\mathcal{F}_w$ to the body-fixed frames $\mathcal{F}_i$. To be specific, each transformation contains two components, the translation vector $t^w_{w,i} \in \mathbb{R}^3$ (the vector translates the origin of the world frame $\mathcal{F}_w$ to the origin of the body-fixed frame $\mathcal{F}_i$ expressed in $\mathcal{F}_w$) and the rotation matrix $R^w_{w,i} \in \mathrm{SO}(3)$ (the rotation between world frame $\mathcal{F}_w$ and body-fixed frame $\mathcal{F}_i$ sharing the same origin), or in a more compact form

$$T^w_{w,i} = \begin{bmatrix} R^w_{w,i} & t^w_{w,i} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix} \in \mathrm{SE}(3), \tag{3.1}$$

  where the symbol $T^c_{a,b}$ stands for transformation from frame a to frame b expressed in frame c and it is similar for the symbols $R^w_{w,i}$ and $t^w_{w,i}$. For simplification, the w symbol in $T^w_{w,i}$ is neglected and T_i is denoted as the pose in world frame $\mathcal{F}_w$ at time step i.

  Similarly, the transformation from body-fixed frame $\mathcal{F}_i$ to some other body-fixed frame $\mathcal{F}_k$ expressed in $\mathcal{F}_i$ is denoted as $T^i_{i,k}$, and $T_{i,k}$ is the simplified case. As a result, if T_i and $T_{i,k}$ are known, then T_k can be computed using

$$T_k = T_{i,k} T_i. \tag{3.2}$$

- the landmark locations l_j in world frame $\mathcal{F}_w$, which are 3-dimensional vectors containing x, y, and z coordinates, and j represents the landmark index.

- the measurements, including distances and bearing angles, of the observed landmarks. The measurements are formulated as 3-dimensional vectors expressed in the current body-fixed frame (the blue vectors in Figure 3.1). They are denoted as $m^i_{i,j}$, which reflect the x, y, and z coordinates from the origin of $\mathcal{F}_i$ to the observed landmark l_j in $\mathcal{F}_i$. For convenience, we denote $m^i_{i,j}$ as $m_{i,j}$, and the following equation shows the transformation from the measurement $m_{i,j}$ to the landmark l_j in world frame $\mathcal{F}_w$

$$\begin{bmatrix} l_j^\top & 1 \end{bmatrix}^\top = T_i \begin{bmatrix} m_{i,j}^\top & 1 \end{bmatrix}^\top. \tag{3.3}$$

  Similarly, the measurements in body-fixed frame $\mathcal{F}_i$ can be related to measurements in some other body-fixed frame $\mathcal{F}_k$ at a different time step. This relation is given by

$$\begin{bmatrix} m_{k,j}^\top & 1 \end{bmatrix}^\top = T_{i,k} \begin{bmatrix} m_{i,j}^\top & 1 \end{bmatrix}^\top. \tag{3.4}$$

3.1.2 General SLAM Principle

Using these definitions, the goals of a SLAM algorithm can be expressed as estimating both T_i at each time step and l_j from the measurements $m_{i,j}$.

The estimating process depends on the relations in (3.2), (3.3) and (3.4), which gives the basic concept of how the SLAM algorithm works. Using (3.4), the SLAM algorithm is able to estimate the frame-to-frame pose transformation if a sufficient number of landmarks is observed in both frames. Specifically, for the 3D case, each pair of measurements can give three independent equations, while in (3.4), there are twelve unknowns. This indicates that in the ideal case, four pairs of measurements are required to give a unique solution of $T_{i,k}$. While in practice, the measurements given by the sensors are noisy, thus the pose transformation $T_{i,k}$ is usually estimated by solving a least square problem or using a SVD method based on a set of measurements. Usually, these methods are applied to obtain the pose transformation between two subsequent frames, denoted as $T_{i-1,i}$ (the red vector in Figure 3.1 represents translation vector $t_{i-1,i}$), since two subsequent frames share most of their observed landmarks. The estimation process of $T_{i-1,i}$ is known as short-term data association in many SLAM frameworks.

After obtaining $T_{i-1,i}$, (3.2) is applied to estimate T_i, assuming the initial pose T_0 is a priori known. The initial pose T_0 is usually assumed to be a 4-by-4 identity matrix (no translation or rotation) if no map information is available beforehand. An alternative is estimating T_0 by comparing the measured data with a prior map.

If T_i is known, (3.3) can be applied to give the estimates of the landmarks in world frame, and if the landmark is also observed at a different pose, this information can conversely enhance the pose estimation at that time step. This is reasonable because the measurements reflect the relations between landmarks and robot poses. Therefore, more accurate landmark locations result in more accurate pose estimations, and vice versa.

The pose and landmark estimating procedure is repeated every time step, when new measurements are obtained. It can be inferred from (3.2), (3.3), and (3.4) that if a sufficient number of landmarks is observed at each time step, the poses and the landmark locations can be calculated from the measurements.

3.1.3 Observation Model

In practice, the sensors are not ideal, so the measurements are noisy in almost all situations. Thus, the observation model is formulated based on (3.3), assuming the measurement noise model is an additive zero-mean Gaussian process $v \in \mathbb{R}^3$ with covariance $V \in \mathbb{R}^{3\times3}$. The covariance matrix V is highly dependent on the choice of the sensor, and can be estimated from the sensor data. In this work, we treated this covariance matrix as a diagonal matrix, and use the same matrix for all the observations. In addition, since this work depends on online datasets for testing and evaluation, the diagonal entries of the matrix are tuned manually. Hereby, the expression of the observation model is given:

$$\begin{bmatrix} m_{i,j}^\top & 1 \end{bmatrix}^\top = T_i^{-1} \begin{bmatrix} l_j^\top & 1 \end{bmatrix}^\top + \begin{bmatrix} v^\top & 0 \end{bmatrix}^\top, \tag{3.5}$$

where the left side of the equation only contains measurements $m_{i,j}$, while the right side is a function of the data to be estimated, namely pose T_i and landmark l_j, and noise v. Also recall that i stands for the time step, and j represents the index of the landmark.

3.2 Factor Graph Construction

It can be seen that in a SLAM problem, tons of poses and landmarks need to be estimated using an even larger number of measurements. As a result, to efficiently organize the measured data and associate it with the corresponding states to be determined, a graph data structure is usually

considered. A graph contains two types of elements: nodes and edges. The nodes represent the data to be determined, and the edges represent the relations between or the conditions on the nodes. For the SLAM problem, the landmark locations and robot poses are considered as nodes, while the measurements are incorporated as edges.

To clearly describe the procedure of constructing the graph, an example graph is abstracted from the SLAM problem in Figure 3.1 and shown in Figure 3.2. For clarity, only three robot poses T_{i-1}, T_i, T_k, two commonly observed landmarks l_j and l_{j+1}, and their relations are plotted in Figure 3.2. In the graph, we denote nodes as circles and edges as rectangles. The first row of Figure 3.2 shows the observed landmark nodes, the third row shows the robot pose nodes at different time steps, and the second row shows the measurements, which relate robot poses to landmarks. In Figure 3.1, the landmark l_j is observed in both frame $\mathcal{F}_{i-1}$ and frame $\mathcal{F}_i$, and the measurements are $m_{i-1,j}$ and $m_{i,j}$, respectively. As a result, we add the edges between pose node T_{i-1} and landmark node l_j with weight $m_{i-1,j}$, and also between T_i and l_j with weight $m_{i,j}$, as shown in Figure 3.2. Note that the graph is underconstrained, because only a few edges are shown in the graph. In practice, more measurement constraints are added to the graph such that a unique solution can be obtained. By adding measurement edges for every pose and landmark node in the SLAM problem, the graph is constructed. Furthermore, since a pose node is created every time step, the constructing operation is incremental.

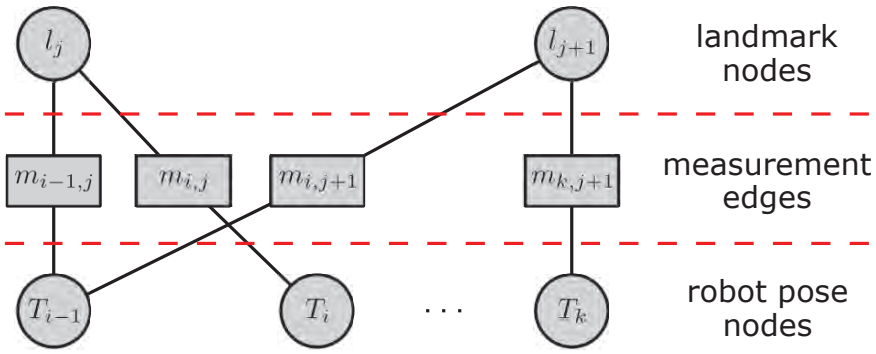


Figure 3.2: An example graph of the SLAM problem in Figure 3.1. The states to be estimated are in circles, whereas the measurements are in rectangles, acting as edges in the graph.

Considering the observation model in (3.5), if we denote

$$\begin{bmatrix} e_{i,j}^\top & 0 \end{bmatrix}^\top = T_i^{-1} \begin{bmatrix} l_j^\top & 1 \end{bmatrix}^\top - \begin{bmatrix} m_{i,j}^\top & 1 \end{bmatrix}^\top, \tag{3.6}$$

then $e_{i,j}$ is a realization of the zero-mean Gaussian process. Thereafter, following the concept in [20] and the definition of Gaussian Probability Density Function (PDF) in [21], the PDF $p(e_{i,j})$ over the variable $e_{i,j}$ can be expressed as

$$p(e_{i,j}) \propto \exp\left(-\frac{1}{2} e_{i,j}^\top V^{-1} e_{i,j}\right). \tag{3.7}$$

Let $p(x|y)$ be a PDF over x conditioned on y, then using this notation and considering (3.6), (3.7) can be rewritten as the PDF $p(m_{i,j}|T_i, l_j)$ over $m_{i,j}$ conditioned on T_i and l_j, which is given by

$$\begin{aligned} p(m_{i,j}|T_i, l_j) &\propto \exp\left(-\frac{1}{2}\left(T_i^{-1}\begin{bmatrix} l_j \\ 1 \end{bmatrix} - \begin{bmatrix} m_{i,j} \\ 1 \end{bmatrix}\right)^\top \begin{bmatrix} V^{-1} & \mathbf{0}_{3\times 1} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix} \left(T_i^{-1}\begin{bmatrix} l_j \\ 1 \end{bmatrix} - \begin{bmatrix} m_{i,j} \\ 1 \end{bmatrix}\right)\right), \\ &= \exp\left(-\frac{1}{2}\begin{bmatrix} e_{i,j}^\top & 0 \end{bmatrix} \begin{bmatrix} V^{-1} & \mathbf{0}_{3\times 1} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} e_{i,j} \\ 0 \end{bmatrix}\right), \\ &= \exp\left(-\frac{1}{2} e_{i,j}^\top V^{-1} e_{i,j}\right), \\ &\propto p(e_{i,j}). \end{aligned} \tag{3.8}$$

However, in SLAM, $m_{i,j}$ is known while T_i and l_j are to be determined, which indicates that the PDF $p(T_i, l_j|m_{i,j})$ over T_i and l_j conditioned on $m_{i,j}$ is required to be maximized. The relation between $p(T_i, l_j|m_{i,j})$ and $p(m_{i,j}|T_i, l_j)$ can be derived using Bayes' Law [21], which is given by

$$p(T_i, l_j|m_{i,j}) = \frac{p(m_{i,j}|T_i, l_j)p(T_i, l_j)}{p(m_{i,j})}, \tag{3.9}$$

where $p(m_{i,j})$ does not depend on T_i or l_j and $p(T_i, l_j)$ is a prior density. Usually, only the prior density of the first pose $p(T_0)$ is known.

Based on the above relation, let X denotes all the nodes to be estimated, i.e. $X = \{T_i, l_j\}$ with i and j representing all the past time steps and all the landmark indices, respectively. Also, let Z denotes all the obtained measurements, i.e. $Z = \{m_{i,j}\}$ for all i and j. Then the following relation can be found:

$$\begin{aligned} p(X|Z) &\propto \prod_i \prod_j p(T_i, l_j|m_{i,j}), \\ &\propto p(T_0) \prod_i \prod_j p(m_{i,j}|T_i, l_j). \end{aligned} \tag{3.10}$$

The above relation motivates the factor graph. To be specific, if we consider $p(T_0)$ and $p(m_{i,j}|T_i, l_j)$ as factors, then $p(X|Z)$ can be seen as a product of all the factors. Therefore, the graph in Figure 3.2 can be drawn in another form using factors and nodes. Considering the measurement edges in Figure 3.2, they connect the poses to the corresponding landmarks given the observation model in (3.5) and the measurements. On the other hand, based on the above analysis, the poses and landmarks can also be linked using factors, e.g., the factor $p(m_{i,j}|T_i, l_j)$ relates T_i and l_j. Accordingly, if we replace all the measurement edges in Figure 3.2 with factors (black squares), Figure 3.3 can be derived. In addition, to also visualize prior information about the initial condition T_0, the prior factor $p(T_0)$ is also included in the factor graph.

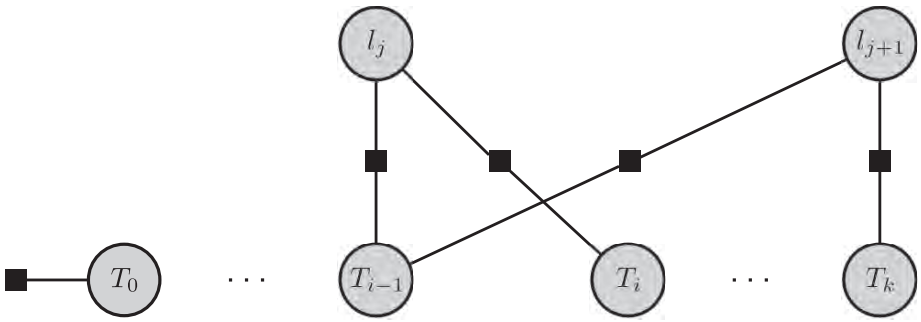


Figure 3.3: Factor graph corresponding to Figure 3.2, where the black squares stand for factors. The factor on T_0 represents the prior factor $p(T_0)$.

From (3.10), it can be observed that to find the best estimates of all the T_i and l_j, $p(X|Z)$ is required to be maximized. Considering (3.8) and (3.10), the following relation can be derived:

$$\begin{aligned} X^* &= \arg\max_X P(X|Z) = \arg\min_X -\log P(X|Z), \\ &= \arg\min_X -\log p(T_0) \prod_i \prod_j p(m_{i,j}|T_i, l_j) \\ &= \arg\min_X -\log p(T_0) - \log \prod_i \prod_j \exp\left(-\frac{1}{2} e_{i,j}^\top V^{-1} e_{i,j}\right) \\ &= \arg\min_X \sum_i \sum_j e_{i,j}^\top V^{-1} e_{i,j}. \end{aligned} \tag{3.11}$$

Note that we omit the prior factor term $p(T_0)$, because usually it is determined beforehand and does not influence the other estimates. Also recall that X contains all the pose and landmark estimates T_i and l_j, and according to (3.8), the value of $e_{i,j}$ is directly determined by T_i and l_j.

The equation (3.11) forms the cost function of an optimization problem, such that iterative optimization methods using Gauss-Newton or Levenberg-Marquardt algorithms can be applied to find the best estimates. Generally speaking, the optimization problem is not convex, so the best estimates given by the iterative optimization methods might be the local optimum but not the global optimum. Furthermore, to obtain reasonable estimates, good initial guesses of the states to be determined are required. In this work, the initial guess for a robot pose is set equally to the estimated pose at the previous time step, since no other prior information of the poses are provided and normally the motion is slow. For the landmarks, the initial guess is given by (3.3). However, instead of using the measurement and the pose at the current time step, the pose estimate and the measurement at the previous one time step are considered, since the current pose is not known at this moment. In addition, the reason that the initial guess of the currently observed landmark can be carried out using the data at the previous time step, is because of the landmark decision method discussed in Chapter 2. Recall that an extracted feature point is considered as a landmark if and only if it can be observed at at least two subsequent time steps.

The program for creating the factor graph is given in the following, which consists of two parts. The first part is mainly responsible for adding pose node into the graph, while the second part is mainly for landmark nodes.

```
// add pose node
void Core::addPose()
{
    if(_step == 0)
    {
        // add prior factor
        _graph.addExpressionFactor(gtsam::Pose3_('x',_step),
            _poses[_step], _noisePrior);
        // insert initial guess for the pose at step 0.
        _initEst.insert(gtsam::Symbol('x', _step), _poses[_step]);
    }
    else
    {
        gtsam::Pose3 odometry, initPose;
        odometry = gtsam::Pose3(); // since no odometry information
                                   // is available, it is kept
                                   // identity.
        initPose = _Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-1)); // use previous pose
                                         // estimate for the
                                         // initial guess of
                                         // the current pose.

        _graph.addExpressionFactor(
            between(gtsam::Pose3_('x',_step-1),
            gtsam::Pose3_('x',_step)), odometry,
            _robustNoiseTrans); // the noise level is set to a
                                // large value since no odometry
                                // information is known.
        _initEst.insert(gtsam::Symbol('x', _step), initPose);
    }
}
```

```
// add landmark nodes
void Core::addLandmarks()
```

```
{
    for(size_t i=0; i<_obs1.size(); ++i)
    {
        if(!_Est.exists(gtsam::Symbol('l', _obs1[i].index)))
        {
            gtsam::Point3 factor_1(_obs1[i].position.x,
                                   _obs1[i].position.y,
                                   _obs1[i].position.z);
            // add factor between pose node and landmark node
            _graph.addExpressionFactor(RGBDFactor_(
                gtsam::Pose3_('x',_step-1),
                gtsam::Point3_('l',_obs1[i].index)), factor_1,
                _robustNoiseObs);
            gtsam::Point3 initPoint = _Est.at<gtsam::Pose3>(
                gtsam::Symbol('x',_step-1)).transform_from(factor_1);
            _initEst.insert(gtsam::Symbol('l', _obs1[i].index),
                initPoint);
        }

        gtsam::Point3 factor_2(_obs2[i].position.x,
                               _obs2[i].position.y,
                               _obs2[i].position.z);
        // add factor between pose node and landmark node
        _graph.addExpressionFactor(RGBDFactor_(
            gtsam::Pose3_('x',_step),
            gtsam::Point3_('l',_obs2[i].index)),
            factor_2, _robustNoiseObs);
    }
}
```

However, both the Guass-Newton and Levenberg-Marquardt algorithms are batch optimization algorithms, which inherently do not suit the incremental property of SLAM, as discussed in Chapter 1. This problem is further explained in the following. When new measurements are obtained at the next time step, new elements are added to the cost function, e.g. the ranges of i and j in (3.11) increase. Then, batch optimization algorithms recompute a solution for all the states to be determined, although in most of the cases, the newly obtained measurements only influence the last few pose and landmark estimates (assuming loop closure does not happen). Thus it is obvious that updating all the states is not computationally efficient. To solve this problem, the iSAM algorithm proposed by [16] is adopted to construct and optimize the factor graph. For implementation, the iSAM algorithm is implemented in the C++ library GTSAM [20], and we mainly depend on this library to enable incremental optimization of the problem. The iSAM algorithm can be called using the following code.

```
    _isam->update(_graph, _initEst);
    _isam->update();
    _Est = _isam->calculateEstimate();
```

The full implementation of the algorithm is given in Appendix A.2.

3.3 Outlier Rejection

An outlier is a statistic concept defined as a data point that differs significantly from other observations. Outliers can occur by chance in any distribution, but they often indicate either measurement errors or that the population has a heavy-tailed distribution [22]. In this work, the

outlier in the measurements are defined based on the error item $e_{i,j}$ in (3.6). Specifically, if $e_{i,j}$ is larger than a threshold, then the corresponding measured data $m_{i,j}$ is considered as a outlier in the measurements. The threshold is a tuning parameter determined by users.

Several approaches for outlier rejection exist, such as M-estimation. M-estimation is the statistical procedure of evaluating an M-estimator on a data set, while the M-estimators are a broad class of estimators for parametric models that are calculated through maximization or minimization of a certain cost function [23]. In this work, the general M-estimator can be defined as

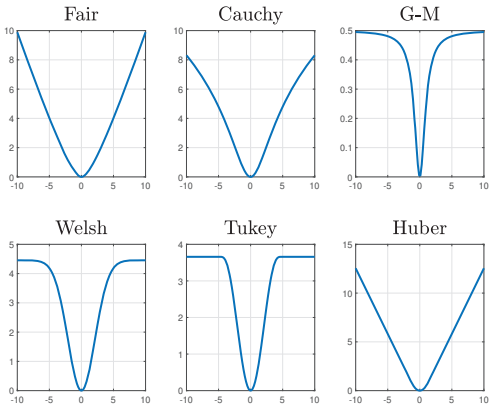
$$X^* = \arg\min_X \sum_i \sum_j \rho(e_{i,j}), \tag{3.12}$$

where X contains all the variables to be determined, including robot poses and landmarks, $e_{i,j}$ is a three dimensional vector computed using (3.6), and the ρ-function is a user's choice depending on different situations.

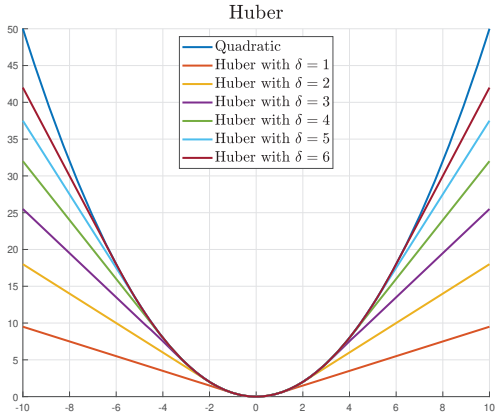
In this sense, the estimator in (3.11) can be seen as a special case of a M-estimator using a quadratic function as ρ-function. When rejecting outliers in the measured data using the M-estimation method, the ρ-function is modified to functions other than quadratic ones, so that outliers do not dominate the solution [21]. The choice of the ρ-function is further discussed in the following section.

3.3.1 The choice of ρ-function for the M-estimator

Many M-estimators exist, depending on different choices of the ρ-function. A few of the commonly used ρ-functions are shown in Figure 3.4a.



(a) Examples of the ρ-functions used in outlier rejection



(b) Huber loss function with different settings of δ, comparing to quadratic function (blue)

Figure 3.4

As we can see in Figure 3.4a, the Cauchy, G-M, Welsh, and Tukey functions are not convex, thus they are not able to guarantee unique solutions [24]. In [24], the author also claims that very rarely the Huber function has been found to be inferior to other ρ-functions. So in this work, the Huber function given by (3.13) is considered as the ρ-function for the estimator in (3.12).

$$\rho(e_{i,j}) = \begin{cases} \frac{1}{2} e_{i,j}^\top V^{-1} e_{i,j}, & \text{if } |V^{-1} e_{i,j}| \leq \delta, \\ \delta |V^{-1} e_{i,j}| - \frac{1}{2}\delta^2, & \text{otherwise,} \end{cases} \tag{3.13}$$

where $V \in \mathbb{R}^{3\times 3}$ is the covariance matrix, and δ is a tuning constant. The symbol $|\cdot|$ stands for L_1-norm of a vector. To show the effect of the M-estimator using the Huber function as its ρ-function, the one-dimensional Huber functions with different settings of the tuning constant, comparing to quadratic function, are shown in Figure 3.4b. It can then be observed that the smaller the tuning constant is, the heavier the penalty of the outlier. In practice, the tuning

constant is chosen approximately to equal the outlier threshold [25]. The C++ implementation for the outlier rejection is given below, while the full implementation can be found in Appendix A.2.

```
// set the diagonal covariance matrix
_noiseObs = gtsam::noiseModel::Diagonal::Sigmas(_sigmasObs);
// choose m-estimator: Huber
_mEstimator = gtsam::noiseModel::mEstimator::Huber::Create(
    _delta);
// create robust noise model
_robustNoiseObs = gtsam::noiseModel::Robust::Create(
    _mEstimator, _noiseObs);
```

In the next section, a Matlab simulation is carried out to show the effect of the Huber function in outlier rejection.

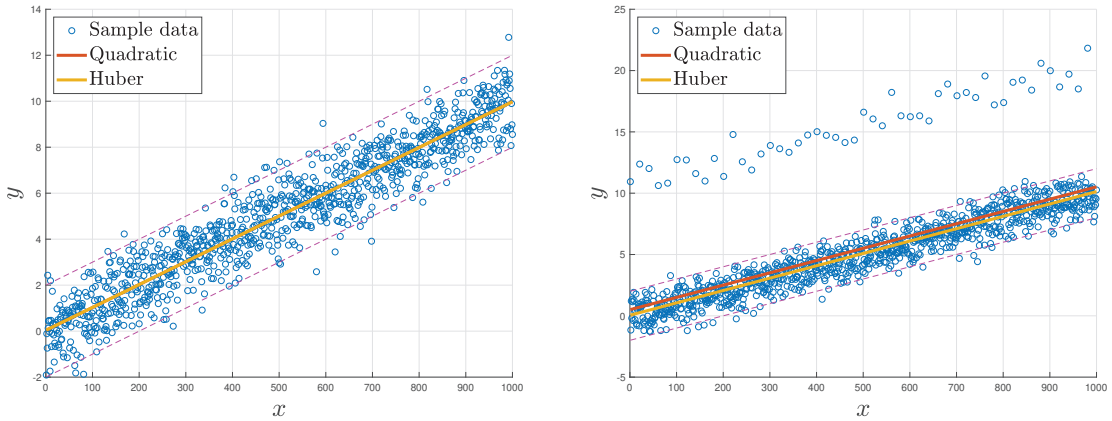
3.3.2 Matlab Simulation Results

To show the effect of the selected ρ-function on outlier rejection, a 2D line fitting problem is considered as an example. The line model is given by

$$y = kx + d, \tag{3.14}$$

where k and d are the parameters to be estimated, and the true values are 0.01 and 0.0, respectively. A sample data of 1000 pairs of (x, y) is fed to the estimators using a quadratic and a Huber function to find the best fits of k and d. The results are shown in Figure 3.5 and Table 3.1. The tuning constant of the Huber function is set to 2 for both cases, since in the case that no outliers exist (Figure 3.5a), the data are approximately bounded by the true value plus or minus 2 (the magenta lines in Figure 3.5b).

Figure 3.5a shows the line fitting result when no outliers in the sample data. The result indicates that when no outliers exist in the measurements, the estimators using the quadratic function and the Huber function give similar results. The same conclusion can also be drawn from Table 3.1. On the other hand, Figure 3.5b shows the result when outliers exist. It can be seen that in this case, the fitted lines given by the estimators have an offset. From Table 3.1, it can be concluded that the estimator using the Huber function outperforms the one using the quadratic one by rejecting the outliers in the sample data.



(a) Estimate results without outliers

(b) Estimate results with outliers

Figure 3.5: Outlier rejection results

Table 3.1: Line fitting result comparison

Cost function	Quadratic	Huber
Error in k (no outliers)	-0.0001	-0.0001
Error in d (no outliers)	0.0359	0.0380
Error in k (with outliers)	-0.0001	-0.0001
Error in d (with outliers)	0.4998	0.0465

In the next section, we apply the graph optimization and outlier rejection method to a real world RGB-D dataset provided by TU Munich [26], and show the outlier rejection significantly increases the overall estimation accuracy.

3.4 Experimental Results

The TU Munich datasets [26] contain the color and depth images of a Kinect sensor along the ground-truth trajectory of the sensor. The data was recorded at full frame rate (30 Hz) and sensor resolution (640x480). The ground-truth trajectory was obtained from a high-accuracy motion-capture system with eight high-speed tracking cameras (100 Hz). In addition, an evaluation tool for measuring the quality of the estimated camera trajectory of visual SLAM systems is also provided. The evaluation tool compares the estimated trajectory with the ground-truth trajectory obtained from the motion-capture system, and outputs the result in a plot. When performing the comparison, the evaluation tool transforms (rotates and translates) the whole estimated trajectory to minimize the translational Root Mean Square Error (RMSE). Based on the transformed trajectory, it not only gives the RMSE, but also some other numerical evaluation results, including the mean, median, maximal, and minimal translational errors, as well as the standard deviations.

The purpose of this test is to show that the proposed factor graph optimization algorithm is able to give a trajectory that agrees with the ground-truth. Also, we want to show that the outlier rejection is essential for the estimation accuracy. To this end, the same sequence of the RGB-D images is fed into the proposed system twice, but with different outlier rejection settings.

For this test, the TU Munich dataset fr1/floor is considered. This sequence contains a simple sweep over the wooden floor in the office. The floor contains several knotholes which are easy to track for visual sensors. Some of the color images as well as the corresponding depth images in this dataset are shown in Figure 3.6. The parameters for both the cases are set according to Table 3.2, where the only difference is the value of the outlier rejection parameter (Noise.constant). Some of these parameters are illustrated in the following.

- Noise.constant is the outlier rejection parameter (the tuning constant δ in (3.13)). It influences the effect of outlier rejection. From Figure 3.4b and (3.13), it can be inferred that infinite δ value for the Huber function can result in a quadratic function. Therefore, in Settings 1, the outlier rejection parameter is set to a large value (10000) to eliminate the outlier rejection's effect. On the other hand, a small value is set for that parameter in Settings 2 to reject the outliers in the measurements.

- Noise.observation is the measurement uncertainty. It is a 3-dimensional vector which determines the diagonal entries of the covariance matrix V in (3.13). The values of the vector are dependent on the choice of the sensor. While this work depends on online real-world datasets instead of a real Kinect sensor, thus the values are tuned manually which may be different from the actual values.

- ORBExtractor.filter is the matches filter ratio and Depth.bound is the depth boundary for selecting landmarks from raw measurements, as described in Chapter 2. The matches filter ratio is a parameter in the range of $[0, 1]$ which influences the filtering result of feature matching (an example is shown in Figure 2.5). A larger value of this ratio results in more

left matches, but also more mismatches. While a smaller ratio can filter out most of the mismatches but the number of the left matches is also less. The depth boundary parameter keeps the landmarks whose depth measurement are within the boundary.

A complete description of the parameters can be found in Appendix B.

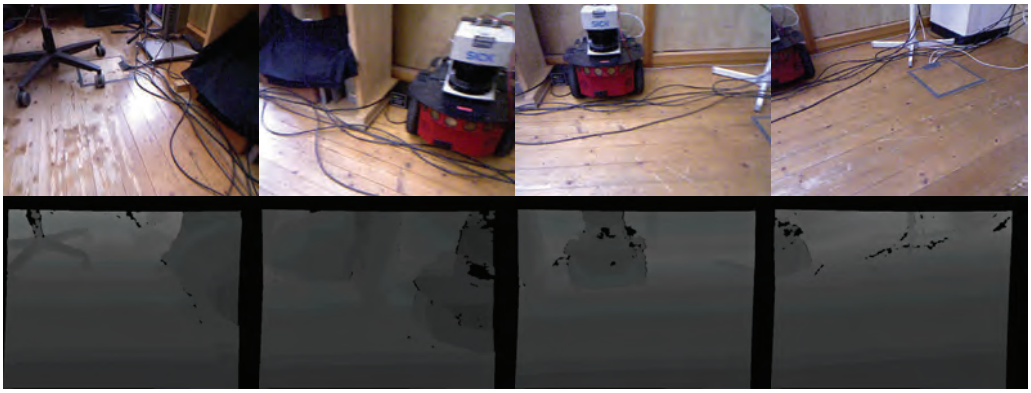


Figure 3.6: Some of the images in dataset fr1/floor

Table 3.2: Settings for the outlier rejection tests

Parameter	Settings 1	Settings 2
Camera.fps	10 Hz	10 Hz
ORBExtractor.numFeatures	1000	1000
ORBExtractor.filter	0.6	0.6
Depth.bound	[0, 20000] (0-4 m)	[0, 20000] (0-8 m)
Noise.observation	[0.01, 0.01, 0.015]	[0.01, 0.01, 0.015]
Noise.constant	10000	0.8

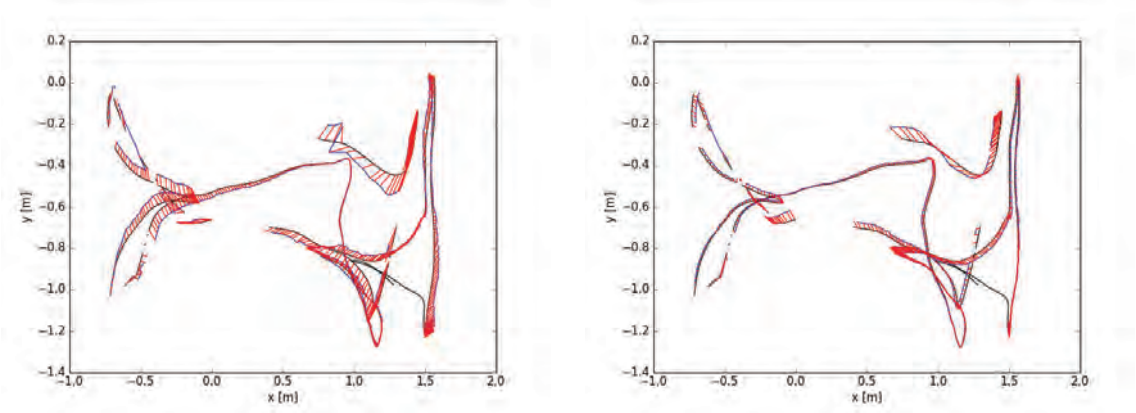
The estimated trajectories given by the two tests are evaluated by the TU Munich tool, and the results are shown in Figure 3.7 and Table 3.3. In Figure 3.7, the trajectories estimated using the settings in Table 3.2 are plotted separately, where the black line stands for the ground-truth obtained from the motion-capture system, the blue line represents the estimated trajectory, and the red lines visually show the difference between the estimated trajectory and the ground-truth. It can then be observed that both the trajectory estimations agree with the ground-truth, yet the estimation using Settings 2 is more accurate than that using Settings 1. The same conclusion can also be drawn from Table 3.3, where several error quantities are computed and listed. These quantities show that the estimation error decreases by approximately 30% using Settings 2, comparing to that using Settings 1.

Table 3.3: Estimated trajectory error comparison

	RMSE	Mean	Median	STD	Min.	Max.
Settings 1	0.094m	0.072m	0.056m	0.061m	0.004m	0.256m
Settings 2	0.065m	0.049m	0.043m	0.042m	0.004m	0.179m
Percentage	30.9%	31.9%	23.2%	31.1%	0.0%	30.1%

Note that STD is for standard deviation.

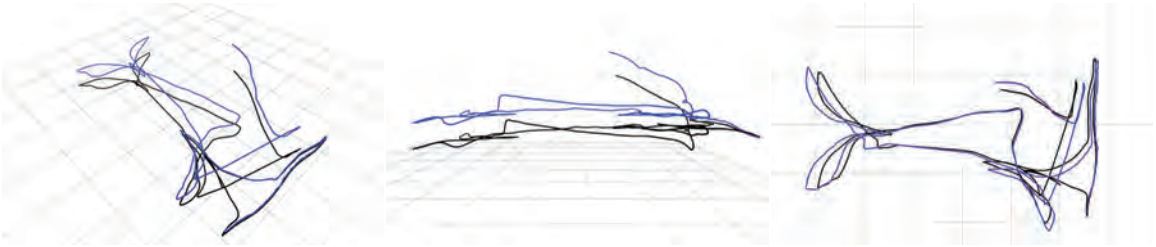
Figure 3.7 only shows the top views of the trajectories, which makes the drift problem hardly visible in the plots. To clearly show the drift of the estimated trajectory, the 3D trajectory plot is given in Figure 3.8, using a ROS visualization tool named Rviz. In that plot, the starting position and orientation of the estimated trajectory are aligned (approximately) with the corresponding



(a) Estimated trajectory using Settings 1 in Table 3.2. (b) Estimated trajectory using Settings 2 in Table 3.2

Figure 3.7: The estimate accuracy increase by outlier rejection. Note that parts of the ground truth trajectory are missing, so no error evaluation for those poses.

point on the ground-truth. By doing this, the drift of the estimated trajectory becomes visible. It can be observed from Figure 3.8 that the estimated trajectory drifts severely in the z-axis over time.



(a) Oblique view (b) Front view (c) Top view

Figure 3.8: the 3D plot of the trajectories, where the black curve represents the ground-truth trajectory given by the motion capture system, and the blue curve represents the trajectory estimated by the proposed system using Settings 2 in Table 3.2.

3.5 Summary

This chapter proposes the construction of the factor graph using the measurements given by the method described in Chapter 2. The outlier rejection method depending on the Huber function is then discussed. The simulation results show that the estimator using a Huber function is capable of rejecting the outliers in the measurements. Thereafter, we implement and test this algorithm in a ROS environment using one of the TU Munich RGB-D datasets [26]. The test results indicate that the outlier rejection increases the overall accuracy of the trajectory estimate, but the trajectory still drifts. Thus in the next chapter, the loop closure function is discussed to compensate the drift in the trajectory estimate for long-term tracking.

Chapter 4

Loop Closure

In the previous chapter we have shown that without a loop closure function, the estimated trajectory drifts over time. Thus in this chapter, the loop closure function is discussed with details for drift compensation. Loop closure consists of two steps: detection and correction. The detection step is responsible for correctly detecting the fact that the robot returns to a known place, while the correction's job is to use the detection results and correct the estimation of both trajectory and map. In the following sections, we discuss the detection and correction steps separately, and show that a pose graph is required for loop closure correction. At last, experimental results are given to show the performance of loop closure.

4.1 Loop Closure Detection

Loop closure requires an efficient and robust loop candidate detection mechanism to recognize the fact that the robot returns to some already visited place. For this purpose, the Bag-of-Words (BoW) model is often considered [6, 27, 28, 29]. In computer vision, the BoW model is defined as the histogram representation of an image based on independent features. In other words, the BoW model represents an image as a value vector that counts the occurrence of features within the image. In the BoW model, the image features are treated as visual words. Accordingly, an image is called a "bag" of words, since only the occurrence of the words matters, but not the order or the position of the words.

The BoW model functions based on a dictionary of the visual words. Usually, the dictionary is created from a large number of images, by extracting frequently appearing features on these images and clustering similar features into one visual word. The working principle of a BoW model is briefly illustrated in Figure 4.1, where the left part of the figure shows three images with different features, the middle part shows the histograms for each image according to an example dictionary, and the right part shows the corresponding value vectors. It can be seen that different images result in distinct histograms based on the same dictionary. Thus, by comparing the histograms of the images, similar images can be found. For efficiency consideration, the histograms are represented as value vectors. As a result, the comparison of the histograms is equivalent to computing the distances between the vectors, where the distance is defined as the L_1-norm of the two vectors, and similar images are the images with short distance.

In the SLAM problem, every obtained color image is abstracted as a BoW model according to a dictionary. Then the BoW model is stored and compared with other BoW models in the database. Since the BoW models are value vectors, the comparison can be much faster than matching the color images directly. At the end, the loop closure candidates can be found. In practice, the loop candidates are further validated before being applied in loop closure correction. For example, if the number of the matched features in the two loop closure candidates is lower than a boundary, then the loop closure candidates are discarded.

When using BoW models for loop closure detection, the quality of the dictionary is essential

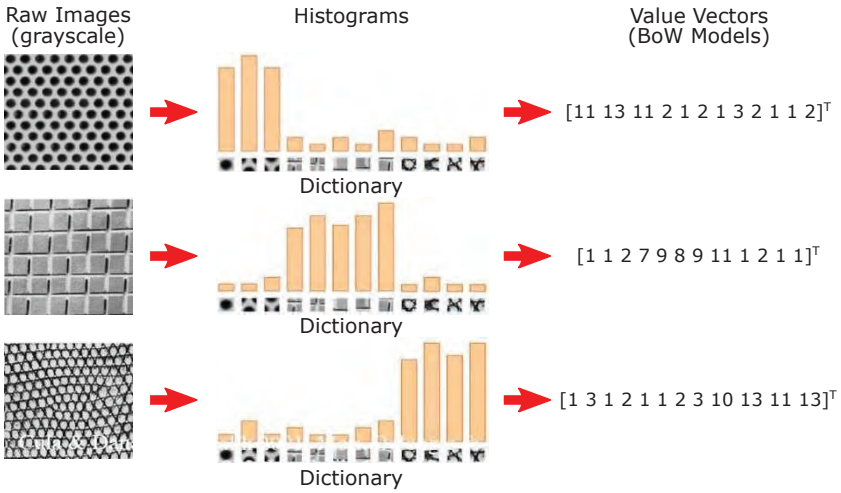


Figure 4.1: A brief working principle illustration of BoW model.

to the performance. Usually, the dictionary is required to be trained in an environment with similar working conditions to the real case. To make sure the BoW model works properly in more scenarios, a large or even huge dictionary is usually needed, like the case for ORB-SLAM. This issue limits the performance and the robustness of loop closure detection.

Due to the above limitation, in this work the iBoW-LCD algorithm, proposed by [27], is utilized as our loop closure detection method. iBoW-LCD makes use of an incremental Bag-of-Words scheme, avoiding any pre-training procedure [27]. As we discussed above, the BoW model works depending on a pre-trained dictionary, which is not needed for iBoW-LCD. Instead, iBoW-LCD trains and updates a dictionary in real-time. This property breaks the limitation of traditional BoW model-based loop closure approaches we mentioned above, while it costs more computational resources. The C++ implementation of iBoW-LCD is given in the following, while the full implementation of the loop closure detection can be found in Appendix A.3.

```
void LoopDetector::ibowMethod(const std::vector<cv::KeyPoint>& kp,
    const cv::Mat& des)
{
    ibow_lcd::LCDetectorResult result;
    _lcdetector->process(_step, kp, des, &result);

    if(result.status == ibow_lcd::LC_DETECTED && result.inliers > 0)
    {
        _isLoop = true;
        _candidate = _steps[result.train_id];
    }
    else
    {
        _isLoop = false;
    }
}
```

After detecting a pair of loop closure candidates, this information is utilized for loop closure correction, which is discussed in the following section.

4.2 Loop Closure Correction

The loop closure correction is performed based on the factor graph constructed as described in Chapter 3. To illustrate the loop closure correction procedure, the SLAM example in Figure 3.1 is again considered. When a loop is detected, it can be seen as common landmarks are observed at two greatly different time steps. In Figure 3.1, the landmark l_{j+1} is observed by the robot at both T_{i-1} and T_k, so there is a loop closure between $\mathcal{F}_{i-1}$ and $\mathcal{F}_k$. Presenting this loop closure in the factor graph, it can be seen as an extra factor connecting T_{i-1} and l_{j+1}, as shown in Figure 3.3. It then can be imagined that this factor influences a large set of states in the factor graph, such as all the robot poses between T_{i-1} and T_k, as well as the landmarks observed along this part of the trajectory. Updating all these states in the factor graph is generally not feasible in real-time, so in practice, the loop closure correction process is usually performed on a pose graph [6, 13]. A pose graph is also a graph based data structure, but instead of storing both the robot poses and the landmark locations, it only stores the robot poses and abstracts the landmark measurements as edges between the poses.

To illustrate the difference and relation between a pose graph and a graph with both landmarks and poses, the graph in Figure 3.3 is considered as an example. It can be observed from Figure 3.3 that the landmark node l_j has connections to pose nodes T_{i-1} and T_i. Therefore, if the landmark node l_j is considered to be deterministic, then the two factors between l_j and the other two pose nodes T_{i-1} and T_i can be transformed to one factor, which links T_{i-1} and T_i directly. Similarly, the same transformation is applied to l_{j+1}, T_{i-1}, and T_k. After doing this, the corresponding pose graph can be generated, which is shown in Figure 4.2. It can been seen in Figure 4.2 that the factor (red square) between T_{i-1} and T_k represents the loop closure constraint, which links two poses at two greatly different time steps.

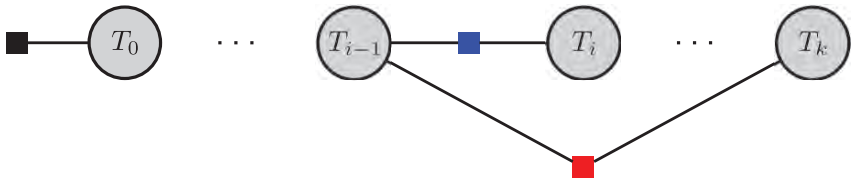


Figure 4.2: The corresponding pose graph to the graph in Figure 3.3, by considering landmarks as deterministic states and only updating pose nodes in Figure 3.3.

Considering (3.2), it can be inferred that the loop closure factor (the red square in Figure 4.2) can be determined by the pose transformation between the two poses. By using a pose graph for loop closure correction, the number of the states to update decreases dramatically, since all the landmark nodes are fixed and only the robot poses are updated. Therefore, when a loop closure is detected, the factor graph optimization algorithm is able to update the states in real-time. The C++ code for adding a loop closure factor to the graph is given in the following. Note that the correction happens in the graph construction process.

```
// add loop closure factor
void Core::addLoopFactor()
{
    _graph.addExpressionFactor(between(
        gtsam::Pose3_('x',_loopIdx1),
        gtsam::Pose3_('x',_loopIdx2)),
        _loopFactor, _robustNoiseLoop);
    _isLoop = false;
}
```

However, due to the incremental property of SLAM, the problem of constructing the pose graph from the full factor graph (the graph containing both poses and landmarks) is still not solved. So

in the next section, we describe the procedure of the pose graph construction, using a node culling mechanism.

4.3 Pose Graph Construction

As we have claimed in the previous section, a pose graph is necessary for loop closure correction. So this section introduces a node culling mechanism to create the required pose graph in real-time. The node culling mechanism is able to remove landmark nodes incrementally at each time step, and also store the removed landmarks for sparse map generation. By doing this, a pose graph can be constructed automatically from the original graph.

The idea is for some fixed horizon, the node culling mechanism removes all the landmark nodes in the graph that only connect to the first pose node in the horizon, and stores the relative locations of these landmarks with respect to that pose. At the next time step, this procedure is repeated. By doing this again and again, only the landmark nodes within the horizon are kept in the graph, while the other landmarks are stored separately. When performing loop closure correction, the pose nodes in the remained factor graph are updated. Accordingly, the landmarks are also updated, because their locations in the world coordinates are determined by both the pose nodes and the stored relative location to these poses. This procedure is visualized in Figure 4.3, where for simplification, the horizon is set to 3 frames.

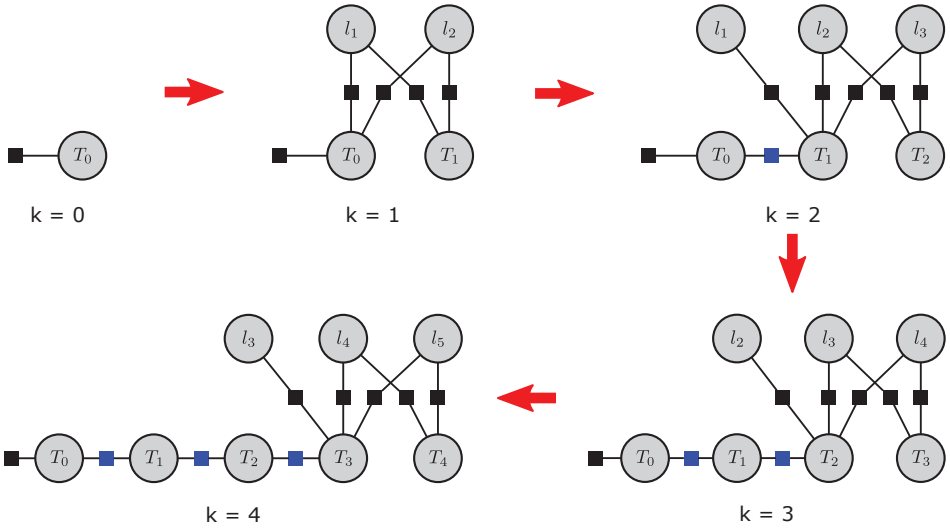


Figure 4.3: The node culling procedure with the horizon parameter equals 3.

In Figure 4.3, at the first time step $k = 0$, no landmark nodes are added into the graph. This is because we only consider features observed in at least two different camera views as landmarks, as described in Chapter 2. Thus at time step $k = 1$, when feature matching information is available, landmarks start to be added. At time step $k = 3$, the graph has met the condition for the node culling action, since the horizon is set to 3. So the node culling mechanism removes the two factors connecting T_0 and l_1, l_2, and replace them by a new factor (the blue square) connecting T_0 and T_1 directly. This procedure is repeated for time steps $k = 3$ and $k = 4$. At $k = 4$, a form of the pose graph is constructed with the landmarks observed only at last two poses.

The procedure of the node culling for implementation is illustrated in the following.

1. Obtain all the factors connected to the last pose in the horizon. This operation creates the initial set $\mathcal{S}$ of factors to be removed;

2. Store the landmarks that has a connection only to the last pose in the horizon;

3. Remove the loop factors from the set $\mathcal{S}$, such that the loop factors are kept in the graph;

4. Remove the factors in the set $\mathcal{S}$, and replace them with new factors connecting poses.

5. The above procedure is repeated at the next time step.

The C++ implementation of the above procedure is given as follows.

```
// remove landmark nodes
void Core::removeLandmarks(const unsigned int& skipSize)
{
    gtsam::VariableIndex variable_index = _isam->getVariableIndex();
    gtsam::VariableIndex::Factors factors_1 =
        variable_index[gtsam::Symbol('x',_step-skipSize)];
    gtsam::VariableIndex::Factors common_factors;

    // store landmarks
    std::vector<gtsam::Point3> landmarks;
    for(size_t i=1; i<_idx; ++i )
    {
        if(!_Est.exists(gtsam::Symbol('l',i)))
            continue;

        gtsam::VariableIndex::Factors factors_l =
            variable_index[gtsam::Symbol('l', i)];

        if(factors_l.size() == 1)
        {
            landmarks.push_back(_Est.at<gtsam::Point3>
                (gtsam::Symbol('l',i)));
        }
    }
    _landmarks.push_back(landmarks);

    // keep all the pose factors including loop factors
    for(size_t i=0; i<_step; ++i)
    {
        if(_step == skipSize)
            continue;
        else if(i <= _step-skipSize+1 && i >= _step-skipSize-1)
            continue;

        gtsam::VariableIndex::Factors factors_2 =
            variable_index[gtsam::Symbol('x',i)];
        // find all the poses connected to the first pose within
        // the horizon.
        std::set_intersection(factors_1.begin(), factors_1.end(),
            factors_2.begin(), factors_2.end(),
            std::back_inserter(common_factors));
    }

    std::sort(common_factors.begin(), common_factors.end());
    std::set_difference(factors_1.begin(), factors_1.end(),
        common_factors.begin(), common_factors.end(),
        std::back_inserter(_factorsToRemove));
```

```
    // start removing current factors and adding new factors
    if(_step == skipSize)
    {
        gtsam::Pose3 priorFactor(_Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-skipSize)));
        _graph.addExpressionFactor(gtsam::Pose3_('x',_step-skipSize),
            priorFactor, _noisePrior);
    }
    else
    {
        gtsam::Pose3 processFactor_1 =
            _Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-skipSize-1)).between(
            _Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-skipSize)));
        _graph.addExpressionFactor(between(
            gtsam::Pose3_('x',_step-skipSize-1),
            gtsam::Pose3_('x',_step-skipSize)),
            processFactor_1, _robustNoisePose);
    }

    gtsam::Pose3 process_factor_2 =
        _Est.at<gtsam::Pose3>(
        gtsam::Symbol('x',_step-skipSize)).between(
        _Est.at<gtsam::Pose3>(
        gtsam::Symbol('x',_step-skipSize+1)));

    _graph.addExpressionFactor(between(
        gtsam::Pose3_('x',_step-skipSize),
        gtsam::Pose3_('x',_step-skipSize+1)),
        process_factor_2, _robustNoisePose);
}
```

With the node culling mechanism, the pose graph can be created using the original factor graph in real-time. The full implementation of the loop closure correction can be found in Appendix A.2. In the next section, the node culling mechanism and the loop closure function is tested using the same dataset in Chapter 3 to show the performance of drift compensation.

4.4 Experimental Results

The purpose of this test is to show that the proposed system with the loop closure module can compensate the trajectory drift over time, which greatly increases the overall estimation accuracy.

As discussed in the previous section, the same dataset used in Chapter 3 (see Figure 3.6) is considered again. This sequence is applied to the proposed system with the loop closure and the node culling mechanism discussed in this chapter, with the parameters following Settings 3 in Table 4.1. For comparison, Settings 2 in Table 3.2 is copied to this table. Note that the loop closure parameters are kept empty in Settings 2, because the loop closure module is disabled in that case. The loop closure parameters are illustrated in the following, while the effects of all the other parameters are described in Appendix B.

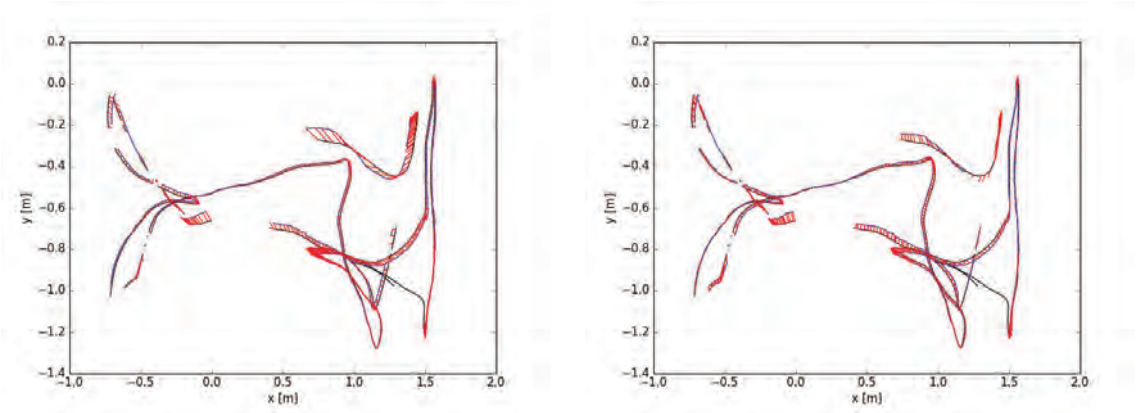
- ISAM2.horizon is the node culling horizon, which determines the number of image frames for graph node removing, as described in the previous section.

- LCDetector.p is the number of the images for the dictionary. It is a parameter for iBoW-LCD [27]. Recall that iBoW-LCD algorithm creates a dictionary in real-time, the image number of dictionary parameter determines the number of images for creating the dictionary, or in other words, the size of the dictionary. While running iBoW-LCD, the dictionary is maintained in real-time by automatically adding and removing BoW models into and from the dictionary.
- LCDetector.min_inliers is another parameter for iBoW-LCD. This parameter is a quantity to reflect the similarity of two images to be considered as loop closures. A higher value of this parameter can result in a more robust loop closure candidate. However, if this parameter is too large, there might be no loop closure found.

Table 4.1: Settings for the loop closure tests

Parameter	Settings 2	Settings 3
Camera.fps	10 Hz	10 Hz
ORBExtractor.numFeatures	1000	1000
ORBExtractor.filter	0.6	0.6
Depth.bound	[0, 20000] (0-4 m)	[0, 20000] (0-4 m)
Noise.observation	[0.01, 0.01, 0.015]	[0.01, 0.01, 0.015]
Noise.constant	0.8	0.8
ISAM2.horizon	-	50
LCDetector.numFeatures	-	1500
LCDetector.p	-	250
LCDetector.min_inliers	-	60

To compare the results, the estimated trajectory is also evaluated using the TU Munich evaluation tool, and the top view of the estimated trajectory as well as the ground-truth is given in Figure 4.4. As usual, the black line represents the ground-truth, the blue line represents the estimation, and the red lines present the difference between them. From Figure 4.4, the improvement in estimation accuracy is not very clear. This is probably because the drift mainly happens in the z-axis (see the front view in Figure 3.8 for reference), which is not apparent in the top view. To show the compensation of the trajectory drift, the 3D plots are also given in Figure 4.5, where from the front view, it can be seen that the drift found in the front veiw of Figure 3.8 has been greatly compensated.



(a) without node culling or loop closure.

(b) with node culling and loop closure.

Figure 4.4: Top views of the estimated trajectories.

To numerically verify the improvement, the translational errors are calculated and given in Table 4.2, from which the improvement in the estimation accuracy is clearly shown. It can then be concluded that the loop closure function decreases the overall estimation error.

(a) Oblique view (b) Front view (c) Top view

Figure 4.5: 3-dimensional trajectory, where the black curve represents the ground truth trajectory given by motion capture system, and the blue curve is the estimated trajectory given by the proposed method.

Table 4.2: Estimated trajectory error comparison

	RMSE	Mean	Median	STD	Min.	Max.
no loop closure	0.065m	0.049m	0.043m	0.042m	0.004m	0.179m
with loop closure	0.030m	0.027m	0.029m	0.013m	0.003m	0.088m
Percentage	53.8%	44.9%	32.6%	69.0%	25.0%	50.8%

Note that STD is for standard deviation.

4.5 Summary

This chapter discusses the two steps needed in loop closure, namely the detection and the correction step. For the detection step, a widely used technique, the bag-of-words model, is reviewed. For the correction step, the necessity of using a pose graph is discussed. Thereafter, the construction method of the pose graph developed in this work is proposed. Finally, experimental results show that the long-term tracking drift we found in Chapter 3 has been greatly decreased. However, there is no dense map generated until now. Thus in the next chapter, the dense mapping method in this work is presented.

Chapter 5

Dense Mapping

In the previous chapter we have shown that after loop closure, the drift in the trajectory estimation is compensated. However, dense mapping is not included until now. Thus, in this chapter, we introduce the dense mapping module, starting by introducing a volumetric map representation named Octomap. Followed by introducing the mapping procedure using the raw sensor data and estimated trajectory. At the end, we show experimental results.

5.1 Octomap Representation

The landmark-based SLAM algorithm is able to estimate both the robot poses and landmark locations simultaneously, and all the landmarks compose a sparse map. However, sparse maps have limitations in performing tasks such as path planning, as claimed in Chapter 1, because sparse maps drop most of the sensor data and only keep a small number of feature points. For the purpose of path planning, dense map representations are required.

Among the dense map representations, Octomap [30] is a mapping approach based on octrees, where an octree is a hierarchical data structure for spatial subdivision in 3D. Octomap uses probabilistic occupancy estimation and explicitly represents not only occupied space, but also free and unknown areas (see Figure 5.1). Furthermore, Octomap is able to update the map representation efficiently and models the sensor data consistently while keeping the memory requirement at a minimum [30].

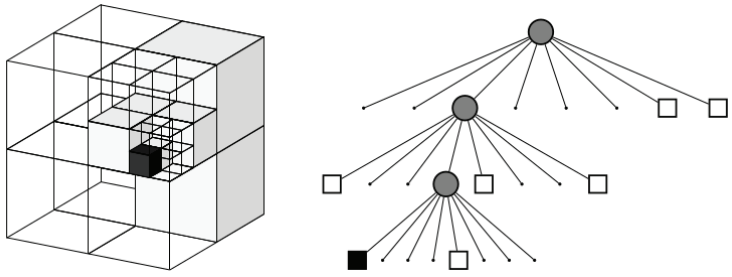


Figure 5.1: Example of an octree storing free (shaded white) and occupied (black) cells. The volumetric model is shown on the left and the corresponding tree representation on the right [30].

Due to the octree structure, the size of the minimal cell can be set with different metric values to give multiple resolution Octomaps.

5.2 Octomap Construction

The RGB-D sensor adopted by this work provides both color image and depth image at each time step. Thus naturally it provides a point cloud for each frame, by aligning the corresponding pixels in both images. Then the mapping process can be described as projecting the points in sensor coordinates to the points in world coordinates. Clearly, they are related by the poses T_i at each

time step, and the relation is also given by (3.3). The difference is that for dense mapping, the $m_{i,j}$ no longer only represent the landmark positions in the sensor coordinates, but all the pixels with depth information.

However, as we claimed in Chapter 1, simply projecting the raw data can only result in a point cloud map, which is highly inefficient. Therefore, the projected point cloud, or the point cloud with respect to the world coordinates, is further processed to generate an Octomap.

Recall that Octomap is a probabilistic mapping approach, because it computes the probabilities of each cell to be occupied, and a threshold on the occupancy probability is applied to give a deterministic volumetric map. The occupancy probability $P(n|z_{0:k})$ for cell n, given the current and the past sensor measurements $z_{0:k}$, is estimated according to [30]

$$P(n|z_{0:k}) = \left(1 + \frac{1 - P(n|z_k)}{P(n|z_k)} \frac{1 - P(n|z_{0:k-1})}{P(n|z_{0:k-1})} \frac{P(n)}{1 - P(n)}\right)^{-1}, \tag{5.1}$$

where k represents the time step, $P(n|z_k)$ the probability of cell n to be occupied given only the current measurement z_k, $P(n|z_{0:k-1})$ the previous probability estimate, and $P(n)$ the prior occupancy probability. When no prior map information is given, the term $P(n)$ is assumed to be 0.5. Also, at time step 0, no previous probability estimates exist, so the term $P(n|z_{0:k-1})$ is neglected.

In case that more than one measurement is obtained at the current time step, (5.1) is modified to

$$P(n|z_{0:k}) = \left(1 + \prod_i \frac{1 - P(n|z_{k,i})}{P(n|z_{k,i})} \frac{1 - P(n|z_{0:k-1})}{P(n|z_{0:k-1})} \frac{P(n)}{1 - P(n)}\right)^{-1}, \tag{5.2}$$

where the sub-index i denotes the i-th measurement at the current time step.

The mapping procedure is briefly illustrated in Figure 5.2. For clarity, we adopt a 2D scenario for illustration, but the main result can be expanded to 3D cases. In that figure, the mesh grid in the first row illustrates the scenario, where the blue curve represents the surface of an obstacle in the environment, the black square stands for the position of the sensor, and the red lines are the measurements.

For simplification, the sensor is assumed to be static for the given time steps. In addition, we assume $P(n) = 0.5$, $P(n|z_k) = 0.7$ given the measurement that the cell is occupied, and $P(n|z_k) = 0.3$ given the measurement that the cell is non-occupied. The mapping procedure is explained in the following:

- The environment is modeled as a set of cells stored using an octree data structure. Since we assume $P(n) = 0.5$, which indicates that no prior map is available, the occupancy probability for each cell is initialized to 0.5.
- When the measurements are captured by the sensor after initialization, the measured data is first transformed to world frame and then utilized to update the probability for each cell, according to (5.2). The updated values for each time step are shown in the second row of Figure 5.2.
- Thereafter, a threshold on the occupancy probability is applied to obtain a deterministic map required for navigation, as shown in the third row of Figure 5.2. In this example, the threshold is set to 0.9. Intuitively, for the cells with occupancy probabilities greater than 0.9, they are considered to be occupied; for those cells with probabilities less than 0.1, they are regarded as non-occupied cells; and the other cells remain unknown.
- The above two steps are repeated when new data is acquired from the sensor at the next time step.

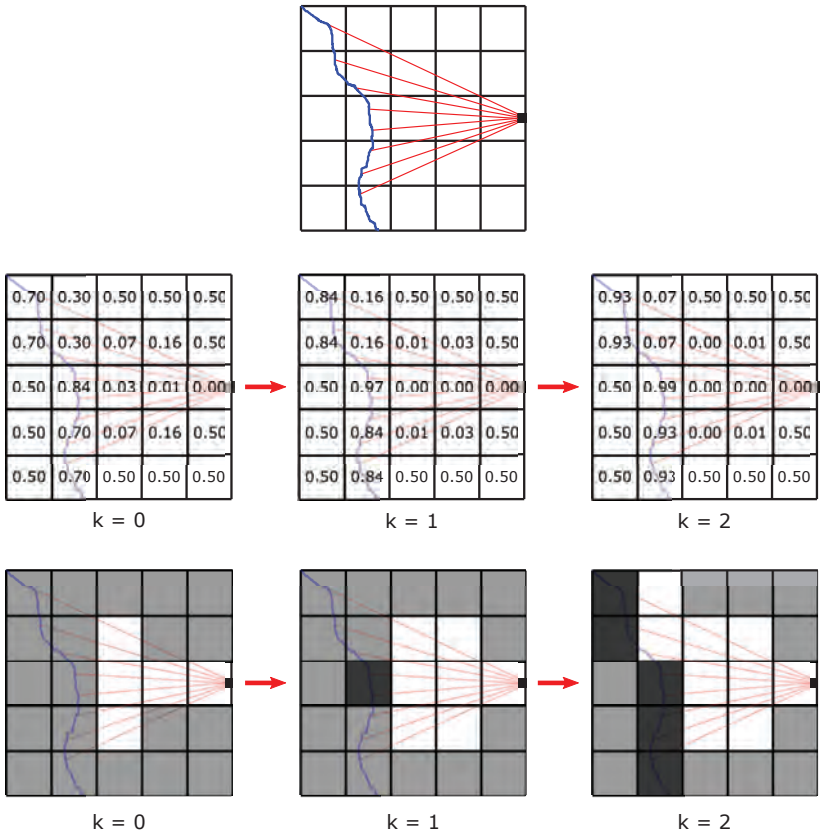


Figure 5.2: The illustration of creating an Octomap. In the first row, the mesh grid shows an example scenario where the surface (blue curve) of an object is measured by a sensor (black square). The captured measurements are presented as the red lines. The three mesh grids in the second row show the probabilities of each cell to be occupied for three subsequent time steps. The mesh grids in the third row give the corresponding occupancy grid maps based on the sensor data for each time step, where the occupied cells (dark grey), the non-occupied cells (white), and the unknown cells (light grey) are presented respectively. It can be imaged that with smaller cell size, the surface can be modeled more accurately.

To study the correlation between the number of the cells and the occupancy probability, a similar 2D scenario in Figure 5.3 is considered, where the size of the whole area is $1m \times 1m$, and the same probability assumptions in the above example (see Figure 5.2) are adopted. In Figure 5.3, similar to Figure 5.2, the blue part stands for an obstacle in the area, the black square represents the sensor, and the red lines are the measurements. In this scenario, we compute the occupancy probability of the whole area with various minimal cell sizes. The results are listed in Table 5.1. From the table, a conclusion can be drawn that the occupancy probability is positively correlated. Intuitively, the smaller the minimal cell size, the smaller the occupancy probability, but the correlation between these two is not linear. The conclusion also indicates that the smaller the minimal cell size, the more accurate the area can be modeled.

Table 5.1: The correlation between the number of the cells and the occupancy probability

Minimal cell size	$1m \times 1m$	$0.5m \times 0.5m$	$0.25m \times 0.25m$	$0.125m \times 0.125m$
Occupancy probability	1.0	0.835	0.47125	0.339375

For implementation, the C++ library for Octomap is utilized. The resolution of the Octomap is set during the initialization, using the following code.

```
_octree = std::make_shared<octomap::ColorOcTree>(_reso);
```

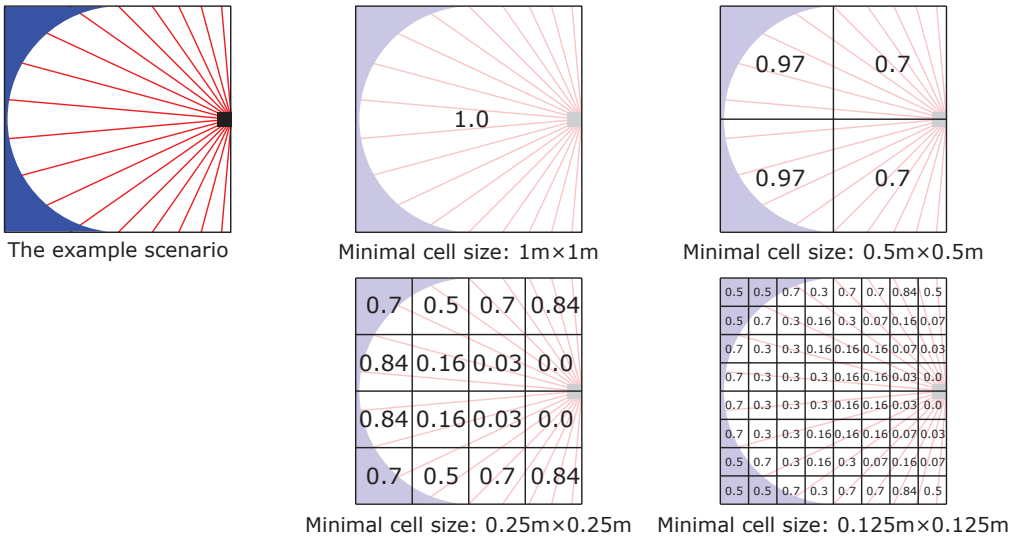


Figure 5.3: The correlation between the number of the cells and the occupancy probability.

The mapping procedure can be describe as follows:

1. Compute the sensor coordinates based on the pixel coordinates and the depth information. This step can also be described by (2.2), but being performed on all the pixels;
2. Reproject the points in sensor coordinates to the world coordinates by incorporating the pose estimate given by the SLAM core module. After this operation, a point cloud in the world coordinates can be obtained;
3. Then process the point cloud to generate the Octomap.

The above procedure is implemented below, while the full implementation of the dense mapping module can be found in Appendix A.4.

```
void DenseRGBD::OctreeMapping(const double& stamp,
    const unsigned int& step, const cv::Mat& imRGB,
    const cv::Mat& imD, const std::vector<Eigen::Isometry3d>& trans)
{
    Eigen::Isometry3d T = trans.back();
    for(int v=0; v<imRGB.rows; ++v)
        for (int u=0; u<imRGB.cols; ++u)
        {
            unsigned int d = imD.ptr<unsigned short>(v)[u];
            if(d <= _lb || d >= _ub)
                continue;
            // align pixel coordinates with depth
            Eigen::Vector3d point;
            point[2] = double(d)/_scale;
            point[0] = (u-_K.at<double>(0,2))*point[2]/
                _K.at<double>(0,0);
            point[1] = (v-_K.at<double>(1,2))*point[2]/
                _K.at<double>(1,1);
            // reprojection
            Eigen::Vector3d pointWorld =T*point;

            PointT p ;
            p.x = pointWorld[0];
```

```
            p.y = pointWorld[1];
            p.z = pointWorld[2];
            p.b = imRGB.data[v*imRGB.step+u*imRGB.channels()];
            p.g = imRGB.data[v*imRGB.step+u*imRGB.channels()+1];
            p.r = imRGB.data[v*imRGB.step+u*imRGB.channels()+2];

            _octree->updateNode(
                octomap::point3d(p.x, p.y, p.z), true);
            _octree->integrateNodeColor(
                p.x, p.y, p.z, p.r, p.g, p.b);
        }
    _octree->updateInnerOccupancy();
}
```

5.3 Experiment Results

For this test, the same dataset used in Chapter 3 and Chapter 4 is considered again to generate an Octomap of the perceived environment. The color image and depth image sequences are fed into the proposed system with all the functions discussed in Chapter 3, Chapter 4, and this chapter, including both loop closure and dense mapping. The parameters are set according to Table 5.2, where the settings are the same as Settings 3 in Table 4.1 with an extra Octomap resolution parameter (Other parameters are described in Appendix B). The output Octomap is visualized using the Rviz software and shown in Figure 5.4. During the test, we found that the dense mapping process is both time consuming and computationally expensive, which is reasonable since it processes all the sensor data.

Table 5.2: Settings for the dense mapping test

Parameter	Settings 4
Camera.fps	10 Hz
ORBExtractor.numFeatures	1000
ORBExtractor.filter	0.6
Depth.bound	[0, 20000] (0-4 m)
Noise.observation	[0.01, 0.01, 0.015]
Noise.constant	0.8
ISAM2.horizon	50
LCDetector.numFeatures	1500
LCDetector.p	250
LCDetector.min_inliers	60
Octomap.resolution	0.05 m

Since we have no ground-truth map for this dataset, the quality of the generated Octomap is difficult to evaluate. But the result shows that the proposed system is able to output a dense Octomap while tracking the sensor.

To better evaluate the memory efficiency and the dense property of the Octomap, another test is carried out after the trajectory is obtained. In this test, several Octomaps are compared with the point cloud maps, using the same trajectory and the dataset in Figure 3.6. The Octomaps with different resolutions and the point cloud maps with different filtering settings (the metric value in the bracket) are shown in Figure 5.5. The effect of the filtering parameter for point cloud map is explained as follows. For a metric value x, only one point is allowed to exist in each x^3 volume. For memory efficiency evaluation, the sizes of the generated Octomaps and the point cloud maps

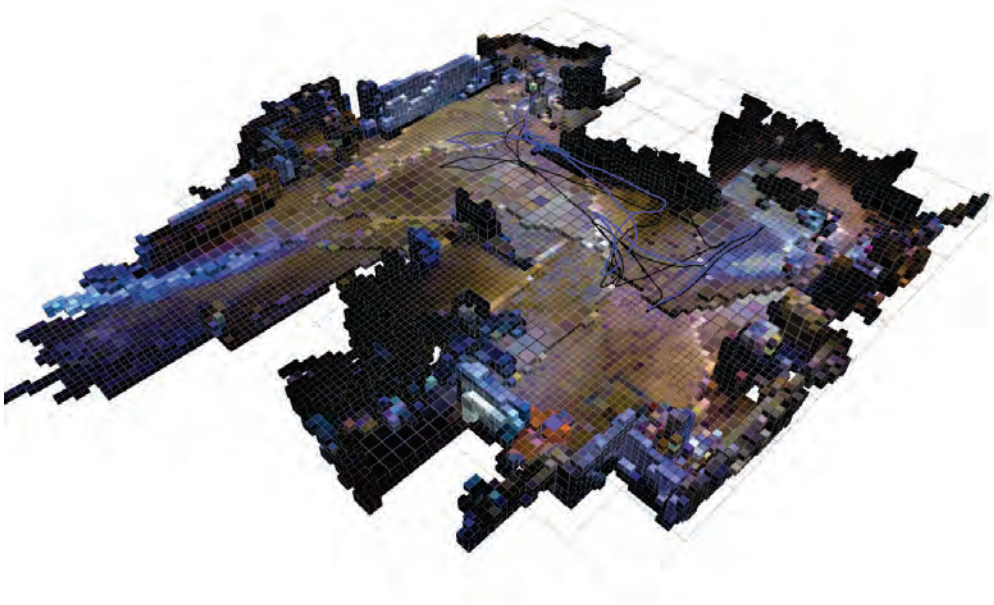


Figure 5.4: The generated Octomap using dataset in Figure 3.6.

are also given in Table 5.3. The it can be observed that by filtering, the size of the constructed point cloud map is greatly reduced, but meanwhile the density is also decreased.

Table 5.3: Map size comparison

Map Representation	Map Size
Point Cloud Map (Raw)	362.74MB
Point Cloud Map (0.005m)	37.08 MB
Point Cloud Map (0.010m)	7.54 MB
Point Cloud Map (0.020m)	1.52 MB
Point Cloud Map (0.050m)	0.19 MB
Octomap (Resolution: 0.005m)	30.33 MB
Octomap (Resolution: 0.010m)	5.22 MB
Octomap (Resolution: 0.020m)	1.01 MB
Octomap (Resolution: 0.050m)	0.16 MB

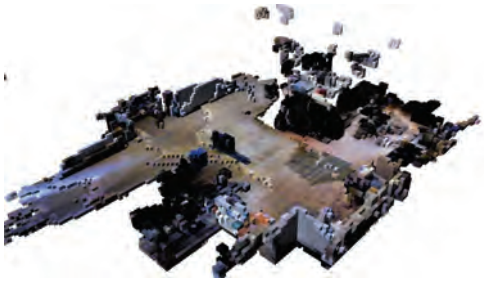
As discussed in Chapter 1, the dense map is required for tasks such as path planning. Furthermore, the map is often stored in the memory of a mobile agent, which is limited. As a result, a preferable map for path planning of a mobile agent should not only be memory efficient, but also provide occupancy information. Based on this concept, the following observations are obtained.

1. For the memory efficiency consideration, it can be seen from Table 5.3 that both the Octomap and the point cloud map can meet the requirement, with proper settings.

2. For the occupancy information consideration, clearly the Octomap is preferable, because point cloud maps are simply sets of points in the space, which do not provide any occupancy information directly. On the other hand, the Octomaps are designed to give such information. This can be seen in Figure 5.5, where the Octomaps are always dense despite of the resolutions.

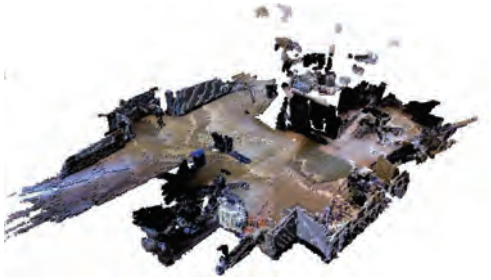
According to the above observations, the conclusion can be drawn that the Octomap representation is more preferred in performing path planning for mobile agents, comparing to the point cloud map.

(a) Point cloud map (0.050m).



(b) Octomap (resolution: 0.050m).

(c) Point cloud map (0.020m).



(d) Octomap (resolution: 0.020m).

(e) Point cloud map (0.010m).

(f) Octomap (resolution: 0.010m).

Figure 5.5: Point cloud maps and Octomaps density comparison

5.4 Summary

This chapter introduces the proposed dense mapping function. We first introduce the Octomap representation used in SLAM. Followed by describing the principle of constructing the Octomap using point clouds at each time step. Finally, the generated Octomap using the same dataset as in Chapter 3 and Chapter 4 with different resolutions are compared with point cloud map representation. The experimental results show that the Octomap representation is memory efficient and suitable for path planning tasks. In the next chapter, all the functions discussed in Chapter 3, Chapter 4, and this chapter are integrated into one system. Thereafter, we evaluate the whole system and analyze the results.

Chapter 6

System Integration and Evaluation

In this chapter, the three modules in the proposed system are integrated. We thus introduce the complete structure of the proposed system. The system integration is followed by evaluation. For this purpose, we first clarify the evaluation method and criterion, and then show the evaluation results comparing to several other SLAM frameworks.

6.1 System Integration

The complete structure of the proposed system is shown in Figure 6.1, where the three modules, the SLAM core, the loop closure, and the dense mapping, run in parallel. All of the three modules take the pairs of images from the RGB-D sensor as inputs.

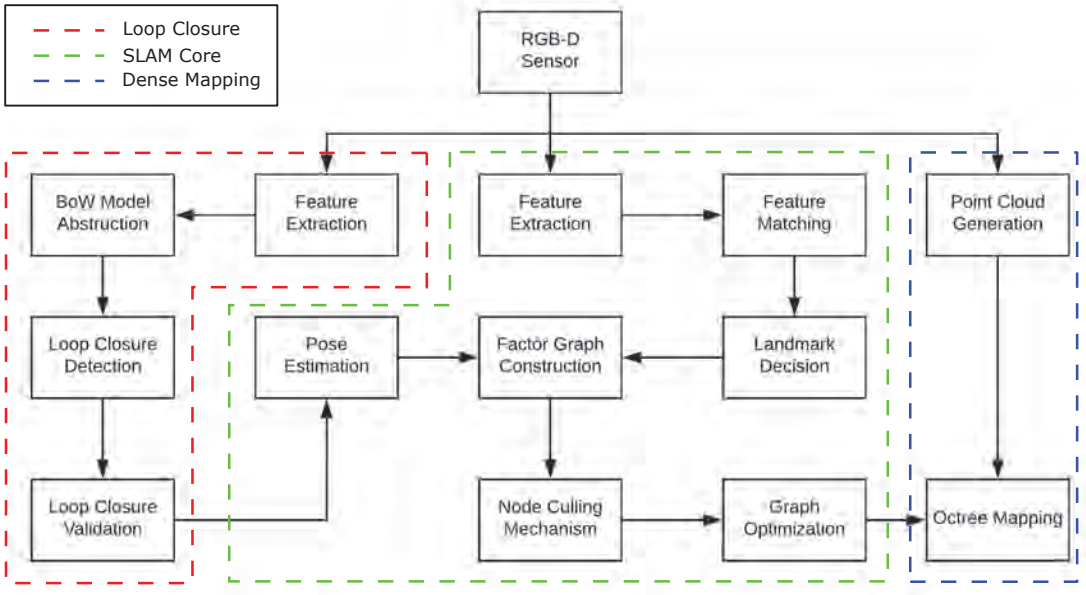


Figure 6.1: The proposed system framework, with three modules: SLAM core, loop closure, and dense mapping.

In this architecture, the SLAM core module is responsible for estimating the trajectory and creating the sparse landmark-based map. As shown in Figure 6.1, the SLAM core module extracts and matches features for landmark decision, as described in Chapter 2. Thereafter, the landmark information is used for constructing the factor graph as described in Chapter 3. Furthermore, loop closure correction, as described in Chapter 4, also happens inside the SLAM core module, since the node culling mechanism functions on the factor graph directly to create the required pose graph. As a result, the loop closure module is mainly for loop closure detection, which is based on iBoW-LCD [27]. Finally, the dense mapping takes the estimated trajectory from the SLAM core module,

and the raw RGB-D data from the sensor to generate the dense Octomap, as described in Chapter 5.

To make the main results of this work easy for usage within other software frameworks, they are programmed using C++ language and a widely used library GTSAM [20] for the factor graph creation and optimization (only in the SLAM core module). Furthermore, the main algorithms are implemented to function completely independent of the ROS environment. Meanwhile, interfaces are also programmed for ROS to call these algorithms. This implementation method guarantees the transportable property of the main results of this work, which is also illustrated in Figure 6.2, where the ROS nodes are programmed following the standard ROS node creation method, and in each node the main algorithms are called.

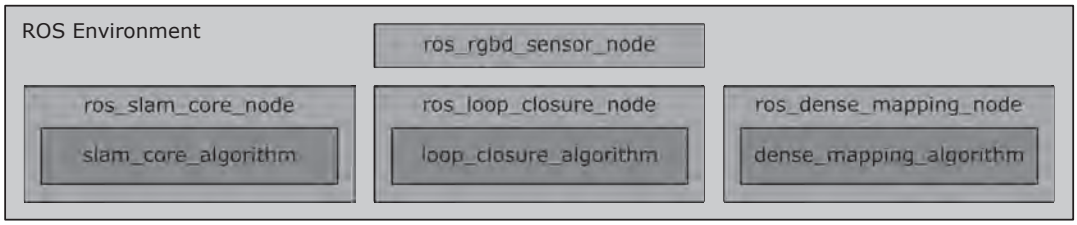


Figure 6.2: Implementation

The main consideration regarding integration is the communications between the three modules and the sensor, because the three modules have different processing times and they require appropriate data transfer to perform correctly. Thus in the next section, the communication issues are discussed, starting with determining the sizes of the communication delays and the processing time. Thereafter, a strategy to handle the communication issues is raised.

6.2 Timing Results and Communication Delays

As can be seen in Figure 6.1, there are five one-way communications happening between modules in the proposed system. They are listed as follows:

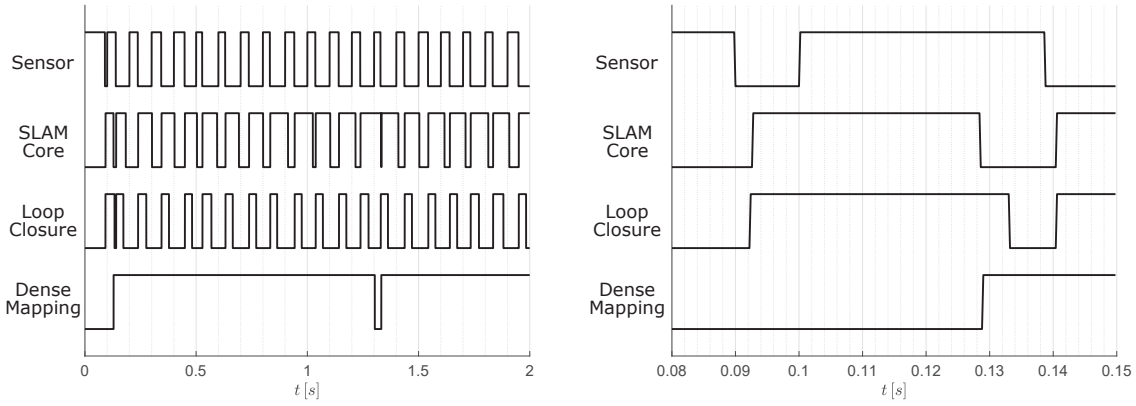
- the loop detection results from the loop closure module to the SLAM core module;
- the estimated trajectory from the SLAM core module to the dense mapping module;
- and the captured images from the RGB-D sensor to the three modules, separately.

To measure the size of the delays, the timing diagrams of the three modules as well as the RGB-D sensor are provided in Figure 6.3. The timing results are obtained using the same dataset in Chapter 3, Chapter 4, and Chapter 5, with the parameters set as Settings 4 in Table 5.2.

Table 6.1: Processing time results

Module	SLAM Core	Loop Closure	Dense Mapping
Average processing time	0.1961 s	1.5438 s	1.2715 s

From Figure 6.3a, it can be seen that the dense mapping process is time consuming comparing to the SLAM core process. This is normal because the dense mapping module performs all the sensor data while the SLAM core module only process a small set of the sensor data (the feature points). In addition, recall that the iBoW-LCD algorithm (the loop closure detection method used in this work) generates and maintains a dictionary in real-time, and the parameter "LCDetector.p" in Table 5.2 defines the number of images required for the dictionary. Therefore, the timing diagram of the loop closure module in Figure 6.3a only presents the time needed for adding the information of each frame into the dictionary. This explains why the processing time for each frame is short. Furthermore, the normal average processing time of each module are listed in Table 6.1. All the



(a) (b)

Figure 6.3: (a) The timing diagrams of the RBG-D sensor and the three parallel modules in time interval $[0, 2]$. (b) The timing diagrams for the time interval $[0.08, 0.15]$.

timing results are obtained under the Ubuntu 16.04 operating system, using the same hardware with an Intel(R) Core(TM) i7-6700HQ CPU and 8GB RAM.

Table 6.2: Delays between modules

Communication	Delay
Sensor to SLAM Core	2 ms
Sensor to Loop Closure	2 ms
Sensor to Dense Mapping	-
Loop Closure to SLAM Core	-
SLAM Core to Dense Mapping	1 ms

Apart from the above observations, the sizes of the delays are measured using the zoomed-in timing diagram in Figure 6.3b. The delays between each modules are list in Table 6.2. The delay between the sensor and the dense mapping module is unknown, because the dense mapping module requires the pose estimate from the SLAM core module, and the pose information reaches the dense mapping module the latest. In addition, the delay between the loop closure module and the SLAM core module is also unknown, because it is hard to verify where the loop closure happens in the diagram.

Based on the previous analysis, the following conclusions can be drawn:

1. The communication delays can be ignored considering the processing time of each module.

2. The processing time of all the three modules are longer than the sensor data capturing time, which indicates that processing every frame is not possible for any of the three modules.

Since the proposed system does not contain an overall frame selection mechanism at this moment, the frame to be processed is selected by each module separately. To be specific, after finishing processing the previous frame, each module gets the newest sensor data (this might be different for each module due to the different processing time) and starts processing it again. In other words, if new sensor data arrives before the module finishes processing the previous data, the newly captured data is dropped by that module. From Table 6.1, it can be inferred that the loop closure module and the dense mapping module drop more sensor data than the SLAM core module. However, this does not mean that the data processed by the loop closure module is also processed by the SLAM core module.

As a result, the above operation leads to the problem illustrated in the following. Recall that the loop closure module is only responsible for detecting loop closure candidates, and the output of this module is a pair of time steps corresponding to the loop closure candidates. Thus it is possible that the sensor data at one of the two time steps is dropped by the SLAM core module. This results in a running error when performing the loop closure correction. This is caused by the principle of the loop closure correction, which adds a new edge into the pose graph discussed in Chapter 4. The new edge is added between the two pose nodes at the two time steps given by the loop closure module. However, if the sensor data at certain time step is dropped by the SLAM core module, the corresponding pose node would not exist in the pose graph. Thus an error occurs when adding the edge into the pose graph.

To solve the problem mentioned above, a simple strategy is adopted in which the SLAM core module checks if the two loop closure candidates exist in the graph (the pose nodes at the two time steps), before performing the loop closure correction. This operation can be done because the sensor data is aligned with a unique time step, and all the time steps of the data received by the SLAM core module are stored. Thus it is easy for the SLAM core module to verify if the data at the two time steps given by the loop closure module has been processed. This method can work, because the SLAM core module processes much more frames than the loop closure module (see the processing time in Table 6.1). Thus most of the loop closures can be processed by the SLAM core module.

To sum up, we present the timing result of the proposed system in this section. In the next section, the estimation accuracy is presented and compared to other SLAM frameworks.

6.3 Evaluation Method and Results

The evaluation of the proposed system mainly depends on the TU Munich datasets [26] and the provided evaluation tools. These datasets and the evaluation tools are also utilized for evaluating other SLAM frameworks such as [8, 13, 28, 31]. As a result, the same evaluation quantities can be obtained for different works.

Figure 6.4 gives some other examples of the datasets apart from the dataset we introduced in Figure 3.6. Note that only the color images are shown, while in the datasets both the color and the depth images are provided. Also note that the name of each dataset consists of two parts, which indicate the sensor index and the scene separately. For example, all the datasets with "fr1" use the same Kinect sensor, but the images are captured from different scenes such as desk and room. Therefore, when performing the evaluation, we try to guarantee that the same parameter settings are used for each sensor in all the scenes, except for the dataset fr1/desk2. For fr1/desk2, the number of inliers is set to 60 instead of 200. The reason is that 200 for the number of inliers results in no loop closure detected when using fr1/desk2. The relevant parameters are listed in Table 6.3 for each sensor. The full description of all the parameters is given in Appendix B.

The estimated trajectories for each dataset are shown in Figure 6.5. In addition, the corresponding translational errors of this work as well as some other SLAM systems using the same datasets are shown in Table 6.4. Note that the data in the first five rows of the table are from other publications, including [8] and [31]. For better analysis, some of the 3D estimated trajectories are also given in Figure 6.6.

6.4 Result Analysis

In general, the conclusion can be drawn from Figure 6.5 that the proposed system is able to estimate the sensor trajectory, and the estimation agrees with the ground-truth. However, considering Table 6.4, the overall estimation accuracy of this work is lower than that of the other

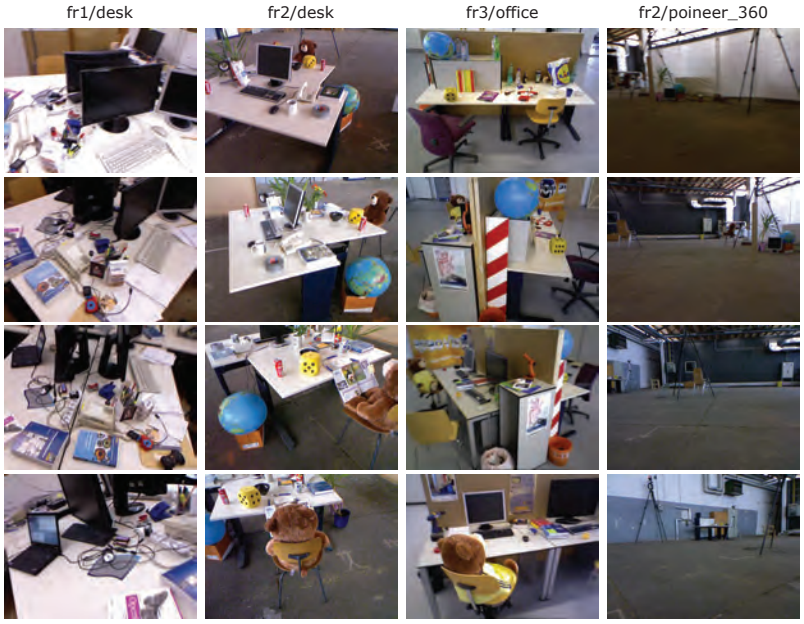


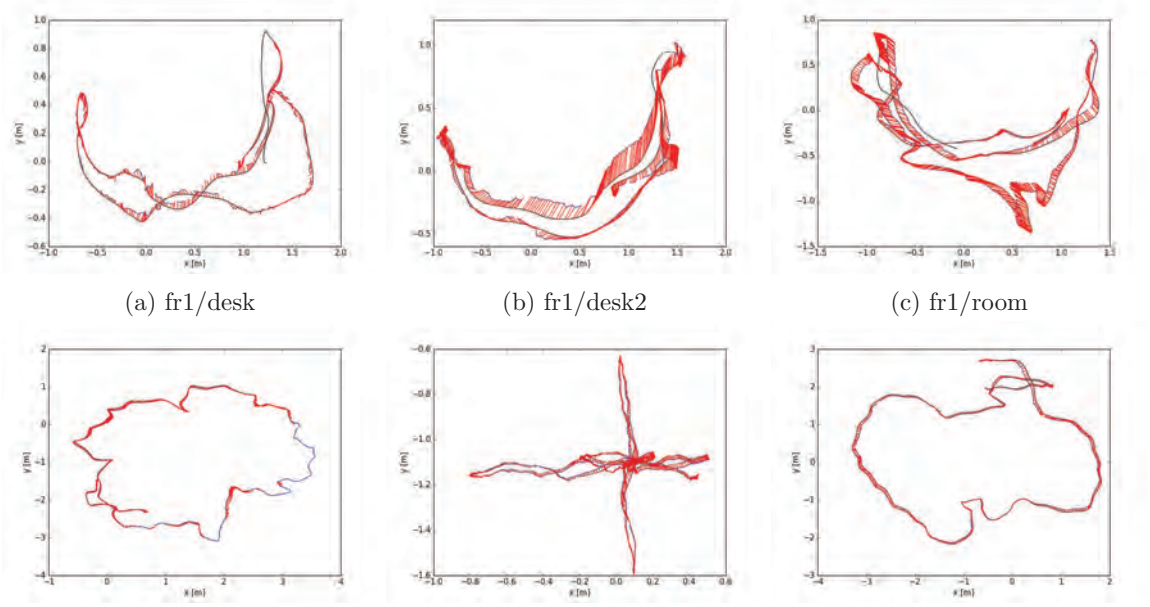
Figure 6.4: Example sequences of color images in TU Munich datasets [26]

Table 6.3: Settings for the evaluation

Parameter	Settings fr1	Settings fr2	Settings fr3
Camera.fps	5 Hz	15 Hz	15 Hz
ORBExtractor.numFeatures	1000	1000	1000
ORBExtractor.filter	0.6	0.6	0.6
Depth.bound	[0, 40000] (0-8 m)	[0, 40000] (0-8 m)	[0, 40000] (0-8 m)
Noise.Observation	[0.01, 0.01, 0.015]	[0.01, 0.01, 0.015]	[0.01, 0.01, 0.015]
Noise.constant	0.4	0.1	0.1
ISAM2.horizon	50	30	30
LCDetector.numFeatures	1500	1500	1500
LCDetector.p	250	250	250
LCDetector.min_inliers	200/60	100	100
Octomap.resolution	0.05 m	0.05 m	0.05 m

SLAM systems. The reasons are analyzed in the following.

From Table 6.4, it can be found that ORB-SLAM2 outperforms the other SLAM systems for most of the datasets listed in the table. Therefore, we compare our system with ORB-SLAM2 to reason about the difference in performance. We found that sudden movements of the sensor exist in most of the datasets, especially fr1/desk2 and fr1/room (see Figure 6.6 for reference, where sudden movements lead to worse rotation estimate). The sudden movements lead to the problem of lacking commonly observed landmarks in subsequent frames, which can result in an inaccurate pose estimate. Therefore, special actions are needed to handle sudden movements, which are absent in our system. On the other hand, ORB-SLAM2 has such mechanisms to cope with this problem, by matching the current frame to a local map. The local map is created based on the sensor data of the previous several frames. Therefore, when choosing landmarks, instead of only matching the current frame to the previous one frame, ORB-SLAM2 tries to find the matches between the current frame and the local map. This mechanism increases the number of matched features and more constraints are applied to pose estimate. In the case that the local map matching method fails, ORB-SLAM2 stops tracking the pose of the sensor and tries to re-localize the sensor according to the already constructed map. The working principle of re-localization is



(a) fr1/desk (b) fr1/desk2 (c) fr1/room

(d) fr2/desk (e) fr2/xyz (f) fr3/office

Figure 6.5: Evaluated trajectories

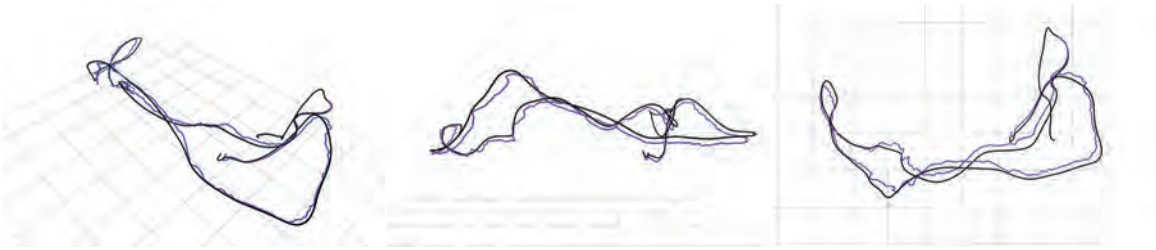
Table 6.4: Evaluation result comparing to some other SLAM systems.

	fr1/desk	fr1/desk2	fr1/room	fr2/desk	fr2/xyz	fr3/office
ORB-SLAM2	**0.016m**	**0.022m**	0.047m	**0.009m**	**0.004m**	**0.010m**
ElasticFusion	0.020m	0.048m	0.068m	0.071m	0.011m	0.017m
DVO-SLAM	0.021m	0.046m	**0.043m**	0.017m	0.018m	0.035m
RGBD-SLAM	0.026m	—	0.087m	0.057m	0.008m	0.032m
Kintinuous	0.037m	0.071m	0.075m	0.034m	0.029m	0.030m
This Work	0.034m	0.088m	0.089m	0.057m	0.018m	0.050m

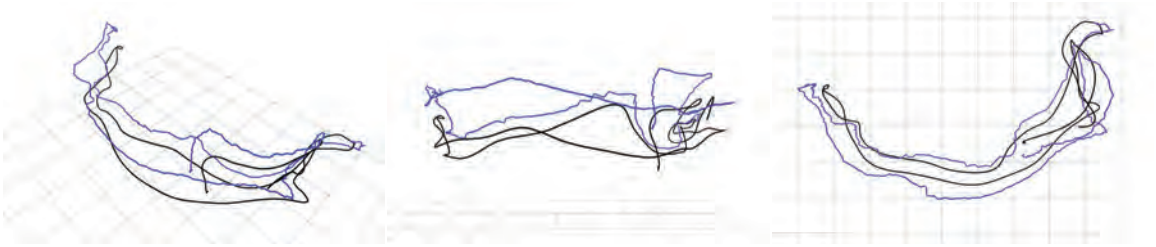
similar to that of loop closure. As a result, in general ORB-SLAM2 outperforms the other systems.

Another important issue is that all the systems listed in Table 6.4 are complete SLAM systems, which contain both front-ends and back-ends. For the front-ends in those systems, a visual odometry (VO) always exists which estimates the location of the sensor based on the frame-to-frame pose transformation, but without loop closure or mapping processes. An accurate VO can provide good initial guesses of poses for the nonlinear optimization problem, as claimed in Chapter 3. Furthermore, the VO also provides extra connections (edges in graph) between the states to be determined. In one word, the VO is an important subject in the front-end of a SLAM system, which requires equivalent research effort to the back-end. On the other hand, since this work mainly focuses on the back-end and dense mapping, it only contains a naive front-end without any VO information. Thus inherently, this work is at a disadvantage when comparing to the other SLAM systems.

Due to the lacking of a proper front-end, the performance of the proposed system is hard to evaluate only from Table 6.4, because it is hardly possible to peel the front-ends off from the other systems and only compare the performance of the back-ends. Thus, no conclusion about the estimation accuracy can be drawn at this moment.



(a) fr1/desk oblique view (b) fr1/desk front view (c) fr1/desk top view



(d) fr1/desk2 oblique view (e) fr1/desk2 front view (f) fr1/desk2 top view

(g) fr1/room oblique view (h) fr1/room front view (i) fr1/room top view

Figure 6.6: The 3D trajectories for fr1/desk, fr1/desk2, and fr1/room.

6.5 Summary

In this chapter, we integrate the three modules introduced in the previous chapters into one system. The method to handle the communication delays are also discussed. Later, we evaluate the proposed system and compare with other SLAM systems, using the same online real-world RGB-D datasets. Based on the evaluation results, we reason about the difference in performance. In the next chapter, this work is concluded and recommendations for future works are also listed.

Chapter 7

Conclusion and Recommendations

In the previous chapter, the architecture of the proposed system was introduced. We also evaluated the proposed system using online RGB-D datasets, and compare our work with other SLAM frameworks. The experiments showed reasonable results of our work. In this chapter, we sum up this work and give several recommendations for future work.

7.1 Objectives Review

This work mainly focuses on developing a back-end of a typical SLAM system, which is capable of handling loop closure and dense mapping. The main difference of this work from the other SLAM frameworks is the usage of the iSAM2 algorithm, instead of batch optimization algorithms. The iSAM2 algorithm is applied on a factor graph that is created and maintained by the SLAM core module. In addition, a node culling mechanism is performed directly on the factor graph to enable real-time loop closure correction.

The proposed system contains three parallel modules, namely the SLAM core module, the loop closure module, and the dense mapping module, running on a CPU. These modules are respectively used for tasks of localizing the robot and creating the sparse map, detecting the loop closure candidates, and creating a dense map of the working area. The three modules process the raw data from the RGB-D sensor and communicate with each other to obtain the required outputs.

Regarding the implementation, the proposed system depends on C++ programming language which is compatible for most of the software platforms, and a widely used library for the graph optimization process. Furthermore, the main algorithms are programmed completely independent of the ROS environment, which makes the main result of this work easy to transport to other platforms. Meanwhile, interfaces are also implemented for running the system in the ROS environment. Therefore, the proposed system fully meets the implementation requirement.

Finally, we tested the system in the ROS environment and compared the proposed system with other SLAM frameworks using various online real-world datasets. The experimental results have been analyzed to evaluate this work. The results show that the proposed system can give a roughly correct trajectory estimate as well as an Octomap of the working area. However, no conclusion can be drawn about the estimation accuracy of the proposed system, since no fair comparison can be made between this work and other systems.

To summarize, the requirements have been met. However, the proposed system still has large room for improvements, in terms of the estimation accuracy, the robustness of the system, and the parameter tuning method. They are suggested as future works and further discussed in the next section.

7.2 Recommendations

Based on our knowledge of the current state-of-the-art SLAM frameworks and the evaluation results of this work, the three main improvements are listed below.

1. Incorporate a proper front-end in the system. The front-end should include a visual odometry (VO) and a keyframe selection. The VO can provide more accurate initial guesses for the nonlinear optimization problem. It also provides extra connections between the states to be determined, which can be used as new edges in the factor graph. Apart from the VO, a keyframe selection is also recommended. Since the back-end is impossible to process every frame, the keyframe selection should pick up the distinct frames from all the sensor frames such that the back-end can make full use of the limited computational resource to give the best estimation.

2. Exploit the sensor data for landmark selection. The exploitation can be carried out in two directions. The first one is taking the advantage of the special structure of the data, e.g. lines, planes. These features can give stronger and more robust constraints on the states to be determined. However, this requires more computational resource such as a GPU. The other direction is making the full use of the feature points, with or without depth information. This can be carried out by adding new factors to the graph based on different observation models.

3. Develop mechanisms for special situations, especially sudden movements. The sudden movement often results in inaccurate or even wrong pose estimates. To cope with this problem, special mechanisms, such as local map matching and re-localization in ORB-SLAM2, can also be included in the proposed system.

4. Tune the parameters in a better way. Specifically, the covariance matrix of the measurement plays an important role in estimating the states. From [32] and [33], we know that the distance measurement of a RGB-D senor usually has significant influences on the depth noise variance. [33] showed that the depth error can be fitted to either a polynomial model or a exponential model, depending on the choice of the sensor. Thus in the future, the covariance matrix can be estimated according to the error model.

Bibliography

[1] Mi robot vacuum.
Available at https://www.mi.com/roomrobot/gallery/.

[2] Drones in agriculture, then and now.
Available at https://blog.dronedeploy.com/.

[3] Mars rover wallpapers.
Available at https://wallpaperplay.com/board/mars-rover-wallpapers.

[4] Opinion: On autonomous cars, motorcyclists, and culpability.
Available at https://www.rideapart.com/articles/264772/.

[5] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J.J. Leonard. Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[6] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics*, 31(5):1147–1163, Oct 2015.

[7] C. Park, S. Kim, P. Moghadam, C. Fookes, and S. Sridharan. Probabilistic Surfel Fusion for Dense LiDAR Mapping. *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, pages 2418–2426, 2017.

[8] R. Mur-Artal and J. D. Tardos. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *CoRR*, abs/1610.06475, 2016.

[9] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*, pages 2564–2571, Nov 2011.

[10] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. volume 3951, 07 2006.

[11] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 778–792, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[13] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard. 3-D Mapping With an RGB-D Camera. *IEEE Transactions on Robotics*, 30:177–187, 2014.

[14] Wikipedia contributors. Iterative closest point — Wikipedia, the free encyclopedia, 2019. [Online; accessed 20-June-2019].

[15] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Incremental Smoothing and Mapping. *IEEE Trans. on Robotics (TRO)*, 24(6):1365–1378, December 2008.

[16] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J.J. Leonard, and F. Dellaert. iSAM2: Incremental Smoothing and Mapping Using the Bayes Tree. *Intl. J. of Robotics Research (IJRR)*, 31:217–236, February 2012.

[17] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[18] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, Nov 2000.

[19] D. Lowe. Distinctive image features from scale-invariant key points. *International Journal of Computer Vision*, 20:91–110, 01 2003.

[20] F.Dellaert. Factor graphs and gtsam: A hands-on introduction. Technical report, Georgia Institute of Technology, 2012.

[21] T. D. Barfoot. *State Estimation for Robotics*. Cambridge University Press, 2017.

[22] Wikipedia contributors. Outlier — Wikipedia, the free encyclopedia, 2019. [Online; accessed 20-June-2019].

[23] Wikipedia contributors. M-estimator — Wikipedia, the free encyclopedia, 2019. [Online; accessed 3-July-2019].

[24] Z. Zhang. Parameter estimation techniques: a tutorial with application to conic fitting. *Image and Vision Computing*, 15(1):59 – 76, 1997.

[25] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[26] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A Benchmark for the Evaluation of RGB-D SLAM Systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

[27] E. Garcia-Fidalgo and A. Ortiz. iBoW-LCD: An Appearance-based Loop Closure Detection Approach using Incremental Bags of Binary Words. *CoRR*, abs/1802.05909, 2018.

[28] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. Mcdonald. Kintinuous: Spatially Extended KinectFusion. 07 2012.

[29] N. Kejriwal, S. Kumar, and T. Shibata. High performance loop closure detection using bag of word pairs. *Robotics and Autonomous Systems*, 77:55 – 65, 2016.

[30] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. Octomap: an efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, Apr 2013.

[31] T. Whelan, R. F. Salas-Moreno, B. Glocker, A. J. Davison, and S. Leutenegger. ElasticFusion: Real-time dense SLAM and light source estimation. *The International Journal of Robotics Research*, 35, 09 2016.

[32] C. V. Nguyen, S. Izadi, and D. Lovell. Modeling Kinect Sensor Noise for Improved 3D Reconstruction and Tracking. In *2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization Transmission*, pages 524–530, Oct 2012.

[33] E. Cabrera, L. Ortiz, B. Silva, E. Clua, and L. Gonçalves. A Versatile Method for Depth Data Error Estimation in RGB-D Sensors. *Sensors*, 18:3122, 09 2018.

Appendix A

System C++ Implementation

A.1 Landmark Decision

The following codes incorporate the whole procedure of selecting landmarks from raw sensor data, as discussed in Chapter 2.

The program starts with an initialization of the system, where it reads parameters from a settings file, and create the feature extractor used for finding landmarks in the raw color image. Furthermore, since the landmark selection process happens inside the SLAM core module, the initialization also includes setting up the iSAM optimizer and the noise model (the covariance matrices, which in this work are assumed to be diagonal). In addition, the initialization also sets the initial pose, which is normally obtained from the ground-truth pose at the starting time step.

```
// constructor
Core::Core(std::string &fileSettings)
    :_step(0), _idx(1), _isLoop(false)
{
    // read parameters from .yaml file
    ReadParameters(fileSettings);

    // initialize odometry
    _poses.push_back(_initPose);

    // initialize isam2
    InitOptimizer();

    // set noises
    SetNoiseModel();

    // create orb_slam2 feature extractor
    _extractor = std::make_shared<ORB_SLAM2::ORBextractor>
        (_numFeatures,1.2,8,20,7);
}
```

The functions used for initialization are given as follows.

```
// read parameters
void Core::ReadParameters(const std::string& fileSettings)
{
    YAML::Node settings = YAML::LoadFile(fileSettings);

    // work path
    _path = settings["IO.path"].as<std::string>();
```

```
// initial pose
std::vector<double> initPose =
    settings["Pose.init"].as<std::vector<double>>();
_initPose = gtsam::Pose3(gtsam::Rot3(gtsam::Quaternion(
    initPose[6],initPose[3],initPose[4],initPose[5])),
    gtsam::Point3(initPose[0],initPose[1],initPose[2]));

// camera intrinsic params
double fx = settings["Camera.fx"].as<double>();
double fy = settings["Camera.fy"].as<double>();
double cx = settings["Camera.cx"].as<double>();
double cy = settings["Camera.cy"].as<double>();
_K = (cv::Mat_<double>(3,3,CV_64F)
    << fx, 0, cx, 0, fy, cy, 0, 0, 1);

// camera distortion params
double k1 = settings["Camera.k1"].as<double>();
double k2 = settings["Camera.k2"].as<double>();
double p1 = settings["Camera.p1"].as<double>();
double p2 = settings["Camera.p2"].as<double>();
double k3 = settings["Camera.k3"].as<double>();
if(k3 == 0)
    _D = (cv::Mat_<double>(4,1,CV_64F) << k1, k2, p1, p2);
else
    _D = (cv::Mat_<double>(5,1,CV_64F) << k1, k2, p1, p2, k3);

// depth scale
_scale = settings["Depth.scale"].as<double>();
// depth boundary
std::vector<double> bound =
    settings["Depth.bound"].as<std::vector<double>>();
_depthLb = bound[0];
_depthUb = bound[1];

// extractor and matcher params
_numFeatures =
    settings["ORBExtractor.numFeatures"].as<unsigned int>();
_ratio = settings["ORBExtractor.filter"].as<double>();

// noise params
std::vector<double> sigmasPrior =
    settings["Noises.prior"].as<std::vector<double>>();
_sigmasPrior = gtsam::Vector6(sigmasPrior.data());
std::vector<double> sigmasTrans =
    settings["Noises.process"].as<std::vector<double>>();
_sigmasTrans = gtsam::Vector6(sigmasTrans.data());
std::vector<double> sigmasLoop =
    settings["Noises.loop"].as<std::vector<double>>();
_sigmasLoop = gtsam::Vector6(sigmasLoop.data());
std::vector<double> sigmasPose =
    settings["Noises.pose"].as<std::vector<double>>();
_sigmasPose = gtsam::Vector6(sigmasPose.data());
std::vector<double> sigmasObs =
```

```
        settings["Noises.observation"].as<std::vector<double>>();
    _sigmasObs = gtsam::Vector3(sigmasObs.data());
    _delta = settings["Noises.constant"].as<double>();

    // isam params
    _horizon = settings["ISAM2.horizon"].as<unsigned int>();
    std::vector<double> thresholdX =
        settings["ISAM2.thres_x"].as<std::vector<double>>();
    _thresholdX = gtsam::Vector6(thresholdX.data());
    std::vector<double> thresholdL =
        settings["ISAM2.thres_l"].as<std::vector<double>>();
    _thresholdL = gtsam::Vector3(thresholdL.data());
    _numFactors = settings["SLAM.thres_l"].as<unsigned int>();
}
```

```
// initialize isam2
void Core::InitOptimizer()
{
    // set ISAM2 parameters
    gtsam::FastMap<char, gtsam::Vector> thresholds;
    thresholds['x'] = _thresholdX;
    thresholds['l'] = _thresholdL;

    _params.setRelinearizeSkip(1);
    _params.setRelinearizeThreshold(thresholds);

    // create ISAM2
    _isam = std::make_shared<gtsam::ISAM2>(_params);
}
```

```
void Core::SetNoiseModel()
{
    _noisePrior = gtsam::noiseModel::Diagonal::Sigmas(_sigmasPrior);
    _noiseObs = gtsam::noiseModel::Diagonal::Sigmas(_sigmasObs);
    _noiseTrans = gtsam::noiseModel::Diagonal::Sigmas(_sigmasTrans);
    _noiseLoop = gtsam::noiseModel::Diagonal::Sigmas(_sigmasLoop);
    _noisePose = gtsam::noiseModel::Diagonal::Sigmas(_sigmasPose);

    _mEstimator =
        gtsam::noiseModel::mEstimator::Huber::Create(_delta);

    _robustNoiseObs =
        gtsam::noiseModel::Robust::Create(_mEstimator, _noiseObs);
    _robustNoiseTrans =
        gtsam::noiseModel::Robust::Create(_mEstimator, _noiseTrans);
    _robustNoisePose =
        gtsam::noiseModel::Robust::Create(_mEstimator, _noisePose);
    _robustNoiseLoop =
        gtsam::noiseModel::Robust::Create(_mEstimator, _noiseLoop);
}
```

After initialization, the landmark selection can be performed, by the following code. This process starts with extracting landmark candidates from the raw color image, and then indexing the landmarks based on the feature matching results.

```
void Core::FrontEnd(const double& stamp, const unsigned int& step,
const cv::Mat& imRGB, const cv::Mat& imD)
{
    // store depth image, stamp, and time step for later uses
    _imDs.push_back(imD);
    _stamps.push_back(stamp);
    _steps.push_back(step);

    // extract features from the color image
    ExtractFeatures(imRGB);

    if(_step == 0)
    {
        BackEnd();
        return;
    }

    // feature matching
    std::vector<cv::DMatch> matches;
    MatchFeatures(_des[_step-1], _des[_step], matches);

    // 3d feature point creation and landmarks decision
    std::vector<unsigned int> trainIndices = _trainIndices;
    std::vector<Landmark> observations = _obs2;

    _trainIndices.clear();
    _obs1.clear();
    _obs2.clear();

    std::vector<cv::Point3d> pts1, pts2;

    for(cv::DMatch m:matches)
    {
        cv::Point3d p1, p2;
        bool res = GetMeas(_step-1, _step, m, p1, p2);

        if(!res)
            continue;

        _trainIndices.push_back(m.trainIdx);

        std::vector<unsigned int>::iterator it = std::find(
            trainIndices.begin(), trainIndices.end(), m.queryIdx);

        if(it != trainIndices.end())
        {
            unsigned int idx =
                std::distance(trainIndices.begin(),it);
            _obs1.push_back({false, observations[idx].index, p1});
            _obs2.push_back({false, observations[idx].index, p2});
        }
        else
        {
```

```
            _obs1.push_back({false, _idx, p1});
            _obs2.push_back({false, _idx, p2});
            ++_idx;
        }
    }

    BackEnd();
}
```

After this operation the landmarks are selected and stored. This information is fed to the back-end to create a factor graph and optimize the poses and landmarks. The back-end is the main function in the SLAM core module, which is introduced in the next section.

A.2 SLAM Core Module

Since the initialization of the SLAM core module has been introduced in the previous section, we directly give the main function for the SLAM core module in the following.

This function creates the required factor graph based on the data given by the landmark selection process. It adds pose nodes and landmark nodes to the factor graph and optimize them at each time step. In addition, the loop closure correction and the node culling mechanism are also incorporated in the SLAM core module.

```
// back-end
void Core::BackEnd()
{
    // add pose node
    addPose();

    // add landmark nodes
    addLandmarks();

    // loop correction
    if(_isLoop)
    {
        addLoopFactor();
    }

    // node culling
    unsigned int skipSize = std::max<unsigned int>(2, _horizon);
    _factorsToRemove.clear();
    if(_step >= skipSize)
        removeLandmarks(skipSize);

    // optimize estimate
    _isam->update(_graph, _initEst, _factorsToRemove);
    _isam->update();
    _Est = _isam->calculateEstimate();

    _graph.resize(0);
    _initEst.clear();

    ++_step;
}
```

The functions used for the back-end are given in the following.

```
// add pose node
void Core::addPose()
{
    if(_step == 0)
    {
        _graph.addExpressionFactor(gtsam::Pose3_('x',_step),
            _poses[_step], _noisePrior);
        _initEst.insert(gtsam::Symbol('x', _step), _poses[_step]);
    }
    else
    {
        gtsam::Pose3 odometry, initPose;
        odometry = gtsam::Pose3();
        initPose =
            _Est.at<gtsam::Pose3>(gtsam::Symbol('x',_step-1));

        _graph.addExpressionFactor(between(
            gtsam::Pose3_('x',_step-1),
            gtsam::Pose3_('x',_step)),
            odometry, _robustNoiseTrans);
        _initEst.insert(gtsam::Symbol('x', _step), initPose);
    }
}
```

```
// add landmark nodes
void Core::addLandmarks()
{
    for(size_t i=0; i<_obs1.size(); ++i)
    {
        if(!_Est.exists(gtsam::Symbol('l', _obs1[i].index)))
        {
            gtsam::Point3 factor_1(_obs1[i].position.x,
                                   _obs1[i].position.y,
                                   _obs1[i].position.z);
            // add factor between landmark and pose
            _graph.addExpressionFactor(RGBDFactor_(
                gtsam::Pose3_('x',_step-1),
                gtsam::Point3_('l',_obs1[i].index)),
                factor_1, _robustNoiseObs);
            gtsam::Point3 initPoint = _Est.at<gtsam::Pose3>(
                gtsam::Symbol('x',_step-1)).transform_from(factor_1);
            _initEst.insert(gtsam::Symbol('l', _obs1[i].index),
                init_point);
        }

        gtsam::Point3 factor_2(_obs2[i].position.x,
                               _obs2[i].position.y,
                               _obs2[i].position.z);
        // add factor between landmark and pose
        _graph.addExpressionFactor(RGBDFactor_(
            gtsam::Pose3_('x',_step),
            gtsam::Point3_('l',_obs2[i].index)),
            factor_2, _robustNoiseObs);
```

```
    }
}
```

```
// add loop closure factor
void Core::addLoopFactor()
{
    _graph.addExpressionFactor(between(
        gtsam::Pose3_('x',_loopIdx1),
        gtsam::Pose3_('x',_loopIdx2)),
        _loopFactor, _robustNoiseLoop);
    _isLoop = false;
}
```

```
// remove landmark nodes
void Core::removeLandmarks(const unsigned int& skipSize)
{
    gtsam::VariableIndex variable_index = _isam->getVariableIndex();
    gtsam::VariableIndex::Factors factors_1 =
        variable_index[gtsam::Symbol('x',_step-skipSize)];
    gtsam::VariableIndex::Factors common_factors;

    // store landmarks
    std::vector<gtsam::Point3> landmarks;
    for(size_t i=1; i<_idx; ++i )
    {
        if(!_Est.exists(gtsam::Symbol('l',i)))
            continue;

        gtsam::VariableIndex::Factors factors_l =
            variable_index[gtsam::Symbol('l', i)];

        if(factors_l.size() == 1)
        {
            landmarks.push_back(
                _Est.at<gtsam::Point3>(gtsam::Symbol('l',i)));
        }
    }

    _landmarks.push_back(landmarks);

    // keep all the pose factors including loop factors
    for(size_t i=0; i<_step; ++i)
    {
        if(_step == skipSize)
            continue;
        else if(i <= _step-skipSize+1 && i >= _step-skipSize-1)
            continue;

        gtsam::VariableIndex::Factors factors_2 =
            variable_index[gtsam::Symbol('x',i)];
        // find all the poses connected to the first pose
        // within the horizon.
        std::set_intersection(factors_1.begin(), factors_1.end(),
            factors_2.begin(), factors_2.end(),
```

```
            std::back_inserter(common_factors));
    }

    std::sort(common_factors.begin(), common_factors.end());
    std::set_difference(factors_1.begin(), factors_1.end(),
        common_factors.begin(), common_factors.end(),
        std::back_inserter(_factorsToRemove));

    // start removing current factors and adding new factors
    if(_step == skipSize)
    {
        gtsam::Pose3 prior_factor(_Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-skipSize)));
        _graph.addExpressionFactor(
            gtsam::Pose3_('x',_step-skipSize),
            prior_factor, _noisePrior);
    }
    else
    {
        gtsam::Pose3 process_factor_1 =
            _Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-skipSize-1)).between(
            _Est.at<gtsam::Pose3>(
            gtsam::Symbol('x',_step-skipSize)));
        _graph.addExpressionFactor(between(
            gtsam::Pose3_('x',_step-skipSize-1),
            gtsam::Pose3_('x',_step-skipSize)),
            process_factor_1, _robustNoisePose);
    }

    gtsam::Pose3 process_factor_2 =
        _Est.at<gtsam::Pose3>(
        gtsam::Symbol('x',_step-skipSize)).between(
        _Est.at<gtsam::Pose3>(
        gtsam::Symbol('x',_step-skipSize+1)));

    _graph.addExpressionFactor(between(
        gtsam::Pose3_('x',_step-skipSize),
        gtsam::Pose3_('x',_step-skipSize+1)),
        process_factor_2, _robustNoisePose);
}
```

A.3 Loop Closure Module

Recall that the loop closure module is mainly for loop closure detection based on iBoW-LCD method [27], as described in Chapter 4 and Chapter 6. As usual, the loop closure module starts with an initialization given in the following.

```
LoopDetector::LoopDetector(std::string &fileSettings)
    :_step(0), _isLoop(false)
{
    ReadParameters(fileSettings);

    // initialize loop detector
```

```
    InitDetector();
}
```

The functions used for initialization are given below.

```
// read parameters
void LoopDetector::ReadParameters(const std::string &fileSettings)
{
    YAML::Node settings = YAML::LoadFile(fileSettings);

    // common params
    _path = settings["IO.path"].as<std::string>();
    _method = settings["LCDetector.method"].as<unsigned int>();
    _numFeatures =
        settings["LCDetector.numFeatures"].as<unsigned int>();

    // ibow-lcd params
    _p = settings["LCDetector.p"].as<unsigned>();
    _nndr = settings["LCDetector.nndr"].as<float>();
    _nndr_bf = settings["LCDetector.nndr_bf"].as<float>();
    _ep_dist = settings["LCDetector.ep_dist"].as<double>();
    _conf_prob = settings["LCDetector.conf_prob"].as<double>();
    _min_score = settings["LCDetector.min_score"].as<double>();
    _island_size =
        settings["LCDetector.island_size"].as<unsigned>();
    _min_inliers =
        settings["LCDetector.min_inliers"].as<unsigned>();
    _nframes_after_lc =
        settings["LCDetector.nframes_after_lc"].as<unsigned>();
    _min_consecutive_loops =
        settings["LCDetector.min_consecutive_loops"].as<int>();
}
```

```
// initialize detector
void LoopDetector::InitDetector()
{
    switch(_method)
    {
        // classical bow method
        case 1:
            _vocab = std::make_shared<DBoW3::Vocabulary>(_db_path);
            _db = std::make_shared<DBoW3::Database>
                (*_vocab, false, 0);
            break;

        // ibow-lcd method
        default:
            _params.p = _p;
            _params.nndr = _nndr;
            _params.nndr_bf = _nndr_bf;
            _params.ep_dist = _ep_dist;
            _params.conf_prob = _conf_prob;
            _params.min_score = _min_score;
            _params.island_size = _island_size;
            _params.min_inliers = _min_inliers;
```

```
            _params.nframes_after_lc = _nframes_after_lc;
            _params.min_consecutive_loops = _min_consecutive_loops;

            _lcdetector =
                std::make_shared<ibow_lcd::LCDetector>(_params);
            break;
    }
}
```

After the initialization, the loop closure module starts functioning, according to the following program. It first extracts the keypoints and the descriptors from the color image, and uses this information for loop closure detection.

```
void LoopDetector::Process(const unsigned int& step,
    const cv::Mat& imRGB)
{
    _steps.push_back(step);

    std::vector<cv::KeyPoint> kp;
    cv::Mat des;
    // Creating feature detector and descriptor
    cv::Ptr<cv::Feature2D> detector = cv::ORB::create(_numFeatures);
    detector->detectAndCompute(imRGB, cv::Mat(), kp, des);

    // start detecting loop candidates
    DetectLoop(kp, des);

    ++_step;
}
```

The main function for detecting a loop closure candidate is given below.

```
// detect loop
void LoopDetector::DetectLoop(const std::vector<cv::KeyPoint>& kp,
    const cv::Mat& des)
{
    switch(_method)
    {
        // bow method, not used currently.
        case 1:
            bowMethod(kp, des);
            break;
        // ibow method
        default:
            ibowMethod(kp, des);
            break;
    }
}
```

```
// ibow method
void LoopDetector::ibowMethod(const std::vector<cv::KeyPoint>& kp,
    const cv::Mat& des)
{
    ibow_lcd::LCDetectorResult result;
    _lcdetector->process(_step, kp, des, &result);
```

```
    if(result.status == ibow_lcd::LC_DETECTED && result.inliers > 0)
    {
        _isLoop = true;
        _candidate = _steps[result.train_id];
    }
    else
    {
        _isLoop = false;
    }
}
```

A.4 Dense Mapping Module

The dense mapping module performs using the raw sensor data and the estimated sensor trajectory. It starts by setting the resolution of the generated Octomap. The initialization procedure is given in the following program.

```
DenseRGBD::DenseRGBD(std::string& fileSettings)
{
    ReadParameters(fileSettings);
    _octree = std::make_shared<octomap::ColorOcTree>(_reso);
}
```

The function used for initialization is given in the following.

```
void DenseRGBD::ReadParameters(const std::string& fileSettings)
{
    YAML::Node settings = YAML::LoadFile(fileSettings);

    // work path
    _path = settings["IO.path"].as<std::string>();
    // intrinsic parameters
    double fx = settings["Camera.fx"].as<double>();
    double fy = settings["Camera.fy"].as<double>();
    double cx = settings["Camera.cx"].as<double>();
    double cy = settings["Camera.cy"].as<double>();
    _K = (cv::Mat_<double>(3,3,CV_64F)
        << fx, 0, cx, 0, fy, cy, 0, 0, 1);
    // depth scale
    _scale = settings["Depth.scale"].as<double>();
    // depth boundary
    std::vector<double> bound =
        settings["Depth.bound"].as<std::vector<double>>();
    _lb = bound[0];
    _ub = bound[1];
    // resolution
    _reso = settings["Octomap.resolution"].as<double>();
}
```

This module functions based on the following program, which is main function of dense mapping module.

```
void DenseRGBD::OctreeMapping(const double& stamp,
    const unsigned int& step, const cv::Mat& imRGB,
    const cv::Mat& imD, const std::vector<Eigen::Isometry3d>& trans)
{
```

```
    Eigen::Isometry3d T = trans.back();
    for(int v=0; v<imRGB.rows; ++v)
        for (int u=0; u<imRGB.cols; ++u)
        {
            unsigned int d = imD.ptr<unsigned short>(v)[u];

            if(d <= _lb || d >= _ub)
                continue;

            Eigen::Vector3d point;
            point[2] = double(d)/_scale;
            point[0] = (u-_K.at<double>(0,2))*point[2]/
                _K.at<double>(0,0);
            point[1] = (v-_K.at<double>(1,2))*point[2]/
                _K.at<double>(1,1);
            Eigen::Vector3d pointWorld =T*point;

            PointT p ;
            p.x = pointWorld[0];
            p.y = pointWorld[1];
            p.z = pointWorld[2];
            p.b = imRGB.data[v*imRGB.step+u*imRGB.channels()];
            p.g = imRGB.data[v*imRGB.step+u*imRGB.channels()+1];
            p.r = imRGB.data[v*imRGB.step+u*imRGB.channels()+2];

            _octree->updateNode(
                octomap::point3d(p.x, p.y, p.z), true);
            _octree->integrateNodeColor(
                p.x, p.y, p.z, p.r, p.g, p.b);
        }
    _octree->updateInnerOccupancy();
}
```

Appendix B

System Parameters

The parameters in Table B.1 are given in a .yaml file, from which the proposed system loads these parameters to function properly. An example of the .yaml file is given at the end of this appendix.

Table B.1: Parameter list

Parameter	description
IO.path	The working directory path. The working directory contains the TU Munich dataset used for tests. The generated trajectory, timing results, and the Octomap file are also output to this directory.
Pose.init	The initial pose of the estimated trajectory. In general, the initial pose does not influence the estimation results. But in case that the trajectory is compared with the ground-truth trajectory in 3D, the initial pose is set according to the ground-truth pose at the starting time step.
Camera.fx	The camera's intrinsic parameter f_x as in (2.3).
Camera.fy	The camera's intrinsic parameter f_y as in (2.3).
Camera.cx	The camera's intrinsic parameter c_x as in (2.3).
Camera.cy	The camera's intrinsic parameter c_y as in (2.3).
Camera.k1	The distortion coefficient k_1 in (2.5).
Camera.k2	The distortion coefficient k_2 in (2.5).
Camera.p1	The distortion coefficient p_1 in (2.5).
Camera.p2	The distortion coefficient p_2 in (2.5).
Camera.k3	The distortion coefficient k_3 in (2.5).
Depth.scale	The depth scale is used for computing the coordinates in metric units. Normally the feature point coordinates are given in pixels, as discussed in Chapter 2. On the other hand, the system requires metric output. This parameter is used for this purpose that transforming the coordinates in pixels to the coordinates in metric units.
Depth.bound	This parameter is a 2-dimensional vector which defines the upper and lower bound of the depth value. When selecting possible landmarks from the raw image, the feature points with depth outside this boundary are neglected. A small range of the depth boundary can result in a few number of landmarks being selected, which may lead to poor estimation. It is recommended to set a large range for this parameter, because the outlier rejection can also help compensating the negative effect caused by this parameter.

Table B.1: Parameter list

IO.ref	This parameter is for visually comparing different trajectories. Set this parameter to the name of a trajectory file in the working directory, the trajectory can then be plotted in the Rviz software. When comparison is not required, this parameter can be kept empty.
Camera.isShown	The option of showing the color image sequence or not. When setting to 1, the human user can see the view of the camera.
Camera.fps	the sensor image input frequency. This parameter defines the rate of the pairs of color and depth images to feed to the proposed system. Note that too high frequency can lead to few images being processed. As a result, the estimation can be very inaccurate.
ORBExtractor. numFeatures	The number of the features for the SLAM core module to extract for each frame. A larger value for this parameter increases the processing time for feature extraction, but may result in better estimation. On the other hand, if the parameter is too small, the SLAM core module may end up with an running error. This is because too few landmark measurements are obtained, which result in the phenomenon that the solution of the states to be determined is not unique.
ORBExtractor.filter	The ratio parameter of the Lowe's method to filtering out mismatched features. This parameter must be in the range of $[0, 1]$. A smaller value can lead to more robust matches, but also less results.
Noise.prior	This parameter is a 6-dimensional vector which defines the diagonal entries of the covariance matrix for the prior pose factor. In other words, this parameter defines the uncertainty of the initial pose. To be specific, the first 3 elements of this parameter determine the orientation uncertainty, while the last 3 elements determine the position uncertainty.
Noise.process	Similar to Noise.prior. This parameter defines the uncertainty of the visual odometry (the pose transformation from one frame to the next one). In this work, visual odometry is absent, thus the elements of this parameter are set to large values.
Noise.loop	Similar to Noise.process. This parameter defines the uncertainty of the loop factor (the pose transformation between two frames at greatly different time steps).
Noise.pose	Similar to Noise.prior. This parameter defines the uncertainty of the pose transformation after the node culling.
Noise.observation	This parameter is a 3-dimensional vector, which defines the uncertainty of the measurements in x, y, z directions separately.
Noise.constant	The tuning constant δ in (3.13) for outlier rejection using the Huber function.
ISAM2.horizon	The node culling horizon parameter, as discussed in Chapter 4. A larger value for this parameter can lead to longer processing time, but possibly better estimation result.

Table B.1: Parameter list

LCDetector.method	Loop detection method choice. At this moment, this parameter can only be set to 0 to enable the loop closure detection.
LCDetector.numFeatures	The number of features to be extracted for loop closure detection. In general, a larger value can give more robust loop detection results, but require longer processing time.
LCDetector.p	Previous images to be discarded when searching for a loop. This is an iBoW-LCD parameter.
LCDetector.nndr	Nearest neighbour distance ratio. This is an iBoW-LCD parameter.
LCDetector.nndr_bf	NNDR when matching with brute force. This is an iBoW-LCD parameter.
LCDetector.ep_dist	Distance to epipolar lines. This is an iBoW-LCD parameter.
LCDetector.conf_prob	Confidence probability. This is an iBoW-LCD parameter.
LCDetector.min_score	Minimal score to consider an image matching as correct. This is an iBoW-LCD parameter.
LCDetector.island_size	Maximal number of images of an island. This is an iBoW-LCD parameter.
LCDetector.min_inliers	Minimal number of inliers to consider a loop. This is an iBoW-LCD parameter.
LCDetector. nframes_after_lc	Number of frames after a loop closure to wait for new loop. This is an iBoW-LCD parameter.
LCDetector. min_consecutive_loops	Minimal consecutive loops to avoid epipolar geometry. This is an iBoW-LCD parameter.
Octomap.resolution	The minimal cell size of the Octomap. The smaller the size, the more accurate the environment can be modeled, but the generated map requires larger memory space.

An example of the .yaml settings file for the dataset fr1/floor is given below.

```
#--------------------------------------------------------
# Common
#--------------------------------------------------------

# path of the input dataset and the output files
IO.path: "dataset_path/rgbd_dataset_freiburg1_floor/"

# initial pose (obtained from the ground-truth file)
Pose.init: [1.4906,-1.1681,0.6610,0.8959,0.0713,-0.0460,-0.4361]

# intrinsic parameters
Camera.fx: 517.306408
Camera.fy: 516.469215
Camera.cx: 318.643040
Camera.cy: 255.313989

# distortion parameters
Camera.k1: 0.262383
Camera.k2: -0.953104
Camera.p1: -0.005358
Camera.p2: 0.002628
Camera.k3: 1.163314

# Depth map values factor
```

```
Depth.scale: 5000.0
Depth.bound: [0.0, 20000.0]

#--------------------------------------------------------
# Sensor
#--------------------------------------------------------

# reference file
IO.ref: ["CameraTrajectory.txt"]

# show color image sequence
Camera.isShown: 0 # 0: not shown, 1: shown

# Camera frames per second
Camera.fps: 10.0

#--------------------------------------------------------
# SLAM core
#--------------------------------------------------------

# number of extracted features
ORBExtractor.numFeatures: 1000
# filter parameter
ORBExtractor.filter: 0.6

# noise params
Noises.prior:       [0.05, 0.05, 0.05, 0.01, 0.01, 0.01]
Noises.process:     [1000, 1000, 1000, 1000, 1000, 1000]
Noises.loop:        [0.005, 0.005, 0.005, 0.001, 0.001, 0.001]
Noises.pose:        [0.005, 0.005, 0.005, 0.001, 0.001, 0.001]
Noises.observation: [0.01, 0.01, 0.015]
Noises.constant: 0.8

ISAM2.horizon: 50

ISAM2.thres_x: [0.01, 0.01, 0.01, 0.01, 0.01, 0.01]
ISAM2.thres_l: [0.01, 0.01, 0.01]

SLAM.thres_l: 5

#--------------------------------------------------------
# Loop closure
#--------------------------------------------------------

# choose loop closure detection method
LCDetector.method: 0 # 0: ibow-lcd 1: classical bow (not used)
LCDetector.numFeatures: 1500

# ibow-lcd params
LCDetector.p: 250
LCDetector.nndr: 0.8
LCDetector.nndr_bf: 0.8
LCDetector.ep_dist: 2.0
LCDetector.conf_prob: 0.985
```

```
LCDetector.min_score: 0.3
LCDetector.island_size: 7
LCDetector.min_inliers: 60
LCDetector.nframes_after_lc: 3
LCDetector.min_consecutive_loops: 5

#--------------------------------------------------------
# Dense mapping
#--------------------------------------------------------

Octomap.resolution: 0.5
```