POLITECNICO DI MILANO

Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Automation and Control Engineering

Simultaneous Localization and Mapping in Urban Scenarios with Non-Rigid OpenStreetMap Priors

Advisor: PROF. MATTEO MATTEUCCI

Co-advisor: MATTEO FROSI

Master Graduation Thesis by:

VERONICA GOBBI

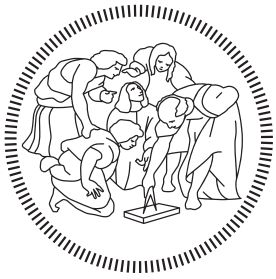
Student Id n. 921095

Academic Year 2020-2021

POLITECNICO DI MILANO

Facoltà di Ingegneria
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Automation and Control Engineering



Simultaneous Localization and Mapping in Urban Scenarios with Non-Rigid OpenStreetMap Priors

Relatore: PROF. MATTEO MATTEUCCI

Correlatore: MATTEO FROSI

Tesi di Laurea Magistrale di:

VERONICA GOBBI
Matricola n. 921095

Anno Accademico 2020-2021

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

This template has been adapted by Emanuele Mason, Andrea Cominola and Daniela Anghileri: *A template for master thesis at DEIB*, June 2015. This version of the paper has been later adapted by Marco Cannici, March 2018.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Prof. Matteo Matteucci, and my co-advisor, Matteo Frosi, for their patience with me and their continuous support.

Thanks to my parents, that always supported me unconditionally in every new adventure, and thanks to my sister, for her contagious determination and continuous encouragement. If it weren't for them, this thesis wouldn't be here. Thanks to all my relatives, that years ago have given me the ticket to continue this journey, and thanks to all the friends that have accompanied me through it.

Finally, thanks to all the people on the mountains, that has always sustained, kept company, and fed me really well.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

An essential ability of mobile robots is to build an accurate map of the surrounding environment and localize themselves precisely into it. This allows the robots to be able to operate reliably in complex or unknown environments using only their own perceptions. This is known, in literature, as the Simultaneous Localization and Mapping (SLAM) problem.

Over the last decade, many approaches have been proposed to solve the SLAM problem, depending on the type of data gathered by the robot. Particularly accurate and well-known are SLAM systems which use LiDAR data and model the problem as a graph optimization problem (Graph SLAM). Recently, few LiDAR Graph SLAM systems have been proposed, which exploit external maps to insert new information into the graph. These approaches extract features from the external map and associate them with LiDAR perceptions to obtain information about the position of the robot with respect to the features. These approaches are not able to recognize eventual mistakes or misplacements of the features, thus their performance and the correctness of information inserted into the graph could be greatly reduced. In this thesis we introduce three versions of a LiDAR Graph SLAM system, which is able to precisely localize the robot and create an accurate map of the environment, by using information about the surrounding buildings, obtained from maps. The systems are also able, on various degrees, to correct the pose of the buildings that are misplaced into the map. The presented systems are based on an already existing LiDAR Graph SLAM system, called hdl_graph_slam.

The first system we propose, called hdl_graph_slam_with_map_priors, introduces a new type of constraint in the graph between a pose of the robot and a building, based on the alignment between the map associated to the buildings and a LiDAR scan. It is able to correct the poses of the robot from drifting but not to correct the poses of the buildings.

The second system is called building_slam_with_buildings_priors Rigid SLAM. With respect to the first system (from which it derives), it improves the calculation of the information matrices of the constraints in order to better fit every single building. Differently from the first system, building_slam-_with_buildings_priors Rigid SLAM is able to correct also the position of buildings, but not precisely.

The last system presented is called building_slam_with_buildings_priors Non-rigid SLAM and constitutes a further improvement of the second system. It improves the calculation of the alignment transformation to adjust perfectly

to each building. It is able to reach a good accuracy in the correction of poses of the buildings.

To test the proposed systems, we used one trajectory from the KITTI dataset, evaluating both accuracy metrics and visual aspects of the obtained trajectories and maps.

SOMMARIO

La navigazione autonoma dei robot è un argomento sempre più studiato nell'ultimo decennio. Molti compiti, che sono ripetitivi o pericolosi per gli esseri umani, possono essere svolti per mezzo di robot mobili non operati da essere umani. Alcuni esempi sono i robot mobili per la pulizia e gli aspirapolvere automatizzati, i veicoli a guida automatica per l'agricoltura e l'allevamento, i robot mobili per la consegna e il trasporto, su strada ma anche all'interno di fabbriche e magazzini, e la ricerca e il soccorso in zone pericolose. Questi robot devono essere costruiti in maniera tale che siano in grado di operare in modo sicuro e affidabile in ambienti complessi e/o sconosciuti, basandosi solo sulle percezioni dei loro sensori. Hanno bisogno di avere una mappa molto accurata di tali ambienti e di localizzarsi in essi.

Il problema chiamato Simultaneous Localization and Mapping (SLAM) affronta il problema di costruire un modello dell'ambiente che circonda il robot, chiamato mappa, e contemporaneamente stimare lo stato del robot che si muove al suo interno. Questo significa localizzare il robot rispetto alla mappa, poiché lo stato del robot è generalmente descritto dalla sua posa, che rappresenta la posizione e l'orientamento, anche se altre quantità possono essere incluse. La mappa descrive vari aspetti dell'ambiente in cui il robot opera, come la posizione di punti di riferimento, ostacoli o edifici, ed è necessaria per localizzarsi con precisione ed affidabilità. Fare affidamento solo sulla percezione del robot nell'istante di tempo corrente, senza considerare le informazioni passate incluse nella mappa, porterebbe, nel tempo, a una stima errata della posizione del robot. Inoltre, la mappa può essere necessaria anche per altri compiti, come la pianificazione del percorso.

Dalla fine degli anni ottanta ad oggi, molti algoritmi sono stati proposti per risolvere questo problema, permettendo ai robot mobili di navigare correttamente. Gli algoritmi più famosi si basano su una formulazione probabilistica del problema di SLAM e includono approcci basati su Extended Kalman FIlters (EKF) [35], Rao-Blackwellized particle filters [54] e metodo della massima verosimiglianza [55]. Più recentemente, sono state proposte applicazioni basate su telecamere e laser range scanner (LiDAR). I sistemi di LiDAR SLAM si basano sulla misure provenienti da un LiDAR. Questi sensori producono scansioni molto precise, in 2D o 3D, dell'ambiente, permettendo una localizzazione accurata del robot e/o la costruzione della mappa. Come output, i LiDAR danno insiemi di punti ad alta frequenza, che sono più semplici da associare l'uno con l'altro per localizzare il robot, rispetto alle immagini prodotte da una telecamera. Quindi i sistemi di LiDAR SLAM sono più robusti rispetto ad altri sistemi SLAM. Inoltre, i sensori LiDAR stanno

diventando sempre più economici, quindi oggi è facile integrarne uno su un robot mobile.

Una famiglia di algoritmi che risolvono il problema dello SLAM, basati sull'ottimizzazione sparsa non lineare, è formata dai metodi basati sui grafi, conosciuti anche come Graph SLAM. Si basano sulla rappresentazione del problema dello SLAM come un grafo (chiamato grafo delle pose), dove i nodi rappresentano pose del robot o punti di riferimento della mappa, e dove gli spigoli rappresentano vincoli tra i nodi, derivanti dalle percezioni dei sensori. Per imporre la coerenza fra i nodi, il grafo viene ottimizzato, sfruttando le informazioni derivate dagli spigoli. Dopo l'ottimizzazione, il grafo soddisfa tutti i vincoli imposti fra i nodi, con un errore minimo, avendo anche regolato le pose dei possibili punti di riferimento, creando così una mappa dell'ambiente. Un esempio di un grafo delle pose può essere visto in Figura 0.1, dove i punti rossi sono i nodi del grafo di posa, le linee nere sono gli spigoli e i punti rosa corrispondono a punti di riferimento associati a edifici del mondo reale.

Per raggiungere una grande accuratezza nello SLAM, negli ultimi anni sono stati proposti molti lavori che sfruttano una mappa esterna precompilata (di solito in altri formati rispetto a quella calcolata dallo SLAM). L'idea alla base di questi lavori è che allineando una mappa interna con una scansione LiDAR, si ottengono informazioni significative sulla posa del robot (poiché la scansione LiDAR è associata ad essa), rispetto alle caratteristiche presenti nella mappa (le caratteristiche si riferiscono a posizioni geografiche, poiché sono estratte da una mappa). La mappa esterna precompilata può essere di molti tipi, per esempio geodati in formato XML o JSON, o immagini aeree. Questi dati devono essere elaborati per estrarne le caratteristiche e successivamente costruire la mappa interna che verrà utilizzata per l'allineamento. Anche se accurati, i sistemi presentati soffrono di eventuali errori presenti nella mappa esterna precompilata. Se le caratteristiche estratte sono diverse da quelle riportate dalla scansione LiDAR, allora l'allineamento può fallire e quindi la localizzazione del robot e la mappa dell'ambiente ricostruita possono essere imprecise. Per esempio, se una edificio è stato modificato ma l'immagine aerea corrispondente non è stata aggiornata, allora l'allineamento può fallire.

CONTRIBUTO DELLA TESI

Per risolvere i problemi associati allo SLAM con mappe esterne, menzionati sopra, in questa tesi presentiamo un nuovo metodo per fare Graph SLAM con i map priors, insieme a tre sistemi basati su di esso. Il contributo principale della tesi è quello di introdurre una nuova modalità di calcolare i vincoli per un sistema di Graph SLAM, che può essere montato su un robot mobile. Questi vincoli sono basati sulla conoscenza delle informazioni derivate dalle

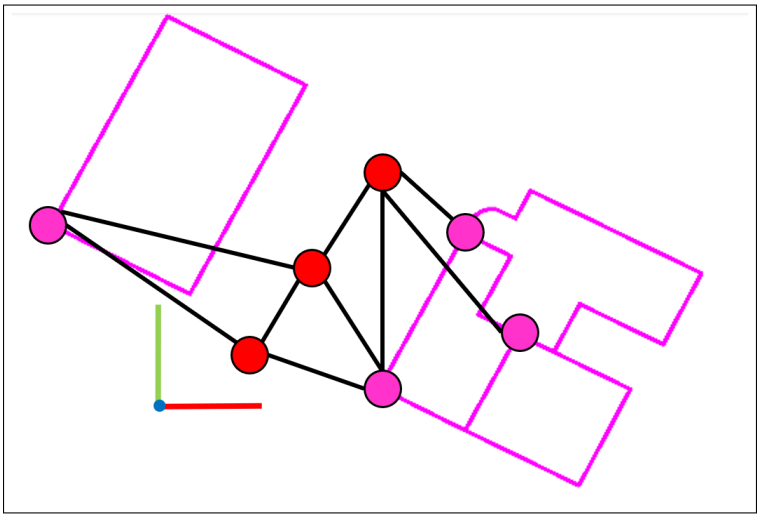


Figura 0.1: Edifici e grafo delle pose. Le linee rosa sono i contorni degli edifici; i punti rossi sono i nodi associati alle pose del robot; i punti rosa sono i nodi associati alle pose degli edifici; le linee nere sono gli spigoli fra i nodi.

mappe disponibili. Le mappe possono essere facilmente ottenute da servizi pubblicamente disponibili come OpenStreetMap o Bing Maps. Essi forniscono una vasta gamma di geodati, che sono informazioni relative ad una posizione sulla terra. Esempi sono i dati amministrativi, come confini e località, i dati riguardanti i trasporti, come strade, ferrovie e aeroporti, i dati di elevazione, i dati riguardanti i rilievi, i dati ambientali e i dati idrografici. Usiamo i geodati per introdurre un nuovo vincolo sugli edifici. Recuperiamo da OpenStreetMap dati geografici sugli edifici, dove si trovano sulla terra, e dati geometrici su di essi, i loro contorni visti dall'alto.

Al giorno d'oggi, i robot mobili sono solitamente dotati di vari sensori. Come detto prima, i sensori LiDAR stanno diventando di uso comune, grazie alla loro precisione e al costo relativamente basso. Questi dispositivi, attraverso l'uso di un laser, forniscono una rappresentazione dei dintorni del robot, sia in 2D che in 3D, come un insieme di punti, chiamati point cloud. Una misurazione fornita da un LiDAR viene anche chiamata scansione (in quanto per raccogliere i dati, il sensore ruota su se stesso, scansionando l'ambiente). Uno dei tanti vantaggi dell'adozione dei LiDAR come sensori principali è la possibilità di eseguire lo scan matching, tra point clouds, per stimare il movimento del robot: cercando di sovrapporre nel miglior modo possibile le scansioni LiDAR prese in due pose consecutive del robot, si può ottenere la trasformazione relativa tra esse. È un modo piuttosto semplice di localizzare il robot, che può essere soggetto nel tempo ad imprecisioni della pose stimate. Abbiamo già detto che nel grafo delle pose del sistema di Graph SLAM, ogni nodo rappresenta una posa del robot, in un certo istante di tempo. Nella Figura 0.1 tali nodi sono i punti rossi, e le linee nere

Figura 0.2: Allineamento tra la scansione del LiDAR e la mappa degli edifici. Le linee rosa sono i contorni degli edifici; in arancione, la scansione LiDAR prima dell'allineamento; in azzurro la scansione LiDAR dopo l'allineamento; la freccia viola rappresenta la trasformazione tra la scansione LiDAR e la mappa degli edifici.

tra due punti rossi sono gli spigoli dati dal fare lo scan matching. Anche ogni edificio è rappresentato nel grafo come un nodo. Per ogni edificio, prendiamo un riferimento, per esempio un angolo, e il nodo dell'edificio nel grafo delle pose rappresenta la posa di tale riferimento, quindi rappresenta la posa dell'edificio. Nella Figura 0.1 gli edifici sono le linee rosa, e i nodi corrispondenti sono i punti rosa.

Data una posa del robot, otteniamo tutte le informazioni associate agli edifici e la scansione LiDAR dell'ambiente circostante. I dati sugli edifici sono usati per inserire nuovi nodi nel grafo delle pose, uno per ogni edificio, e sono usati per costruire una mappa dell'ambiente, che contiene gli edifici e ne fornisce una rappresentazione spaziale, come una mappa standard visibile da qualsiasi servizio di mappe online. Questa mappa è l'insieme di tutti gli edifici rosa, come si può vedere nella Figura 0.1 o nella Figura 0.2. La scansione LiDAR è allineata con la mappa degli edifici, che vuol dire che la scansione LiDAR è spostata e ruotata per cercare di farla coincidere nel miglior modo possibile con la mappa degli edifici. Otteniamo una trasformazione che, se applicata alla scansione LiDAR, la rende allineata con la mappa degli edifici. Questo meccanismo è descritto nella Figura 0.2: la scansione LiDAR prima dell'allineamento è in arancione, mentre la mappa degli edifici è

rappresentata dagli edifici rosa. Eseguiamo l'allineamento e otteniamo la scansione LiDAR allineata, che è la point cloud azzurra. La trasformazione ottenuta è rappresentata visivamente dalla freccia viola.

Poiché la scansione del LiDAR è associata ad una specifica posa del robot, possiamo collegare tale posa del robot con gli edifici attraverso l'uso di questa trasformazione. Infatti questa è usata per inserire uno spigolo fra il nodo della posa del robot e i nodi degli edifici. Questi spigoli sono visibili nella Figura 0.1 come le linee nere che collegano i punti rosa e i punti rossi. Poiché gli edifici sono georeferenziati, significa che stiamo fornendo al robot informazioni utili per localizzarsi meglio. Allo stesso tempo, gli edifici costituiscono la mappa dell'ambiente. Poiché gli edifici sono nodi nel grafo, allora stiamo anche fornendo informazioni per migliorare la precisione della mappa ricostruita dell'ambiente, il che significa che miglioriamo la precisione nel posizionamento degli edifici che sono non sono posizionati correttamente in OpenStreetMap. I meccanismi descritti sopra costituiscono il primo sistema che introduciamo all'interno di questa tesi, chiamato hdl_graph_slam_with_map_priors. Questo sistema fa da base per gli altri due.

Il secondo sistema è chiamato building_slam_with_buildings_priors Rigid SLAM. Rappresenta un miglioramento rispetto al primo sistema per quanto riguarda la precisione delle posizioni degli edifici. Quando inseriamo uno spigolo nel grafico, ci viene richiesto di specificare la precisione di quello spigolo attraverso l'uso di una matrice, chiamata matrice di informazione. Questa matrice dice quanto ci fidiamo che la trasformazione associata ad uno spigolo sia corretta. Nel primo sistema, la matrice di informazione era la stessa per ogni spigolo che collegava gli edifici e una posa del robot. In building_slam_with_buildings_priors Rigid SLAM, siamo in grado di calcolare una matrice di informazione diversa e su misura per ogni edificio. L'ottimizzazione è in grado di capire di quali bordi fidarsi di più e quali bordi diffidare, migliorando così il posizionamento degli edifici.

Il terzo sistema si chiama building_slam_with_buildings_priors Non-rigid SLAM e mantiene tutte le innovazioni introdotte nel secondo sistema. In questo sistema, invece di fare l'allineamento tra tutti gli edifici e la scansione LiDAR, si eseguono più allineamenti, uno per edificio, ottenendo così trasformazioni più precise degli altri sistemi. Se nel secondo sistema, solo il calcolo delle matrici di informazione dei bordi era stato migliorato, ora anche il calcolo delle trasformazioni che costituiscono gli spigoli è stato migliorato. Otteniamo una correzione ancora più accurata degli edifici mal posizionati.

Riassumendo, in questa tesi presentiamo due contributi principali rispetto allo stato dell'arte nello SLAM con map priors, concentrandoci sui sistemi di LiDAR Graph SLAM. In primo luogo, presentiamo un nuovo approccio per unire i prodotti dello SLAM (mappa e traiettoria stimata) e le mappe esterne. Questo ci permette di ottenere una stima precisa della posa del robot

utilizzando solo la stima dell'odometria data dal LiDAR, la chiusura dei loops e i map priors. Non c'è bisogno di usare GPS o IMU, che sono costosi e consumano energia, e a volte non sono disponibili o non sono accurati (come nei tunnel o dove non c'è il servizio GNSS). La principale novità introdotta è che usiamo la conoscenza degli edifici dalle mappe, e integriamo un edificio non solo come una semplice misurazione associata a una posa del robot, ma come un'entità esistente da sola che può essere vista da molte pose del robot. Un secondo contributo, oltre a localizzare meglio il robot, è che anche la mappa può essere migliorata. In particolare, miglioriamo la posizione e l'orientamento degli edifici. Gli edifici scaricati da OpenStreetMap possono soffrire di imprecisioni a causa dell'imprecisione dei dispositivi GPS utilizzati per la mappatura e a causa di errori umani nella raccolta dei dati e nel caricamento di tali su internet. A causa dell'allineamento, le pose degli edifici vengono corrette per essere coerenti con le scansioni LiDAR. Essendo il dispositivo LiDAR di solito molto preciso, si assume che esse siano corrette. Questo rende il sistema in grado di costruire una mappa ancora più precisa e di localizzarsi il robot in una maniera più accurata nell'ambiente. I sistemi possono anche essere usati solo per correggere le mappe: invece di dover rimappare ogni singolo edificio uno per uno, è sufficiente mandare un robot mobile per le strade. Tutti gli edifici che vengono visti dal robot possono quindi essere corretti.

In tutti e tre i sistemi, gli edifici vengono scaricati, analizzati e inseriti nel grafico poco prima di eseguire l'ottimizzazione. Questa procedura viene fatta per tutte le pose del robot che non sono mai state associate ad edifici. Per ogni posa, scarichiamo gli edifici che la circondano. I dati vengono scaricati in formato XML da OpenStreetMap, vengono analizzati ed inseriti in un vettore di oggetti, ognuno dei quali rappresenta un edificio. Per ognuno di questi edifici, un nodo corrispondente viene inserito nel grafo delle pose. Tutti gli edifici sono uniti per creare una mappa degli edifici. Successivamente, viene fatto l'allineamento tra la scansione LiDAR associata alla posa del robot e la mappa degli edifici. Da tale procedura, si ottiene una trasformazione. In hdl_graph_slam_with_map_priors e building_slam_with_buildings_priors Rigid SLAM, per ogni edificio associato ad una posa del robot, questa trasformazione viene utilizzata per costruire una trasformazione relativa tra la posa del robot e la posa dell'edificio. Questa trasformazione relativa viene utilizzata per costruire uno spigolo che viene inserito nel grafo delle pose, fra il nodo associato alla posa del robot e il nodo dell'edificio. A partire da un singolo nodo associato a una posa del robot, ci possono essere più spigoli che lo collegano a molti nodi di edifici che si trovano nei suoi dintorni, mentre un nodo di un edificio può avere più spigoli provenienti da diversi nodi associati a diverse pose del robot che vedono l'edificio nei loro dintorni. In hdl_graph_slam_with_map_priors, la matrice di informazione associata ad uno spigolo è la stessa per ogni spigolo proveniente dalla stessa posa

del robot verso diversi edifici. In building_slam_with_buildings_priors Rigid SLAM, siamo in grado di calcolare una matrice di informazione diversa per ogni spigolo tra la stessa posa del robot e diversi edifici nei suoi dintorni.

In building_slam_with_buildings_priors Non-rigid SLAM, invece, la trasformazione è usata per inizializzare una seconda procedura di allineamento. Per ogni edificio, eseguiamo un allineamento tra la singola mappa dell'edificio e la scansione LiDAR associata alla posa del robot. Otteniamo una seconda trasformazione, specifica per ogni edificio, che viene utilizzata per calcolare la trasformazione relativa fra quell'edificio e la posa del robot. Questa trasformazione relativa viene utilizzata per inserire uno spigolo nel grafo delle pose, fra il nodo associato alla posa del robot e il nodo dell'edificio. La matrice di informazione è calcolata nello stesso modo di building_slam_with_buildings_priors Rigid SLAM, in maniera specifica per ogni edificio, ma utilizzando la seconda trasformazione associata ad esso.

DESCRIZIONE DELLA TESI

La tesi è così composta:

- Il capitolo 2 presenta una spiegazione dei principali strumenti e metodologie utilizzati nella tesi: SLAM e Graph SLAM, gli algoritmi di registrazione utilizzati per l'allineamento e OpenStreetMap. Viene anche fornita una revisione dei lavori presenti in letteratura che sono alla base di questa tesi, descrivendo come introdurre map priors in un sistema di Graph SLAM

- Il capitolo 3 presenta il sistema di SLAM chiamato hdl_graph_slam, che è un sistema già esistente usato come base per il resto del lavoro presentato. Descriviamo i suoi principali componenti e meccanismi, in particolare quelli che vengono successivamente modificati per introdurre un nuovo tipo di vincolo

- Il capitolo 4 descrive un sistema SLAM chiamato hdl_graph_slam_with_map_priors. Introduce in hdl_graph_slam gli edifici come entità nel grafo di posa e come fonte di informazioni per un nuovo tipo di vincolo. Descriviamo l'intero flusso di lavoro del sistema, dal download degli edifici, all'allineamento tra una scansione e la mappa degli edifici, all'inserimento nel grafo delle pose di nodi e spigoli

- Il capitolo 5 è diviso in due sezioni, ognuna delle quali presenta una modifica di hdl_graph_slam_with_map_priors, che sono mirate alla correzione degli edifici che sono mal posizionati su OpenStreetMap. La prima sezione, che descrive un sistema chiamato building_slam_with_buildings_priors Rigid SLAM, introduce una modalità di correzione

degli edifici che si basa sull'allineamento "globale" tra la scansione LiDAR e la mappa degli edifici, quindi ancora soggetto a grosse imprecisioni. La seconda sezione, che presenta un sistema chiamato building_slam_with_buildings_prior Non-rigid SLAM, introduce un altro modo di correggere gli edifici basato su un allineamento "locale" tra la scansione LiDAR e ogni singolo edificio, quindi più preciso della metodologia introdotta nella prima sezione

- Il capitolo 6 presenta i risultati dell'esecuzione del sistema con una serie di dati pre-raccolti. Descriviamo i dati e gli strumenti utilizzati per valutare i risultati. Vengono presentati e discussi i dati numerici sulla correttezza della posa finale del robot rispetto alla ground truth del terreno. Infine, una valutazione visiva dettagliata mostra come il sistema sia in grado di correggere le pose degli edifici

- Il capitolo 7 è una panoramica del lavoro presentato nella tesi e include possibili miglioramenti da effettuare in futuro

1 INTRODUCTION

1.1 MOTIVATION

Autonomous robot navigation has been an increasingly studied topic in the last decade. A lot of tasks, which are repetitive or hazardous for humans, may be carried out by means of unmanned mobile robots. Examples include mobile robots for cleaning and automated vacuums, agriculture and farming automated driving vehicles, mobile robots for delivery and transportation, on the road but also inside factories and warehouses, and search and rescue in dangerous zones. These robots need to be built in a way which allows them to operate safely and reliably in complex and/or unknown environments, based only on the perceptions of their onboard sensors. They need a very accurate map of said environment and they need to localize themselves into it.

The Simultaneous Localization and Mapping (SLAM) problem addresses the problem of constructing a model of the environment surrounding the robot, called map, while simultaneously estimating the state of the robot moving within it. This means localizing the robot with respect to the map, since the state of the robot is generally described by its pose, representing position and orientation, although other quantities may be included. The map may describe various aspects of the environment in which the robot operates, such as the position of landmarks, obstacles, or buildings, and it is needed for the robot to localize itself with accuracy and reliability. Just relying on the perception of the robot at the current time instant, without considering the past information included in the map, would lead, over time, to a wrong estimation of the pose of the robot. Moreover, the map may also be needed for other tasks, such as path planning.

From the end of the eighties until now, many algorithms have been proposed to solve this problem, allowing mobile robots to navigate properly. The most famous algorithms are based on a probabilistic formulation of the SLAM problem and include approaches based on extended Kalman filters (EKF) [35], Rao-Blackwellized particle filters [54], and maximum likelihood estimation [55]. More recently, applications based on cameras and laser range scanners (LiDAR) have been proposed. LiDAR SLAM systems are based on perception from a LiDAR. These sensors produce very precise 2D or 3D scans of the environment, allowing for accurate localization of the robot and/or construction of the map. As output, they produce sets of points at a high frequency, which are easier to associate one with the other in

order to localize the robot, with respect to images produced by a camera. Thus LiDAR SLAM systems are more robust with respect to other SLAM systems. Moreover, LiDAR sensors are becoming increasingly cheap, thus it is convenient nowadays to integrate one of them onto a mobile robot.

A family of algorithms which solve the SLAM problem, based on nonlinear sparse optimization, is formed by graph-based methods, also known as Graph SLAM. They are based on the representation of the SLAM problem as a graph (called pose graph), where nodes represent poses of the robot or landmarks of the map, and where edges represent constraints between nodes, arising from perceptions of sensors. To enforce coherence between the nodes, the graph is optimized, exploiting the information derived from the edges. After the optimization, the graph satisfies all the imposed constraints between nodes, with a minimum error, while also having adjusted the poses of possible landmarks, thus creating a map of the environment. An example of a pose graph can be seen in Figure 1.1, where the dots are the nodes of the pose graph, the black lines are the edges and the pink nodes correspond to landmarks associated to real world buildings.

To achieve great accuracy in SLAM, over the past years many works have been proposed, which exploit an external pre-computed map (usually in other formats than the one computed by SLAM). The idea behind this works is that by aligning an internal map with a LiDAR scan, one can obtain meaningful information about the pose of the robot (since the LiDAR scan is associated with it), with respect to the features presents in the map (features refer to geographic positions, because they are extracted from a map). The external pre-computed map may be provided in many ways, such as geodata in the XML or JSON format, or such as aerial images. These data need to be processed in order to extract features from them and subsequently build the internal map that will be used for the alignment. Although accurate, the presented systems suffer from eventual errors present in the external pre-computed map. If features are different from what the LiDAR scan reports, then alignment may fail and thus the localization of the robot and the reconstructed map of the environment may be inaccurate. For example, if a building has been modified but the corresponding aerial image has not been updated, then the alignment may fail.

1.2 THESIS CONTRIBUTIONS

To solve the problems associated with SLAM aided with external maps, mentioned above, in this thesis we present a new method for Graph SLAM with map priors, along with three systems based on it. The main contribution of the thesis is to add a new way of computing constraints for a Graph SLAM system, which can be mounted on a mobile robot. These constraints are based on the information derived from available maps. Maps can be easily retrieved

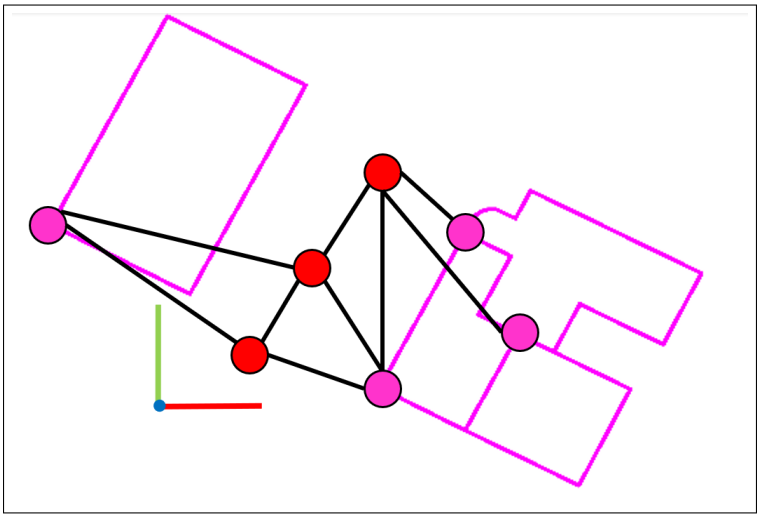


Figure 1.1: Example of buildings and pose graph. The pink lines are the outlines of the buildings; the red dots are the nodes associated with the poses of the robot; the pink dots are the nodes associated with the poses of the buildings; the black lines are the edges between nodes.

from publicly available services such as OpenStreetMap or Bing Maps. They provide a large array of geodata, which is information related to a location on the Earth. Examples are administrative data, such as boundaries and locations, transportation data, such as roads, railways, and airports, elevation data about terrains and reliefs, environmental data, and hydrography data. We use the geodata to introduce a new constraint about buildings. We retrieve from OpenStreetMap geographic data about the buildings, where they are located on the Earth, and geometric data about them, their outlines as seen from above.

Nowadays, mobile robots are usually provided with multiple sensors. As stated before, laser rangefinder sensors (LiDARs) are becoming common, due to their accuracy and relatively low cost. These devices, through the use of a laser, provide a representation of the surroundings of the robot, either in 2D or 3D as a set of points, called point cloud. A measurement provided by a LiDAR is also called scan (as to gather data, the sensor rotates around itself, scanning the environment). One of the many advantages of adopting LiDARs as main sensors is the possibility to perform scan matching, between point clouds, to estimate the motion of the robot: by trying to overlap in the best way possible LiDAR scans taken at two consecutive poses of the robot, the relative transformation between them, can be retrieved. It is a simple way of localizing the robot, which may be subjected to the drift of the estimated pose over time. We already said that in the pose graph of the Graph SLAM system, each node represents a pose of the robot, at a certain time instant. In Figure 1.1 such nodes are the red dots, and the black lines between two red

Figure 1.2: Example of alignment between LiDAR scan and buildings map. The pink lines are the outlines of the buildings; in orange, there is the LiDAR scan before the alignment; in light blue is the LiDAR scan after the alignment; the purple arrow represents the transformation between the LiDAR scan and the map of the buildings.

dots are edges given by the scan matching. Each building is also represented in the graph as a node. For each building, we take a reference, such as a corner, and the building node in the pose graph represents the pose of such reference, thus the pose of the building. In Figure 1.1 we can see the buildings as the pink lines, and the corresponding nodes as the pink dots.

Given a pose of the robot, we retrieve all the information associated with the buildings and the LiDAR scan of the surrounding environment. Data about buildings is used to insert new nodes in the pose graph, one for each building, and it is used to build a map of the environment, which contains the buildings and yields a spatial representation of them, such as a standard map visible from any online map service. This map is the collection of all the pink buildings, as can be seen in Figure 1.1 or in Figure 1.2. We align the LiDAR scan with the map of the buildings, meaning we move and rotate the LiDAR scan to try to make it coincides in the best way possible with the map of the buildings. We obtain a transformation that, if applied to the LiDAR scan, makes it aligned with the map of the buildings. This mechanism is described in Figure 1.2: the LiDAR scan before the alignment is in orange, while the map of the buildings is represented by the pink buildings. We perform the alignment and obtain the rotated and translated LiDAR scan,

which is the light blue point cloud. The transformation obtained is visually represented by the purple arrow.

Since the LiDAR scan is associated with a specific pose of the robot, we can link the pose with the buildings through the use of this transformation. Indeed this transformation is used to insert an edge between the robot pose node and the nodes of the buildings. These edges can be seen in Figure 1.1 as the black lines connecting pink dots and red dots. Since the buildings are geo-referenced, it means that we are providing the robot useful information to localize itself better. At the same time, the buildings constitute the map of the environment. Since the buildings are nodes in the graph, then we are also providing information to improve the accuracy of the reconstructed map of the environment, which means, we improve the accuracy in the positioning of buildings that are misplaced. The mechanisms described above constitute the first system that we introduce within this thesis, called hdl_graph_slam_with_map_priors. This system acts as a foundation for the other two.

The second system is called building_slam_with_buildings_priors Rigid SLAM. It represents an improvement with respect to the first system, regarding the accuracy of the positions of buildings. When we insert an edge into the graph, we are required to specify the accuracy of that edge through the use of a matrix, called information matrix. This matrix tells how much we trust the transformation associated with an edge to be correct. In the first system, the information matrix was the same for each edge connecting buildings and a pose of the robot. In building_slam_with_buildings_priors Rigid SLAM, we are able to compute an information matrix different and tailored to each building. The optimization is able to understand which edges to trust more and which edges to wary off, thus improving the positioning of buildings.

The third system is called building_slam_with_buildings_priors Non-rigid SLAM and maintains all the innovations introduced in the second one. In this system, instead of doing the alignment between all the buildings and the LiDAR scan, we perform many alignments one for each building, thus obtaining more precise transformations than the other systems. While in the second system the computation of information matrices of edges was improved, now also the computation of transformations underlying the edges is improved. With this system, we obtain an even more accurate correction of misplaced buildings.

Summing up, in this thesis we present two main contributions with respect to the state of the art in SLAM with map priors, focusing on LiDAR Graph SLAM systems. First, we present a new approach to fuse SLAM products (map and estimated trajectory) and existing external maps. This allows us to achieve a precise estimation of the pose of the robot by using only LiDAR odometry estimation, loop closure, and map priors. There is no need to use

GPS or IMU, which are expensive and energy consuming, and sometimes are unavailable or not accurate (e.g., in tunnels or GNSS-denied zones). The main novelty introduced is that we use the information of buildings from maps, and integrate a building not only as a mere measurement associated with a pose of the robot, but as an entity existing by itself, which can be seen from many poses of the robot. A second contribution, apart from localizing the robot better, is that also the map can be improved. In particular, we improve the position and the orientation of the buildings. The buildings downloaded from OpenStreetMap may suffer from inaccuracies because of imprecision in the GPS devices used to map the buildings and because of human errors in the collection of data and in the upload. Because of the alignment, the buildings poses are corrected in order to be coherent with the LiDAR scans. Being the LiDAR sensor usually really precise, we assume them to be correct. This makes the system able to build an even more precise map and to localize the robot in a more accurate way in the environment. The systems may even be used just to correct the maps: instead of having to remap each building one by one, it is enough to send out a mobile robot on the streets. All the buildings that are seen by the robot can be then corrected.

In all three systems, buildings are downloaded, parsed, and inserted into the graph shortly before performing optimization. This procedure is done for all the poses of the robot that has never been associated with buildings. For each pose, we download the buildings surrounding it. Data is downloaded as an XML from OpenStreetMap, which is parsed into a set of objects, each one representing a building. For each of these buildings, a corresponding node is inserted into the pose graph. All the buildings are put together to create a map of the buildings. Next, the alignment is done between the LiDAR scan associated with the pose of the robot and the map of the buildings. From it, we obtain a transformation. In hdl_graph_slam_with_map_priors and building_slam_with_buildings_priors Rigid SLAM, for each building associated with the robot pose, this transformation is used to build a relative transformation between the pose of the robot and the pose of the building. This relative transformation is used to build an edge that is inserted into the pose graph, between the node associated with the pose of the robot and the node of the building. Spanning from a single node associated to a pose of the robot there may multiple edges connecting it to many nodes of buildings that are in its surroundings, and a building node may have edges coming from different nodes associated to poses of the robot which see the building in their surroundings. In hdl_graph_slam_with_map_priors, the information matrix associated with an edge is the same for every edge coming from the same pose of the robot towards different buildings. In building_slam_with_buildings_priors Rigid SLAM, we are able to compute a different information matrix for every edge between the same pose of the robot and different buildings in its surroundings.

In building_slam_with_buildings_priors Non-rigid SLAM, instead, the transformation is used to initialize a second alignment procedure. For each building, we perform an alignment between the single building map and the LiDAR scan associated with the robot pose. We obtain a second transformation, specific for each building, that is used to compute the relative transformation between that building and the pose of the robot. This relative transformation is used to insert an edge into the pose graph between the node associated with a pose of the robot and the building node. The information matrix is computed in the same way as building_slam_with_buildings_priors Rigid SLAM, specific for each building, but using the second transformation associated to it.

1.3 THESIS OUTLINE

The thesis is structured in the following way:

- Chapter 2 presents an explanation of the main tools and methodologies used in the thesis: SLAM and Graph SLAM, the registration algorithms used for the alignment, and OpenStreetMap. A review of works found in literature, which are at the base of this thesis, is also provided, describing how to introduce map priors into a Graph SLAM system
- Chapter 3 presents the SLAM system called hdl_graph_slam, which is an already existing system used as the foundation for the rest of the work presented. We describe its main components and mechanisms, with particular focus on the parts which are later modified to introduce a new type of constraint
- Chapter 4 describes a SLAM system called hdl_graph_slam_with_map_priors. It introduces into hdl_graph_slam the buildings as entities in the pose graph and as a source of information for a new type of constraint. We describe the entire workflow of the system, from the download of buildings, to the alignment between a scan and the map of the buildings, to the insertion into the pose graph of nodes and edges
- Chapter 5 is divided into two sections, each one presenting a modification of hdl_graph_slam_with_map_priors, which are targeted to the correction of buildings that are badly positioned in OpenStreetMap. The first section, which describes a system called building_slam_with_buildings_priors Rigid SLAM, introduces a way of correcting the buildings which is based on the "global" alignment between the LiDAR scan and the map of the buildings, thus still subjected to major inaccuracies. The second section, which describes a system called building_slam_with_buildings_priors Non-rigid SLAM, introduces another way of

correcting buildings based on a "local" alignment between the LiDAR scan and every single building, thus more precise than the methodology introduced in the first section

- Chapter 6 presents the results of running the system with a set of pre-collected data. We describe the data and the tools used to evaluate the results. Numerical data about the correctness of the final pose of the robot with respect to the ground truth are presented and discussed. Lastly, a detailed visual evaluation shows how the system is able to correct the poses of the buildings
- Chapter 7 is an overview of the work presented in the thesis and it includes possible future improvements

2

STATE OF THE ART

Many real-world tasks associated with robot navigation require a map of the environment, such as transportation, search and rescue, automated vacuum cleaning, and many others. An accurate map allows for mobile robots to operate in complex and unknown environments only based on their onboard sensors and without relying on external reference systems such as GPS. Therefore, Simultaneous Localization and Mapping (SLAM) serves the purposes of building a detailed map of the environment in which the robot moves, and maintaining an accurate sense of the location of the robot.

In the first section of this chapter (Section 2.1), we describe the SLAM problem in detail, giving both its formulation and mathematical aspects. In Section 2.2, we explain accurately one of the SLAM representations, named Graph SLAM. Graph SLAM models the SLAM problem as a graph, where its nodes are the poses of the robot and its edges correspond to constraints between the poses. This representation highlights the spatial structure of the SLAM problem, for immediate and intuitive comprehension. This structure allows estimating maps with trajectories longer than the ones used in other popular approaches (e.g., EKF SLAM [35]). Thus, Graph SLAM is suitable to solve Full SLAM problems, in which the entire trajectory of the robot is estimated and the Online SLAM problem, in which only the last pose of the robot is estimated.

In Section 2.3, we describe some of the registration algorithms that are used to perform scan matching, but also to align perceptions of the robot (i.e., LiDAR scans) with data from maps, in order to create constraints to be inserted into the pose graph, as explained in Section 2.5. In the fourth section (Section 2.4) we describe OpenStreetMap, a publicly available map provider. A great variety of data can be retrieved from OpenStreetMap in different formats, using specific API. In robotics, this data is especially useful for autonomous driving and path planning. In particular, the robot may use data OpenStreetMap to better localize itself in the environment and build an accurate map of it. Lastly, in Section 2.5 we describe the use of map priors in the context of Graph SLAM. Data from OpenStreetMap, such as aerial images or data about roads and buildings, can be used to insert new nodes and edges into the pose graph, thus facilitating the solution of the SLAM problem.

Generally, the GPS signal is precise enough to allow for precise localization of a mobile robot. However, in many cases, the GPS signal becomes unreliable, for example in "corridor"-like environments or in mountainous zones, or

even becomes totally unavailable, as in tunnels and galleries. Additionally, GPS sensors are high energy-consuming and expensive. Thus, map priors are advantageous because they allow the robot to localize, with good accuracy, even without GPS.

2.1 SLAM

We describe the probabilistic structure underlying the SLAM problem and the typical workflow of a SLAM system.

2.1.1 *Probabilistic Model*

A sought skill for mobile robots is being able to build a map of the environment and to simultaneously localize within this map. This is known as the *Simultaneous Localization and Mapping* (SLAM) problem [1].

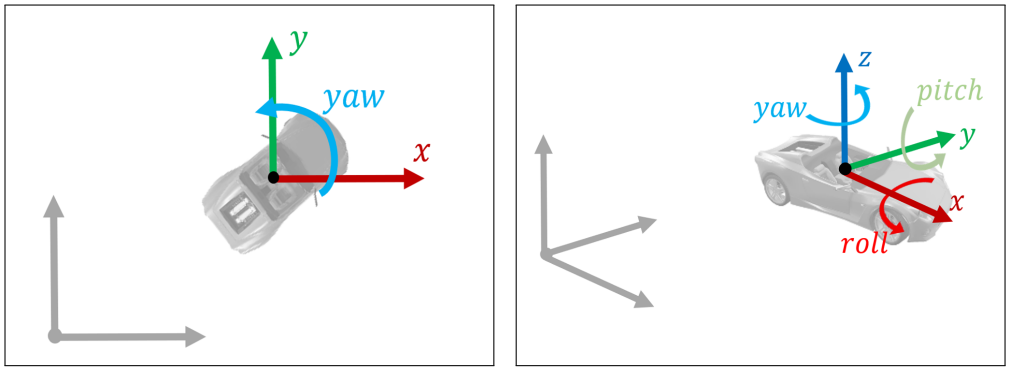
Solving the SLAM problem means estimating both the trajectory of the robot and the map of the environment in which the robot moves, using data coming from sensors equipped on the robot itself. Sensor measurements are uncertain due to their inherent noise, and for this reason, the SLAM problem is described by means of probabilistic tools.

The robot moves in an unknown environment, represented by the map $\mathbf{m}$, following a trajectory described by the sequence of random variables $\mathbf{x}_{1:T} = \{\mathbf{x}_1, \dots, \mathbf{x}_T\}$, representing the poses of the robot at regular intervals of time. While moving, the robot acquires a sequence of odometry measurements $\mathbf{u}_{1:T} = \{\mathbf{u}_1, \dots, \mathbf{u}_T\}$ and perceptions of the environment $\mathbf{z}_{1:T} = \{\mathbf{z}_1, \dots, \mathbf{z}_T\}$. Odometry is the measure of how much a robot has traveled from a certain reference. It can be provided by many sensors and techniques. An example is given by odometers, sensors that measure the revolution of the wheels of the robot [22]. Perceptions of the environment include a wide variety of quantities and sensors, such as LiDAR and radar scans, GPS coordinates, IMU measurements, and camera images.

The SLAM problem is characterized by the following models. The *motion model* $p(\mathbf{x}_t|\mathbf{x}_{1:t-1}, \mathbf{z}_{1:t}, \mathbf{u}_{1:t})$ represents the probability of the robot at time t to be in the pose $\mathbf{x}_t$, given the previous poses and given all the odometry measurements and perceptions. The *sensor model* $p(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{z}_{1:t-1}, \mathbf{u}_{1:t}, \mathbf{m})$ expresses the probability of performing the observation $\mathbf{z}_t$, given previous perceptions and given all the poses of the robot and odometry measurements.

Solving the SLAM problem, in probabilistic terms, means estimating the *posterior probability* of the trajectory of the robot $\mathbf{x}_{1:T}$ and the map $\mathbf{m}$ of the environment given all the measurements and an initial position $\mathbf{x}_0$:

$$p(\mathbf{x}_{1:T}, \mathbf{m}|\mathbf{z}_{1:T}, \mathbf{u}_{1:T}, \mathbf{x}_0) \tag{2.1}$$



(a) (b)

Figure 2.1: Degrees of Freedom of a mobile robot. (a) Degrees of freedom of a mobile robot on the 2D plane: translation along x (dark red), translation along y (dark green), rotation around z (light blue); (b) Degrees of freedom of a mobile robot in the 3D space: translation along x (dark red), translation along y (dark green), translation along z (dark blue), rotation around x (light red), rotation around y (light green), rotation around z (light blue).

The initial position $\mathbf{x}_0$ can be chosen arbitrarily. The poses $\mathbf{x}_{1:T}$ and the odometry $\mathbf{u}_{1:T}$ are represented as 2D transformations in $SE(2)$ or 3D transformations in $SE(3)$.

If the robot has n degrees of freedom, the motion of the robot is described by the group of $(n+1) \times (n+1)$ transformation matrices, that is called $SE(n)$, *Special Euclidean Group* of dimension n [22].

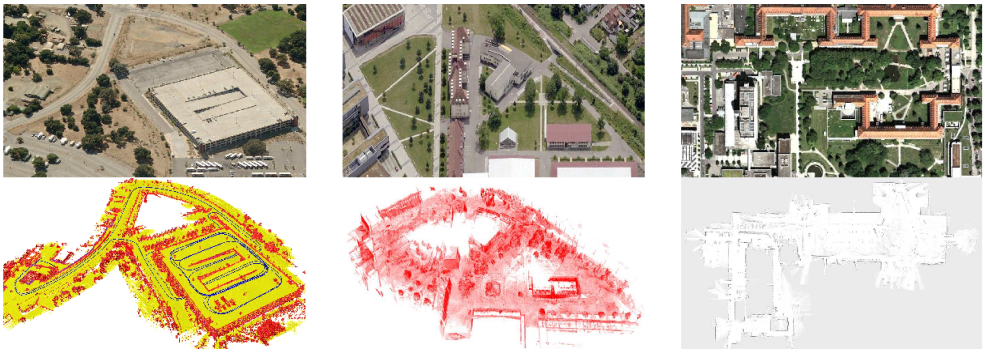
These transformations are in the form

$$\left\{ \left(\begin{pmatrix} R & v \\ 0 & 1 \end{pmatrix} \middle| R \in SO(n) \text{ and } v \in \mathbb{R}^n \right) \right\} \tag{2.2}$$

where R is the rotation matrix, which represents the rotation of a n dimensional body in $\mathbb{R}^n$ and v is the translation vector, corresponding to the translation of the body in $\mathbb{R}^n$. The rotation matrix and translation vector are independent one from the other. $SO(n)$, the *Special Orthogonal Group* of dimension n, represents the set of all nonsingular $n \times n$ real-valued matrices which columns are orthogonal and with determinant equal to 1, also called group of n-dimensional rotation matrices.

if $n = 2$, we are in the $SE(2)$ group and the robot translates and rotates in a 2D planar space. The robot has three degrees of freedom: two for translation and one for rotation, as represented in Figure 2.1a.

if $n = 3$, we are in the $SE(3)$ group and the robot translates and rotates in the 3D space. The robot has six degrees of freedom: three for translation and three for rotation, as represented in Figure 2.1b. Depending on the considered axis, rotations are named in different ways. In the field of autonomous driving,



(a) (b) (c)

Figure 2.2: Maps representations. (a) 3D map of the Stanford parking garage (bottom), and corresponding satellite view (top); (b) Point cloud map acquired at the University of Freiburg (bottom), and corresponding satellite view (top); (c) Occupancy grid map acquired at the hospital of Freiburg (bottom), and corresponding satellite view (top) representation; unobserved regions are gray, free space is white and black points are occupied space. Grisetti et al. [1]. Copyright © 2010, IEEE.

a rotation around the x axis is called *roll*, around the y axis is named *pitch* and the movement around the z axis is the *yaw*.

The map can be modeled in various formats. Common representations include sparse landmark-based maps, dense models, or raw sensor measurements-based maps, depending on the purpose. In Figure 2.2 there are examples of dense map representations: Figure 2.2a is a 3D multilevel surface map of a garage, Figure 2.2b is a 3D point cloud of a campus, and Figure 2.2c is an occupancy grid of a floor of a hospital.

To estimate the posterior given in Equation 2.1, the SLAM problem needs to have a well-defined structure, which arises from the static world and Markov assumptions. The static world assumption assumes that past and future data are independent given the current pose of the robot $\mathbf{x}_t$. The Markov assumption, based on the static world one, assumes that the motion model can be written as:

$$p(\mathbf{x}_t|\mathbf{x}_{1:t-1}, \mathbf{z}_{1:t}, \mathbf{u}_{1:t}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) \tag{2.3}$$

and that the sensor model can be written as:

$$p(\mathbf{z}_t|\mathbf{x}_{0:t}, \mathbf{z}_{1:t-1}, \mathbf{u}_{1:t}, \mathbf{m}) = p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{m}) \tag{2.4}$$

By removing the dependence from time history from these two models, we gain a simplified structure of the SLAM problem.

The structure can be represented in a convenient way through different tools and formulations. The graph-based, also known as network-based, formulation is one of them, which models the estimated trajectory of the

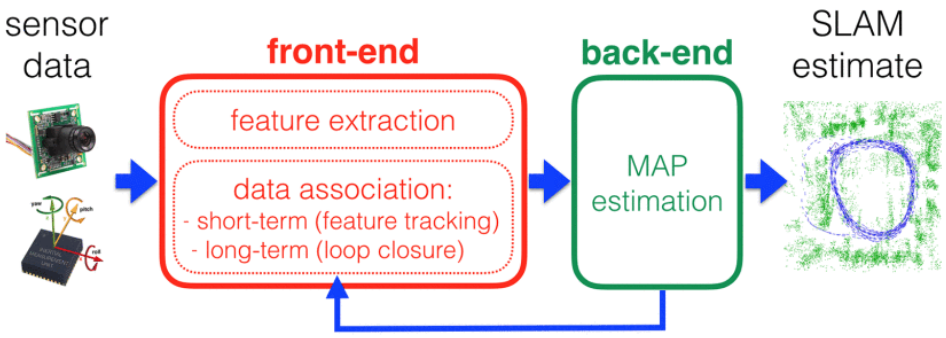


Figure 2.3: Typical structure of a SLAM system. Cadena et al. [49]. Copyright © 2016, IEEE.

robot and the relationships with sensor measurements as a graph. This representation is intuitive to adopt, since it highlights the spatial structure of the system, it makes it easy to add new types of perceptions and it allows to create large maps.

2.1.2 *SLAM Workflow*

A typical SLAM system, described in Figure 2.3, is made up of two components: the front-end and the back-end [49].

The *front-end* extracts relevant features form the sensor data into abstract models independent from the sensor data representation. It is also in charge of performing the *data association* between a measurement and a pose of the robot from which the measurement is most likely to have acquired from. It may also provide an initial guess for variables to be used in optimization. The data association module is divided into short-term and long-term blocks. Short-term data association is in charge of associating features in consecutive sensor measurements (e.g., scan matching), while long-term data association (also called loop closure) is in charge of relating new measurements with older poses.

The *back-end* utilizes the abstract data coming from the front end to estimate the trajectory of the robot and the map of the environment as a maximum a posteriori (MAP) problem (see the previous subsection for more information). The back-end sends back information to the front-end, to support loop closure.

2.2 GRAPH SLAM

The graph-based or network-based formulation of the SLAM problem, abbreviated as *Graph SLAM*, allows to easily formulate the SLAM problem by using a graph. The nodes of the graph correspond to the poses of the robot

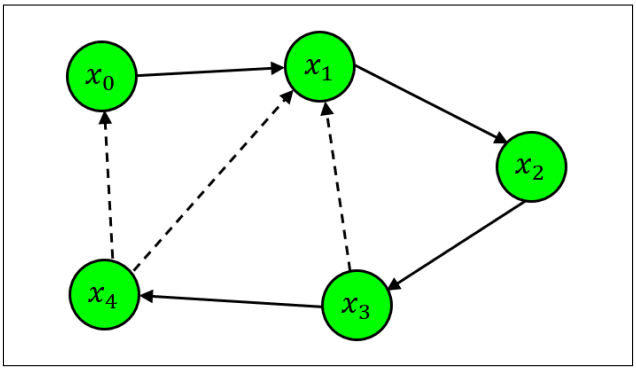


Figure 2.4: Pose graph. Pose graph structure, associated to the Graph SLAM problem formulation; nodes (light green) represent a robot pose x_i; edges between consecutive poses (straight arrows) model odometry measurements; other edges (dashed arrows) model other kinds of perceptions.

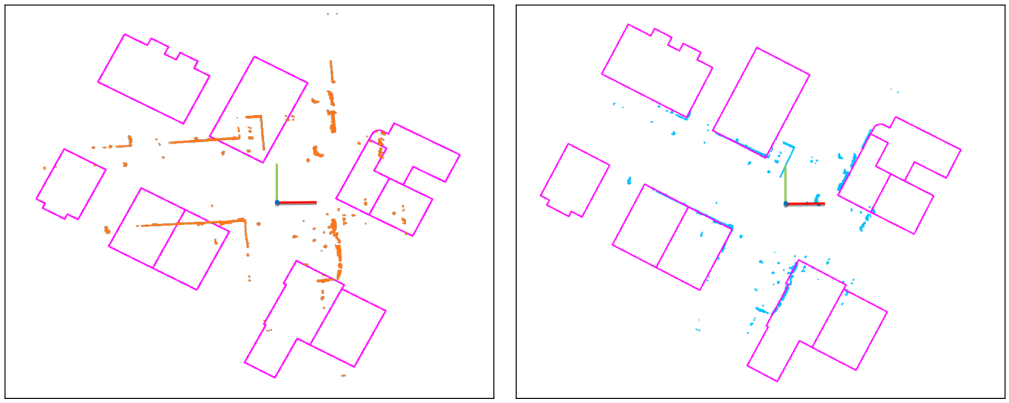
at different points in time, hence the name of *pose graph*, and the edges of the graph represent constraints between the poses [1]. Edges are obtained from observations of the environment or from movement actions carried out by the robot. The graph is optimized through least squares to find the poses of the nodes that are most consistent with the measurements modeled by the edges. The map is accurately reconstructed after optimization [1].

2.2.1 *Graph Structure*

A node in the graph represents a robot pose and a measurement acquired at that position. It can be identified in Figure 2.4 as the light green circle, representing the generic robot pose x_i. An edge between two nodes represents a spatial constraint between two robot poses, and it is represented in Figure 2.4 by the arrows. A constraint is a probability distribution over the relative transformation between two poses. A transformation can be either an odometry measurement between sequential robot positions (straight arrows in Figure 2.4) or an alignment of observations taken at two different robot locations, known as loop closure (dashed arrows in Figure 2.4). From Figure 2.4, we can see how edges may arise from the observation of parts of the environment already seen, represented by single nodes having associated multiple arrows, both incoming and outgoing.

In Graph SLAM, raw sensor measurements are replaced by edges in the graph, which can be thus seen as *virtual measurements*.

To each edge between two nodes is associated a probability distribution over the relative positions corresponding to the nodes, thus it is influenced by the measurements associated with these nodes.



(a) (b)

Figure 2.5: Data association between a LiDAR scan and the buildings map. (a) The LiDAR scan (orange) needs to be associated to the map, represented by the buildings (magenta); (b) Once the LiDAR scan is aligned (light blue) with the map, it can be associated to the buildings.

2.2.2 *Graph SLAM Workflow*

In Graph SLAM, the SLAM problem resolution is divided into two parts: graph construction and graph optimization, including all the steps of the typical SLAM workflow, described in Subsection 2.1.2.

Graph construction consists of the construction of the graph, as explained in Subsection 2.2.1. It is part of the SLAM *front-end*, and it depends on sensors. One of the main tasks of the front-end is to perform *data association*. Given the nature of the SLAM problem, a single observation z_t might result in multiple edges connecting different nodes. To reduce the complexity of the problem, we need to estimate the most likely constraint resulting from an observation, which is the data association. An example can be seen in Figure 2.5. In Figure 2.5a we have a LiDAR scan taken from a certain pose of the robot (orange point cloud), represented by a node in the pose graph, that has to be associated with the buildings seen by that pose, each building also represented by a node in the pose graph. To do it, we need to perform data association between the LiDAR scan and the map, represented by the buildings themselves (magenta outlines). In Figure 2.5b we can see the LiDAR scan now correctly aligned with the map (light blue point cloud), thus the node of the robot pose can be correctly linked with the nodes of the buildings through the insertion of new edges in the graph.

Graph optimization consists of determining the most likely configuration of the poses given the edges of the graph. It is part of the SLAM *back-end* and it uses an abstract representation of the data independent of sensors, namely, the pose graph.

2.2.3 *Optimization*

Without loss of generality, under the assumption that the observations are affected by Gaussian noise and the data association is known, the goal of a Graph SLAM algorithm is to compute a Gaussian approximation of the posterior (Equation 2.1) over the trajectory of the robot.

To achieve so, we need to compute the mean of this Gaussian as the configuration of the nodes which maximizes the likelihood of the observations. Once this is known, we can compute the information matrix of the Gaussian. The information matrix accounts for the uncertainty in the representation of the Gaussian; if its elements have a high value, this means that the uncertainty is low, and vice versa. Let $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)^T$ be a vector of parameters, with $\mathbf{x}_i$ the pose of the node i. Let $\mathbf{z}_{ij}$ and $\mathbf{\Omega}_{ij}$ be the mean and information matrix of a virtual measurement between the node i and the node j. This virtual measurement is a transformation that makes the observations acquired from i maximally overlap with the observation acquired from j.

Let $\hat{\mathbf{z}}_{ij}(\mathbf{x}_i, \mathbf{x}_j)$ be the prediction of a virtual measurement given the configuration of the nodes $\mathbf{x}_i$ and $\mathbf{x}_j$, which is the relative transformation between the two.

The error function that computes the difference between the expected measurement $\hat{\mathbf{z}}_{ij}$ and the real measurement $\mathbf{z}_{ij}$ is then computed:

$$\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{z}_{ij} - \hat{\mathbf{z}}_{ij}(\mathbf{x}_i, \mathbf{x}_j) \tag{2.5}$$

The log-likelihood l_{ij} of a measurement $\mathbf{z}_{ij}$ is

$$l_{ij} \propto [\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j)]^T \mathbf{\Omega}_{ij} [\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j)] \tag{2.6}$$

This is used to compute the negative log-likelihood $F(\mathbf{x})$ of all observations

$$F(\mathbf{x}) = \sum_{<i,j> \in \mathcal{C}} \mathbf{e}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij} \tag{2.7}$$

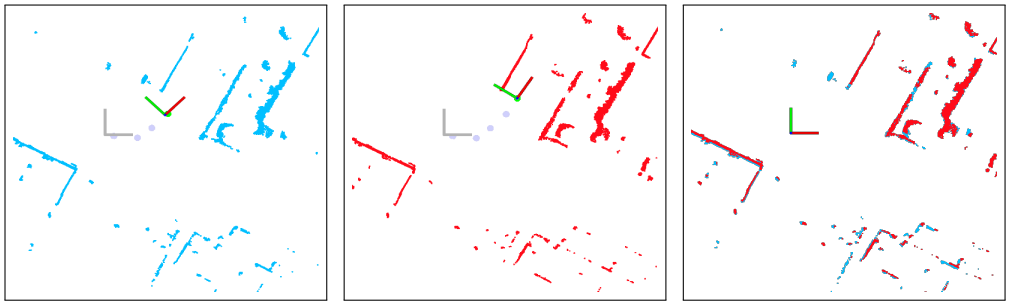
The best configuration $\mathbf{x}^*$ of the nodes is the one which minimizes $F(\mathbf{x})$, which means to solve the following equation:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} F(\mathbf{x}) \tag{2.8}$$

Equation 2.8 can be solved through standard non-linear least-squares optimization methods, like Gauss-Newton or Levenberg-Marquardt.

The information matrix H of the system is obtained by projecting the measurement error in the space of the trajectories via the Jacobians.

Following this procedure, both the mean and the information matrix of the Gaussian approximation of the robot pose posterior through the use of a graph can be computed.



(a) (b) (c)

Figure 2.6: Registration of two point clouds. (a) point cloud acquired in pose $t = 1$ (blue) in local reference frame (colored reference frame); (b) point cloud acquired at $t = 2$ (red) in local reference frame (colored reference frame); (c) aligned point clouds in global reference frame.

2.2.4 *G2O*

g2o is an open-source C++ framework for building pose graphs and optimizing graph-based nonlinear error functions [15].

g2o achieves a performance that is comparable with implementations of state-of-the-art algorithms, such as $\sqrt{\text{SAM}}$ [38], SPA [39], sSBA [40], and RobotVision [41], and is able to accept general forms of nonlinear measurements.

We use this framework to perform the construction and optimization of the graph in the systems proposed in the thesis, since it is easy to introduce new types of constraints, due to the abstraction of data, and it allows us to use many base classes for nodes and edges, which can be easily combined, redefined, and extended.

2.3 POINT CLOUD REGISTRATION ALGORITHMS

3D *registration*, also known as scan matching, is the problem of finding the transformation that better aligns two point clouds [19]. When point clouds are acquired by 3D sensors from different viewpoints, registration finds the relative pose between views in a global coordinate frame, such that the overlapping areas between the point clouds match as well as possible. An example of 2D registration between point clouds at different viewpoints can be seen in Figure 2.6. In Figure 2.6a and Figure 2.6b there are two point clouds (blue and red), acquired at two consecutive poses of the robot (green dot). Each point cloud is in the local reference frame (colored reference frame) of the pose of the robot. Gray dots represent previous poses of the robot, and the gray reference frame is the global reference frame (the same for both figures).

In Figure 2.6c, we can see that the two point clouds have been aligned in the global reference frame (now the colored one).

PCL is an open-source library that contains state-of-the-art algorithms for filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation [20]. We use PCL to perform scan matching between point clouds, exploiting the multiple algorithms provided by the library.

2.3.1 *ICP*

The ICP algorithm is a method that performs registration between two point clouds [19]. It works by finding, first, correspondences between points in the two clouds: two closest points in the Cartesian space are considered to correspond one to the other. Then, the algorithm estimates a transformation that minimizes the Euclidean distance between the pairs of points of each correspondence, using the least squares method. The process of determining correspondences and computing the transformation is iteratively repeated until multiple termination criteria are satisfied. The source point cloud converges towards the target point cloud as the correspondences increasingly become better and better.

2.3.2 *GICP*

The GICP algorithm is based on ICP and performs a point-to-point alignment using local surface information in the form of covariance matrices [19].

2.3.3 *NDT*

Normal - distributions transform, or NDT, is a general 3D surface representation with applications in scan registration, localization, loop detection, and surface-structure analysis [21].

LiDARs usually produce data in the form of point clouds. After applying NDT to the original point cloud, the scanned surface is instead represented by a piecewise smooth function with analytic first and second-order derivatives.

The smooth function representation allows using standard methods from the numerical optimization literature, such as Newton's method, for scan registration.

2.3.4 *NDT-OMP*

The NDT-OMP package [16] contains OpenMP-boosted NDT (NDT-OMP) and GICP (GICP-OMP) algorithms derived respectively from NDT and GICP

algorithms in the PCL library. These algorithms are SSE-friendly and multi-threaded.

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C++ and other programming languages on many platforms [42]. SSE (Streaming SIMD Extensions) is an instruction set for microprocessors used for parallel processing [43].

2.3.5 *Fast GICP*

The Fast GICP package [17] is a collection of fast point cloud registration algorithms based on GICP. It contains a multi-threaded GICP algorithm (FastGICP), multi-threaded and GPU implementations of the voxelized GICP (FastVGICP) algorithm [18]. All these algorithms expose the PCL registration interface, so they can be easily integrated with it.

2.4 OPENSTREETMAP

One of the possible representations associated with SLAM maps can be derived from OpenStreetMap. Additionally, we can derive information and measurements from OpenStreetMap to be inserted in the graph.

OpenStreetMap (OSM) is a collaborative project to create a free editable map of the world [8]. An example of map derived from OSM is represented in Figure 2.7, which shows the district of Città Studi in Milan. The main outcome of this project is the geodata underlying the map, which can be used in the production of paper and electronic maps, geocoding of address and place names, route planning, and many other services. These are particularly useful in the field of robotics, to better localize the robot, build precise maps and perform accurate path planning.

2.4.1 *OSM Data Structure*

OSM implements a conceptual data model of the physical world based on components called *elements* [5]. There are three types of elements: nodes, ways, and relations. All the elements can have one or more associated tags.

A *node* represents a point on the surface of the Earth. It comprises at least an ID number and a pair of coordinates corresponding to the latitude and longitude of the point. Nodes have multiple purposes: some define standalone point features in the world, such as a bench or entrances of buildings. Others can be used to either define the shape of a way or be members of a relation.

A *way* is an ordered list of nodes that define a polyline. It can be used to represent linear features, such as roads or rivers, or the boundaries of areas

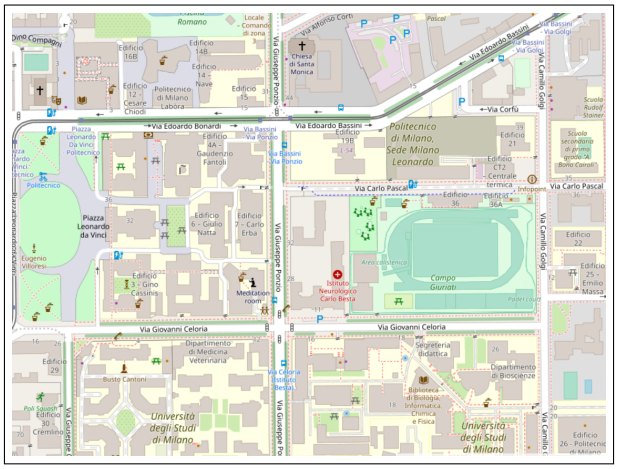


Figure 2.7: Map from OpenStreetMap of Città Studi, Milan

and polygons (closed way), like buildings or forests. The latter is called closed way and has the first and last nodes coinciding.

A *relation* is a multi-purpose data structure that tells the relationship between two or more data elements. It is an ordered list of nodes, ways, and other relations. The meaning of the relation is defined by its tags. For example, a relation could represent a multipolygon, which is an area with holes or a route made up by many subways (e.g., a highway).

A *tag* describes the meaning of the particular element to which it is attached. It is made up of two fields: a key and a value, both represented as strings of characters. The key describes the meaning of the tag, such as "highway", and it is unique. The value is the description of the key, such as "residential". There is no fixed dictionary of tags, but standard conventions are adopted and documented on the OSM website [6].

Each element has a set of common *attributes*, such as the id of the element, the user who last modified it, and the timestamp of the last modification among others. For each element, tags and full editing history are also stored.

2.4.2 *Accessing OpenStreetMap Data*

It is possible to access and download map data from the OSM dataset in many ways [12]. First, OSM data can be accessed through a single file called *Planet.osm* [2]. This file contains all the nodes, ways, and relations that make up the entire world map. Being this a heavy file, smaller subsections of the map are usually accessed. Alternatively, OSM has an *editing API* (also called *main API*) for fetching and saving raw geodata from/to the OSM database [11]. Other dedicated tools and services can be used to access OSM data, such as Overpass API.

2.4.3 *Overpass API*

The *Overpass API* is a read-only API that allows accessing parts of the OSM map data selected in a custom way [7]. It allows the client to send a query through an HTTP GET request to the API server, which will, in turn, send back the dataset that resulted from the query. There exist two languages to write a query: Overpass XML or Overpass QL. The *Overpass QL* syntax is more concise than Overpass XML and is similar to C-like programming languages. The *Overpass XML* syntax is safeguarded, because it uses more explicit named parameters than QL. We chose the QL language because of its ease of use.

The response can be in different formats [14], such as OSM XML, OSM JSON, custom templates, and pretty HTML output. For the development of this thesis, we choose to get data in OSM XML format, since the C++ Boost library, already required for other packages to work, has an integrated XML parser.

2.4.4 *Overpass QL*

The basic mechanism underlying Overpass API is that streams of OSM data are generated and modified by statements, which are executed one after another [4]. A statement is a block of code that performs a generic action. OSM elements are then selected through search criteria, e.g., location, type of objects, tag properties, proximity, or combination of them specified through a query. Overpass QL source code is divided into *statements*, and every statement ends with a semicolon ; [3].

Overpass QL manipulates *sets*, so that statements write their results into sets, and sets are then read by subsequent statements as input. A named set can be specified as the input, or as the output, of a statement. If a set name is not specified, the default set, named _ (underscore), is used to read from and write to. Once a new result is assigned to an existing set, its previous content will be replaced and will be no longer available.

There are different types of statements, grouped into settings, block statements, and standalone queries: *settings* are optional global variables set in the first statement of the query; *block statements* group statements together (e.g., intersections and loops); *standalone queries* are complete statements on their own.

A major category of standalone queries are *query statements*, made up of the type of the element to be retrieved (node, way, relation, or a combination of them) and at least one *filter* that describes the object to be retrieved.

There are different types of filters, which can be used to filter objects in the input result set or to perform recursion. Examples of filters on the input set are the bounding box or the area filters, used to select all the elements

within a certain rectangular box or specified area, or tag filters, used to filter elements based on the key and/or value of their tags.

A generic *recursion query* replaces all objects in the input set with the union of all their recursively related objects [4]. Recursion queries can be filters, or they can exist as standalone queries. In the former case, they act on the specified elements to be retrieved; in the latter, they act on the input set (specified or default).

Another type of standalone query is represented by the *out statement*, which outputs the contents of a set. The set can be either specified or the default input set _ will be used.

2.4.5 *OpenStreetMap XML*

Data downloaded from OSM comes in the form of XML formatted .osm files [12]. *OSM XML* format follows an XML schema definition that was first used by the main API [13]. It consists mainly of a list of instances of OSM data primitives (nodes, ways, and relations).

The tags contained in the OSM XML are:

- *XML suffix*: specifies UTF-8 character encoding

```
<?xml version="1.0" encoding="UTF-8"?>
```

- *OSM element*: contains the version of the API and the generator that produced the file

```
<osm version="0.6" generator="CGImap 0.0.2">
        ...
</osm>
```

Inside the OSM element there are:

- *Bounds*: represents the bounding box from which data are retrieved

```
<bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900"
     maxlon="12.2524800"/>
```

- *Node*: represents a node element and contains its tags (if present)

```
<node id="1831881213" version="1" changeset="12370172" lat
    ="54.0900666" lon="12.2539381" user="lafkor" uid="75625"
    visible="true" timestamp="2012-07-20T09:43:19Z">
        <tag k="name" v="Neu Broderstorf"/>
        <tag k="traffic_sign" v="city_limit"/>
</node>
```

- *Way*: represents a way element and contains the references (*nd*) to the composing nodes and the tags of the way. *nd* are a reference to the nodes that constitute the way, each having the attribute *ref*, which contains the *id* of the corresponding node

```
<way id="26659127" user="Masch" uid="55988" visible="true" version
    ="5" changeset="4142606" timestamp="2010-03-16T11:47:08Z">
        <nd ref="292403538"/>
        <nd ref="298884289"/>
        ...
        <nd ref="261728686"/>
        <tag k="highway" v="unclassified"/>
        <tag k="name" v="Pastower Strasse"/>
 </way>
```

- *Relation*: represents a relation element and contains the references to the composing elements and the tags of the relation (if present). The attribute *ref* of *member* contains the *id* of the element specified by *type*

```
<relation id="56688" user="kmvar" uid="56190" visible="true"
    version="28" changeset="6947637" timestamp="2011-01-12T14:23:49
    Z">
        <member type="node" ref="294942404" role=""/>
        ...
        <member type="node" ref="364933006" role=""/>
        <member type="way" ref="4579143" role=""/>
        ...
        <member type="node" ref="249673494" role=""/>
        <tag k="name" v="K{\"u}stenbus Linie 123"/>
        <tag k="network" v="VVW"/>
        <tag k="operator" v="Regionalverkehr K{\"u}ste"/>
        <tag k="ref" v="123"/>
        <tag k="route" v="bus"/>
        <tag k="type" v="route"/>
</relation>
```

These elements are not always all present.

2.4.6 *Overpass API XML*

The XML returned as a response from Overpass API generally follows the conventional OSM XML [14] described before. Elements (ways, nodes, and relations) are defined in the same manner. However, differently from OSM XML, there is no *bounds* tag and there may be other additional tags.

A response has always the following header and footer:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<osm version="0.6" generator="Overpass API">
<note>The data included in this document is from www.openstreetmap.org.
    It has there been collected by a large group of contributors. For
    individual attribution of each item please refer to https://www.
    openstreetmap.org/api/0.6/[node|way|relation]/#id/history </note>
<meta osm_base="date"/>
...
</osm>
```

Additional tags are:

- *note* tag: contains a copyright reminder
- *meta* tag: contains, in the *osm_base* attribute, a date; it means that all edits that have been uploaded before this date are included

Here is a complete example:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="Overpass API 0.7.56.9 76e5016d">
<note>The data included in this document is from www.openstreetmap.org.
    The data is made available under ODbL.</note>
<meta osm_base="2021-06-03T11:55:28Z"/>
  <node id="1205619743" lat="48.9850817" lon="8.3935543"/>
  <node id="1205619817" lat="48.9852221" lon="8.3935237"/>
  <node id="1205619851" lat="48.9851845" lon="8.3936350"/>
  <node id="1205619875" lat="48.9851194" lon="8.3934430"/>
  <way id="104492674">
    <nd ref="1205619743"/>
    <nd ref="1205619875"/>
    <nd ref="1205619817"/>
    <nd ref="1205619851"/>
    <nd ref="1205619743"/>
    <tag k="addr:city" v="Karlsruhe"/>
    <tag k="addr:housenumber" v="14"/>
    <tag k="addr:postcode" v="76199"/>
    <tag k="addr:street" v="Mainstrasse"/>
    <tag k="building" v="yes"/>
    <tag k="source" v="LA-KA"/>
  </way>
</osm>
```

From lines 1 to 5 we can see the header as specified by Overpass API. Aside from tags of the header, other elements inside *osm* are nodes and ways. First, there is a list of nodes. Each node specifies a point on the Earth using latitude and longitude. After that, there is a way. Inside it, there are the *nd*, whose attribute *ref* coincides each one with the id of a node specified above. The attribute *ref* of the first *nd* and the last coincide, meaning that this is a closed way. This particular way represents a building, because of the *tag* with

key equal to "building" and corresponding value equal to "yes". Other *tags* describe features and information about the building (e.g., the address).

2.5 MAP PRIORS

For a mobile robot to operate reliably, a consistent map of the environment is necessary. Because errors in the pose of the robot accumulate over time, building large maps from odometry and LiDAR often leads to a drift in the trajectory estimate. GPS information can be used to compensate for that drift. However, the GPS signal is not always present or may be disturbed. This may result in poor positioning performance. Performing loop closures reduces the drift, but it forces the robot to visit places that it has already seen.

Possible methods to solve this problem consist of coupling the information from publicly available maps, like aerial photographs or OpenStreetMap data with standard localization or SLAM approaches [9]. By using this information, when the GPS is not present or not precise, the robot is able to obtain a better accuracy on the pose of the robot with respect to the case without map priors and is able to correctly detect loop closures.

Hentschel and Wagner [29] use data about buildings from OSM to initialize the particles of the Monte Carlo localization. Other tasks that use OSM information are path planning and light control of the robot. Floros, Zander, and Leibe [30] align the trajectory of the robot with information about roads extracted from OSM through chamfer matching and use it to weight the particles of the Monte Carlo localization. Kümmerle et al. [28] relate LiDAR measurements with aerial images, use the resulting information to initialize Monte Carlo localization and then optimize using Graph SLAM. Vysotska and Stachniss [9] associate LiDAR measurements with data associated to buildings retrieved from OpenStreetMap, and use the resulting information to add constraints into the graph.

Kümmerle et al. [28] and Vysotska and Stachniss [9] approaches are advantageous with respect to the others because they use LiDAR measurements, which are precise and usually already available in the system. In the following subsection, we describe the work from the latter, since it is easier to directly process data about buildings than aerial images. Additionally, OSM does not provide directly aerial images but has to rely on external sources, while it provides an easily accessible API to download data about buildings.

2.5.1 *OpenStreetMap Priors*

Vysotska and Stachniss [9] relate LiDAR measurements with the data associated to buildings, coming from OSM, and insert it into the pose graph.

(a) (b)

Figure 2.8: Buildings from OpenStreetMap. (a) Screenshot from OpenStreetMap; (b) outlines of buildings as downloaded from OpenStreetMap.

To include this additional knowledge into the graph optimization process, they extend the error function 2.5 as:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \mathbf{F}(\mathbf{x}) + \mathbf{F}^{map}(\mathbf{x}) \tag{2.9}$$

where $\mathbf{F}^{map}(\mathbf{x})$ is the error introduced by the mismatch between the observations made by the robot and the map information. In the graph, this represents a constraint between a robot pose $\mathbf{x}_i$ and OSM information. It is defined as:

$$\mathbf{F}^{map}(\mathbf{x}) = \sum_i \mathbf{e}_i^{map}(\mathbf{x})^\top \mathbf{\Omega}_i \mathbf{e}_i^{map}(\mathbf{x}) \tag{2.10}$$

where $\mathbf{e}_i^{map}(\mathbf{x})$ is the error function, and $\mathbf{\Omega}_i$ is the corresponding information matrix.

To define this new constraint, the correct data association is needed between the map and the LiDAR measurements made by the robot. Measurements from the LiDAR are in the form of 2D or 3D scans. From these scans, line extraction is performed to remove non-buildings objects. The final result is a set of potentially disconnected lines. From OSM data, they build a map of lines representing the walls of buildings in the environment. Figure 2.8 shows an example of such map of lines representing buildings, with the corresponding OSM map.

A registration algorithm, ICP, is then used to match the current LiDAR scan to the map of buildings. The transformation estimated by ICP, that is the misalignment between the current pose of the robot and the map, is used to build the error function $\mathbf{e}_i^{map}(\mathbf{x})$.

The error function is the difference between the current pose of the robot $\mathbf{x}_i$ and the pose $\hat{\mathbf{x}}_i$ computed by aligning the scan with the building. Assuming the pose is a $SE(2)$ element, they divide the pose $\mathbf{x}_i$ into translation

component t_i and rotational component θ_i (with its corresponding rotation matrix R_i). The error function is defined as:

$$\boldsymbol{e}_i^{map}(\boldsymbol{x}_i) = \begin{pmatrix} \hat{R}_i^T(\mathbf{t}_i - \hat{\mathbf{t}}_i) \\ \theta_i - \hat{\theta}_i \end{pmatrix} \tag{2.11}$$

The information matrix $\boldsymbol{\Omega}_i$ of the constraint is computed as $\boldsymbol{\Omega}_i = (\Sigma_i^{ICP})^{-1}$. Σ_i^{ICP} is the covariance matrix of the ICP alignment and is computed as

$$\Sigma_i^{ICP} = 2\sigma^2 H_{ICP}^{-1} \tag{2.12}$$

where H_{ICP} id the Hessian matrix of the ICP error function and σ^2 is the variance factor.

Equation 2.9 is then solved with Levenberg-Marquardt optimization using the g2o framework.

Experiments have been carried out through the use of a mobile robot equipped with a Velodyne LiDAR [9]. The only measurements available are odometry and LiDAR scans.

The first experiment shows that by using map priors, the error in the final pose of the robot is about 1 m, while without them is about 5 m. Additionally, with map priors, the internal map of the robot is better aligned with the structure of the environment and the loop closure was correctly detected, while it was not detected without them. A second experiment, executed in a more complex environment than the first, shows that by using map priors the robot is able to correct inaccuracies in its map and to detect correctly loop closures even when the environment is complex and highly cluttered. The error on the final pose passes from 22 m without map priors to 0.5 m with them. Other experiments have been carried out to show that the system is able to work correctly even in presence of map inconsistencies, such as demolished buildings that still appears on OSM or new buildings that are still not registered on OSM. The execution time is just slightly increased.

A disadvantage is that a precise alignment between buildings and LiDAR scans is required in order to get good accuracy on the poses of the robot. The alignment becomes difficult to be performed when there is a lot of clutter in the environment or in zones when there are few buildings. Other problems arise when the buildings are not correctly positioned in OSM and thus the alignment may not be able to work properly. Additionally, the map is not representative of the real environment, and thus the precision on the localization of the robot decreases. The wrong positioning of buildings happens especially in OpenStreetMap since it is a collaborative project carried out by different people with different tools.

A possible improvement, which constitutes one the multiple contributions of this thesis and that will be explained in Chapters 4 and 5, is the possibility to also improve the map by correcting the positions of buildings through

the alignment between the map of buildings and LiDAR scans. Differently from what explained in this subsection, in which the alignment is just used as a measure attached to the pose of the robot, buildings become real entities in the graph at the same level of the robot poses, thus each building can separately influence the pose of the robot and its position can be potentially corrected with respect to errors in OpenStreetMap. Crucial is the fact that, since buildings are now nodes, a building can be potentially observed by many robot poses, thus we can enrich the pose graph with many new informative edges.

3

HDL GRAPH SLAM

hdl_graph_slam is an open source ROS package for real-time 6 Degrees of Freedom (DoF) SLAM using 3D LiDAR SLAM. It performs 3D Graph SLAM with scan matching-based odometry estimation and loop detection [10]. It also supports several graph constraints, such as GPS, IMU acceleration, IMU orientation, and floor planes (detected in a point cloud). It uses the g2o framework to build the pose graph and optimize it.

This package has been tested with Velodyne (HDL-32E, VLP16) and RoboSense (16 channels) sensors in indoor and outdoor environments, but can be used with other sensors such as Velodyne HDL-64E and Ouster OS1.

3.1 ROS

The *Robot Operating System* (ROS) is an open-source, meta-operating system for robots [31]. It provides the services of an operating system, such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for writing, managing, and running code across multiple computers.

At runtime level, ROS is structured as a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several styles of communication, including synchronous communication over services, asynchronous streaming of data over topics, and storage of data on a parameter server.

3.1.1 *Basic Concepts*

The basic concepts at the base of the peer-to-peer network of ROS processes are nodes, the master, the parameter server, messages, services, topics, and bags [32].

Nodes are processes that perform computation. They are meant to allow different machines to communicate over the same ROS network, by running a single node on each of them (though more nodes can still be used on the same device). A robot system is usually made up of many nodes. For example, one node controls a LiDAR, one node performs localization, one node deals with

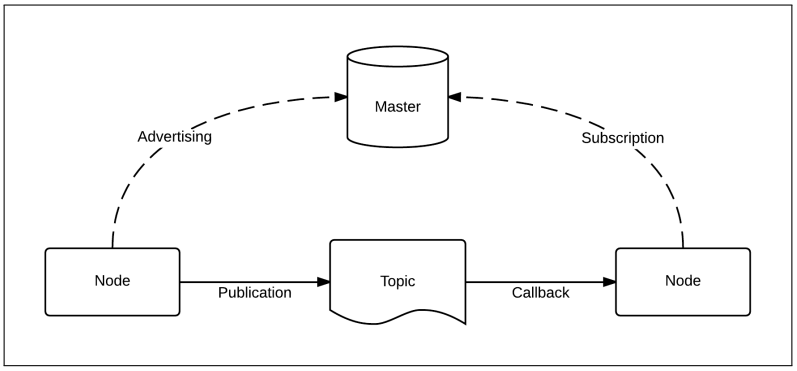


Figure 3.1: ROS components

path planning, and so on. A ROS node is written using a ROS client library, such as roscpp or rospy.

Nodelets are alternative to nodes, but provides the same functionalities. They are designed to run multiple algorithms in a single machine. Because of the shared memory, there is no cost of copying messages between processes, and thus communication between nodelets is easier and more efficient with respect to the communication between nodes [33]. The *master* provides name registration and lookup to the rest of the network. It makes the nodes able to find each other, exchange messages, and invoke services. *Names* in the ROS network constitutes a hierarchical naming structure used by all the resources (i.e., nodes, parameters, topics, and services) [44]. Each resource has a name and it is defined within a namespace, that may be shared with others. Each resource can create resources within its namespace and it can access resources within, or above, its namespace. Names are resolved in a relative way so that a resource does not need to know in which namespace it is. For example, the node named "/ns/node1" has namespace "/ns", so if it requests the resource named "node2", the name will be resolved to "/ns/node2".

The *parameter server* allows data to be stored in a central location, which is the master, to be read and written by the nodes in the network.

Messages are data structures exchanged between nodes, allowing them to communicate with each other. They may contain standard primitive types (int, float, bool, ...), nested structures, arrays, but also more complex and customized entities (e.g., point clouds or images). Messages are exchanged into the ROS network through a publish/subscribe mechanism. A node, named *publisher*, sends a message over the network by publishing it into a *topic*. A node interested in reading a certain type of data will subscribe to the appropriate topic, hence the name of *subscriber*. A topic is identified by a unique name and by the type of message published on it. There may be multiple publishers and/or subscribers for a single topic and a node may publish and/or subscribe to different topics.

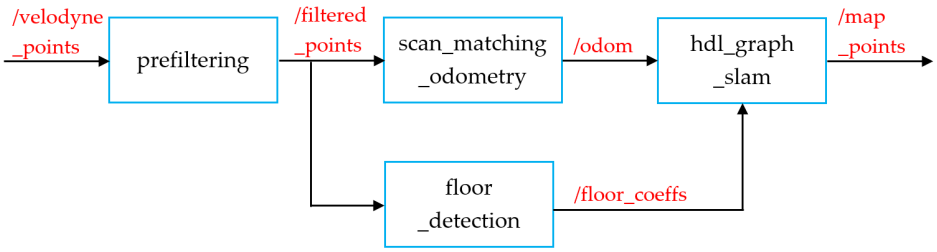


Figure 3.2: hdl_graph_slam system architecture

A *callback function* is the user-defined function automatically executed when a node receives data from the topics to which is subscribed. There is a callback function for each subscribed topic.

As stated above, the publish/subscribe model, illustrated in Figure 3.1, allows nodes to communicate with each other in a fast and efficient way. However, this model is not appropriated for request/reply interactions, which are frequent in a distributed system. *Services* implement the request/reply model through the use of a pair of messages: one for the request and one for the reply. A node offers a service under a name and a client uses the service by sending the request message and waiting for the reply.

Bags are a format for saving and playing ROS message data. They are used for storing data, such as sensor data, which can be difficult to collect, but necessary for developing and testing algorithms.

3.2 SYSTEM ARCHITECTURE

hdl_graph_slam consists of four ROS nodelets, each having its specific purpose, typical of a SLAM system:

- prefiltering_nodelet
- scan_matching_odometry_nodelet
- floor_detection_nodelet
- hdl_graph_slam_nodelet

Figure 3.2 shows how the nodelets (blue rectangles) interact between themselves by reading/publishing topics (red). The prefiltering_nodelet reads a point cloud from the topic */velodyne_points*, filters it and publishes the resulting cloud onto */filtered_points*. This topic is read by scan_matching_odometry_nodelet, which performs scan matching and publishes an odometry message over the topic */odom*. */filtered_points* is also read by floor_detection_nodelet, which detects a floor plane into the point cloud and

publishes the corresponding coefficients on the */floor_coeffs* topic. Odometry and floor coefficients are read by hdl_graph_slam_nodelet, which inserts all the constraints into the pose graph and performs optimization. The resulting map of the robot is published onto the topic called */map_points*.

3.2.1 *prefiltering_nodelet*

prefiltering_nodelet reads a point cloud, filters it, and then publishes it over the network so that it can be used by other nodelets.

It subscribes to the topic over which point clouds are published, called */velodyne_points*. When a point cloud is received, it is optionally deskewed (i.e., compensated for the ego-motion of the robot) using data from the IMU and transformed from the cloud reference frame to the *base_link* frame of the robot. The point cloud is then distance filtered, so that points that are too close or too distant are removed. After that, the cloud is downsampled, to reduce uniformly the number of points. Lastly, it is filtered to remove outliers using a statistical or radius filter [45] and published over the ROS network under the topic */filtered_points*.

3.2.2 *scan_matching_odometry_nodelet*

Odometry is the measure of how much a robot has traveled from a certain reference. Odometry can be modeled and provided in many ways. It is usually given by odometry sensors, which measure the revolution of the wheels of a robot [22]. Such measurements are inaccurate because of wheel slippage, surface imperfections, modeling errors, and others. hdl_graph_slam estimates odometry through scan matching [23], as it operates on point clouds coming from LiDARs.

scan_matching_odometry_nodelet subscribes to the topic */filtered_points* from prefiltering_nodelet and estimates the pose of the robot by iteratively applying a scan matching between consecutive point clouds. The estimated odometry is in the form of a $SE(3)$ transformation, which represents the linear and angular distance traveled from the previously estimated odometry. It is published over the ROS network under the topic */odom* as a *nav_msgs::Odometry* message, one of the multiple messages already provided by the ROS framework, so that it can be used by hdl_graph_slam_nodelet.

3.2.3 *floor_detection_nodelet*

floor_detection_nodelet detects floor planes by RANSAC [46]. The detected floor planes are published so that they can be used by hdl_graph_slam_nodelet.

floor_detection_nodelet reads a point cloud from the topic */filtered_points* published from prefiltering_nodelet. It performs a height filtering on the point cloud and optionally normal filtering, and then tries to detect a 3D plane in the point cloud using RANSAC.

If a plane is detected, the coefficients of the plane are published as a *hdl_graph_slam::FloorCoeffs* message which contains an array of four coefficients of a 3D plane, on the topic */floor_coeffs*. The point cloud containing the inliers of the plane is also published under the topics */floor_points* and */floor_filtered_points*.

3.2.4 *hdl_graph_slam_nodelet*

hdl_graph_slam_nodelet adds different types of constraints into a pose graph, based on data received from the external environment and from other nodelets. It performs loop detection and optimizes the pose graph, to compensate for the accumulated error of the scan matching.

3.3 HDL_GRAPH_SLAM_NODELET

We describe hdl_graph_slam_nodelet in detail since it is the nodelet that will be modified to introduce the building constraints, explained in Chapter 4.

hdl_graph_slam_nodelet subscribes to various ROS topics associated with the data from which constraints are built, i.e., odometry, floor coefficients, GPS, and IMU, all with the respective timestamps. The corresponding callback functions process the data received and store them in FIFO queues.

An optimization function is periodically called through the use of a timer. This function calls the so-called *flush queue functions*. Flush queue functions, one for each type of data received, are in charge of parsing the queues; by processing data inside them and inserting the results into the corresponding keyframe, which is the data structure containing all the data associated with a robot pose, and into the pose graph. The most important flush function is the keyframe flush function, which creates the keyframe and inserts the corresponding pose of the robot and the edge between the previous pose and the current one in the graph. GPS and IMU flush functions insert GPS priors and IMU orientation and acceleration priors in the graph. The floor flush function inserts the $z = 0$ floor node and the edges with keyframe nodes.

If the flush queue functions create new keyframes or insert new data into an already existing keyframe, then the optimization and the loop detection are performed.

In the following subsections, we explain some fundamental concepts to the system: keyframes, coordinates frames, how constraints are built and how optimization works.

3.3.1 *Keyframes*

hdl_graph_slam is based on keyframes estimated by the front-end of a SLAM system. The term *keyframe* comes from Visual SLAM systems, which estimate a camera trajectory and the environment given the images taken by the camera. To get accurate results, at non-prohibitive computational cost, just some frames taken by the camera are kept, the so-called keyframes [24]. In Visual SLAM keyframes are required to store the features extracted from each image, which are subsequently used to track the pose of the camera and to, eventually, reconstruct the environment. On the other hand, LiDAR SLAM performs tracking using LiDAR scans in order to estimate the odometry, and then build the map using the LiDAR scan themselves [48]. Nevertheless, the Graph SLAM structure can be implemented in both systems.

Standard graph-based approaches constantly add new nodes to the graph. As a result, memory and computational requirements grow over time, preventing long-term mapping applications. A continuously growing graph slows down graph optimization and makes it more and more costly to identify loop closures [25]. To solve this problem, not all the frames in Visual SLAM or not all the LiDAR scans in LiDAR SLAM are retained, thus only the keyframes remain.

hdl_graph_slam does not use all the odometry poses estimated by the scan matching to build a keyframe: a *keyframe_updater* decides to build a new keyframe if the corresponding odometry is distant from the previous one by certain linear and angular quantities, or a certain time duration has passed.

A *keyframe* represents a pose of the robot. Because of that, the terms *keyframe*, *keyframe pose* and *robot pose* are interchangeable and used as synonyms in this thesis.

In hdl_graph_slam, a keyframe is a data structure that contains all the information relative to a pose of the robot and it is used in the pose graph to add nodes and constraints associated with said pose.
The structure of the keyframe is explained in Figure 3.3:

- *stamp*: time at which the keyframe was acquired
- *odom*: odometry $\mathrm{SE}(3)$ pose, computed by the scan matching
- *accum_distance*: total distance traveled by the robot since the start, up to this keyframe; it is computed by the keyframe_updater using only the estimated odometry
- *cloud*: 3D point cloud associated to the current LiDAR scan
- *floor_coeffs*: coefficients of the plane detected in the point cloud
- *utm_coord*: GPS coordinates

KeyFrame
stamp : ros::Time odom : Eigen::Isometry3d accum_distance : double cloud : pcl::PointCloud<pcl::PointXYZI> floor_coeffs : Eigen::Vector4d utm_coord : Eigen::Vector3d acceleration : Eigen::Vector3d orientation : Eigen::Quaterniond node : g2o::VertexSE3*
id() : long = node→id() estimate() : Eigen::Isometry3d = node→estimate()

Figure 3.3: Class diagram of a keyframe.

- *acceleration*: 3D IMU linear acceleration
- *orientation*: 3D IMU orientation
- *node*: reference to the g2o node in the pose graph associated to this keyframe
- *id()*: returns the id of the keyframe, that is the same of *node*
- *estimate()*: returns the current pose estimate of the robot associated to this keyframe

The only mandatory elements of a keyframe are *cloud*, *odom* and *node*, because hdl_graph_slam is a LiDAR Graph SLAM system with optionally IMU and GPS priors.

3.3.2 *Coordinate Frames*

hdl_graph_slam follows the ROS conventions for coordinate frames of mobile platforms [26]. The coordinates frames defined by the system following such conventions are *base_link*, *odom* and *map*.

base_link is rigidly attached to the mobile robot base. It can be placed in any arbitrary position and orientation, also depending on the hardware. *base_link* tells the pose of the robot in the reference frame of choice, e.g., the pose of the robot in *odom* frame is the transformation between *odom* and *base_link*.

odom is a world-fixed frame. The pose of a robot in *odom* frame can drift over time, so it cannot be used as a long-term global reference. However, such pose in *odom* frame is guaranteed to be continuous, which means it evolves

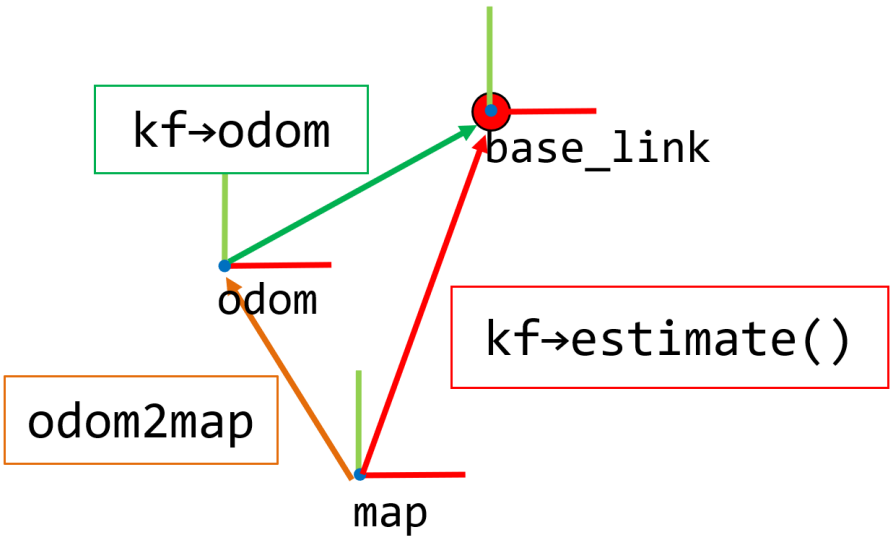


Figure 3.4: Keyframe estimate as a composition of transformations. In red the transformation between *map* and *base_link*, which is the keyframe estimate; in orange the transformation between *map* and *odom*, which is odom2map; in dark green the transformation between *odom* and *base_link* which is the keyframe odometry; the red dot is the keyframe node.

smoothly over time without abrupt jumps. It is usually computed based on an odometry source.

map is a world-fixed frame. The pose of a robot in *map* frame does not drift over time, however, it is not continuous, meaning that there could be discrete jumps. It can be therefore used as a long-term global reference. The pose of the robot in *map* frame is constantly recomputed using sensor measurements in order to eliminate the drift. The transform from *odom* to *base_link*, $\mathrm{TF}_{\mathtt{odom}\rightarrow\mathtt{base\_link}}$, which is the dark green arrow in Figure 3.4, is computed and published by the scan matching from scan_matching_odometry _nodelet and it is the same as the one contained in the *nav_msgs::Odometry* message published by the same nodelet and stored in the keyframe. Thus, the odometry contained in a keyframe is the pose of the robot in *odom* frame.

The transform from *map* to *odom*, $\mathrm{TF}_{\mathtt{map}\rightarrow\mathtt{odom}}$, which is the orange arrow in Figure 3.4, is computed and published by the hdl_graph_slam_nodelet. The nodelet builds the graph using various information, in which there is also the information relative to odometry given by the scan matching and then optimizes it. After the optimization, the estimates of the g2o nodes representing the poses of the robot are updated: because of how the graph is built, these estimates are in *map* frame. Thus, the estimates tell the transforms between *map* and *base_link*, $\mathrm{TF}_{\mathtt{map}\rightarrow\mathtt{base\_link}}$, the red arrow in Figure 3.4. hdl_graph_slam_nodelet takes the last keyframe stored in the system, and,

since its $TF_{map \to base\_link}$ is known from its g2o node estimate and its $TF_{odom \to base\_link}$ is known from its odometry, computes and publishes the transform between *map* and *odom*:

$$TF_{map \to odom} = TF_{map \to base\_link} * (TF_{odom \to base\_link})^{-1} \tag{3.1}$$

This transform is stored into the system with the name `odom2map` and is continuously updated, whenever optimization is performed.

Transforms between *base_link* and specific sensors coordinate frames are user defined.

3.3.3 *Types of Constraints*

g2o provides different classes to insert nodes and edges in the pose graph. They can be easily extended to create new types of nodes and edges to better suit each need.

In the following, we distinguish two types of g2o edges, namely "constraints" and "priors". A *constraint* is a g2o edge between two g2o nodes, thus it describes a relative pose between them. A *prior* is a g2o edge that defines a measure attached to a g2o node (i.e., a unary edge). Odometry, for example, is a constraint because it defines a relative transformation between two keyframes poses (see Subsection 3.3.4 for more details). GPS data is, instead, inserted into the pose graph as a prior because it is a measure of where the robot is in the world at a certain time, and it does not involve another pose of the robot (see Subsection 3.3.5 for more details).

3.3.4 *Odometry Constraints*

An odometry constraint represents the relative motion between two consecutive robot poses, using the data received from scan_matching_odometry_nodelet. The callback function is called *cloud_callback*, and it is executed when data from the point cloud topic (from prefiltering_nodelet) and from the odometry topic (from scan_matching_odometry_nodelet) are received at the same time and synchronized. The function calls keyframe_updater with the newly received data: in case of positive response, i.e., when a certain distance has been traveled or some time has passed, a new keyframe is created and put into the queue named *keyframe_queue*. The new keyframe is initialized with the received point cloud and odometry.

The flush queue function that creates and inserts keyframe nodes and odometry constraints into the pose graph is called *flush_keyframe_queue*. This function processes all the keyframes in the queue in order of insertion. For each `keyframe`, a new g2o node is created and inserted into the graph using

as initial estimate a $SE(3)$ transformation, called $keyframe\_pose$, which is the pose of the robot in *map* frame:

$$keyframe\_pose = odom2map * keyframe \to odom \tag{3.2}$$

Given the previous keyframe, $prev\_keyframe$, the relative pose between it and the current keyframe, which defines an odometry constraint between them, is given by:

$$relative\_pose = (keyframe \to odom)^{-1} * prev\_keyframe \to odom \tag{3.3}$$

This relative pose is used to insert an edge between the g2o nodes corresponding to the previous and current keyframes.

3.3.5 *GPS Priors*

GPS constraints are designed to compensate for the accumulated rotation error of the scan matching in outdoor environments where the ground is not flat [23] and as global landmarks to anchors to.

GPS coordinates are usually provided in the form of latitude, longitude, and altitude (*lla*). hdl_graph_slam converts them in the UTM coordinate system and then to the ENU coordinate system. The UTM coordinate system divides the earth into 60 zones (vertical bands of 6° each) and projects them on a plane using the transverse Mercator projection [27]. A point on a zone can be specified using x and y, both specified in meters, taking as origin the intersection between the equator and the zone's central meridian. To specify a location on the Earth using UTM, we need to specify the zone, x, and y.

All the positions used in hdl_graph_slam, i.e., keyframe positions and GPS, are referred to the GPS coordinates of the first keyframe stored into the system, called *zero_utm*. This means that from UTM coordinates we pass to ENU coordinates. ENU coordinates are an arbitrary coordinate system defined on a planar projection of the Earth [47]. A point on this planar projection is specified through x, called *easting*, y, called *northing*, and z, called *up*. In our case, the origin of the coordinate system is zero_utm.

hdl_graph_slam uses a 3D vector that specifies easting, northing, and altitude.

Moreover, the system supports several types of ROS GPS messages: nmea_msgs::Sentence, geographic_msgs::GeoPointStamped and sensor_msgs::NavSatFix. In all cases, these messages are eventually converted into a geographic_msgs::GeoPointStamped and inserted into the queue called *gps_queue*. If the altitude is not provided, then it will be set to NaN.

The flush function is called *flush_gps_queue*. For each keyframe stored in the system that does not have any associated GPS data, it searches for a GPS message in the gps_queue that is the closest in time to the keyframe. If the

closest GPS message is not too far in time from the keyframe (max 0.2 seconds of difference) then the GPS message is converted from lla to UTM, referred to zero_utm, and inserted into the keyframe. Finally, the GPS is inserted into the graph as a prior associated to the current keyframe node.

3.3.6 *IMU Priors*

hdl_graph_slam waits for *sensor_msgs::Imu* messages and when it receives them, it executes the *imu_callback* function. This function stores the IMU message into the *imu_queue*.

The flush function is called *flush_imu_queue*. In a similar fashion to flush-_gps_queue, for each keyframe that doesn't have any IMU message associated, it searches for the IMU message in the queue that is closest in time to the keyframe and that is not too far in time (max 0.2 seconds of distance) from the keyframe. If such IMU message is found, then the orientation is extracted from it as a quaternion and linear acceleration is extracted as a 3D vector. Both are converted from the IMU reference frame to *base_link* and inserted into the keyframe. Orientation is inserted into the graph as a quaternion prior associated to the keyframe pose. Linear acceleration is inserted into the graph as a 3D vector prior associated to the keyframe pose with the direction of the acceleration as the vertical pointing downward (as the gravity vector).

3.3.7 *Floor Constraints*

Floor constraints are designed to compensate for the accumulated rotation error of the scan matching in large flat indoor environments [23].

Floor plane coefficients messages published by floor_detection_nodelet are read by hdl_graph_slam_nodelet and put in the *floor_coeffs_queue* by the corresponding callback function *floor_coeffs_callback*.

The flush function is called *flush_floor_queue*. This functions cycles over all the messages in the queue and for each of them, it searches a keyframe that has the same timestamp. If that keyframe is found and it is the first time the system executes the function, the plane $z = 0$ is added as a g2o plane node. After that, in any case, an edge between the keyframe node and the plane node is added using the received floor coefficients as a relative transformation between the two. hdl_graph _slam assumes the robot to have a planar motion since the keyframes are always linked to the plane $z = 0$. A downside of this assumption is that it may be true for indoor environments since buildings' floors are generally flat, but not for many outdoor scenarios (e.g., urban), because roads usually have slopes.

3.3.8 *Loop Detection and Optimization*

The task of optimization is determining the most likely configuration of the poses given the edges of the graph [1]. The problem that optimization has to solve is a non-linear least square problem.

The optimization function, called *optimization_timer_callback* is run every 3 seconds (although this time interval can be changed). This function calls all the flush functions. If one or more flush functions return true, which means new data has been inserted into the graph, then loop detection and optimization are executed.

hdl_graph_slam uses the g2o package to perform optimization [23], using the Levenberg-Marquardt algorithm. Other algorithms can be used, such as Gauss-Newton.

Loop detection may be performed using various registration algorithms. When a loop is detected, an edge between the two keyframes that constitute the loop is inserted. To find a loop, for each keyframe, the system searches for candidates in a certain range from the keyframe node. For each of them, scan matching is performed between the candidate point cloud and the current keyframe point cloud. The alignment with the lowest error is retained as a loop.

3.3.9 *Information Matrix*

The *information matrix* takes into account the uncertainty of the Gaussian approximation of an edge. When inserting an edge into the graph, we need to provide a corresponding information matrix. The information matrix $\mathbf{\Omega}$ is the inverse of the covariance matrix $\mathbf{\Sigma}$ [35]:

$$\mathbf{\Omega} = (\mathbf{\Sigma})^{-1} \tag{3.4}$$

The *covariance matrix* $\mathbf{\Sigma}$ is a squared symmetric positive semi-definite matrix that gives the covariances between each pair of elements from a set of random variables [34]. For an edge stemming from a transformation, these elements are the components of the state vector of the type of transformation.

A generic element of the covariance matrix is σ^2_{ij}, the *covariance* between the i-th and j-th components. Given that, the components of a measure are independent one from the other, so the covariances which are not on the main diagonal are equal to zero. The elements on the diagonal, corresponding to $i = j$, are the *variances* $\sigma^2_{ii} = \sigma^2_i$ of the i-th component, with σ_i being the *standard deviation*. Because of these considerations and because of equation 3.4, the components of an information matrix are in the form $1/\sigma_i$.

If we select a low variance of a component, which means that the uncertainty on the measure of it is believed to be low, the corresponding value on the information matrix is high. Vice versa, if the variance is high, the uncertainty

on the measure of that component is believed to be high, and the corresponding value on the information matrix is low.

The information matrices for all the types of constraints can be computed in a fixed way. Each constraint of the same type has always the same information matrix.

Since odometry and loop detection constraints are defined between two $SE(3)$ poses of the robot, they are constraining all the 6 Degrees of Freedom of the robot (which in this case are the components mentioned above). The Degrees of Freedom correspond to translations in the directions of Cartesian axes (i.e., x, y and z) and the rotation around the same axes (i.e., qx, qy and qz), respectively. In hdl_graph_slam, the variances on x and y are the same, and the variances on qx, qy, and qz are also equal between themselves. The information matrix for an odometry constraint is in the following form:

$$\mathbf{\Omega}_{odometry} = \begin{pmatrix} 1/\sigma^2_{xy_o} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/\sigma^2_{xy_o} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/\sigma^2_{z_o} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/\sigma^2_{q_o} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/\sigma^2_{q_o} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/\sigma^2_{q_o} \end{pmatrix} \tag{3.5}$$

where the subscript o stands for odometry, indicating the standard deviations specific for odometry.

Since GPS constraints define a 3D vector of ENU coordinates, or 2D if altitude is not provided, the information matrix tells the uncertainty on such coordinates. Thus, the information matrix for 3D GPS constraints is:

$$\mathbf{\Omega}_{GPS} = \begin{pmatrix} 1/\sigma^2_{xy_{gps}} & 0 & 0 \\ 0 & 1/\sigma^2_{xy_{gps}} & 0 \\ 0 & 0 & 1/\sigma^2_{z_{gps}} \end{pmatrix} \tag{3.6}$$

Also, GPS specifies the same standard deviation for x (easting) and y (northing) and a different one for z (altitude). If altitude is not provided, then the information matrix reduces to the 2×2 upper block of $\mathbf{\Omega}_{GPS}$.

The information matrix of orientation constraints defines the uncertainty along the three rotational degrees of freedom provided by the quaternion defining the constraint. The standard deviation σ_q is the same for all of them. The information matrix is:

$$\mathbf{\Omega}_{orientation} = \begin{pmatrix} 1/\sigma^2_q & 0 & 0 \\ 0 & 1/\sigma^2_q & 0 \\ 0 & 0 & 1/\sigma^2_q \end{pmatrix} \tag{3.7}$$

Acceleration constraints provide the linear acceleration along the x, y and z coordinates of the 3D space. A unique standard deviation σ_{acc} for all the dimensions is provided. The information matrix is:

$$\mathbf{\Omega}_{acceleration} = \begin{pmatrix} 1/\sigma_{acc}^2 & 0 & 0 \\ 0 & 1/\sigma_{acc}^2 & 0 \\ 0 & 0 & 1/\sigma_{acc}^2 \end{pmatrix} \tag{3.8}$$

A plane constraint basically tells how much the detected plane of the current keyframe varies with respect to the plane $z = 0$. The information matrix tells the uncertainty along the three directions x, y and z of a plane in 3D. Also here only one standard deviation σ_{plane} is provided. The information matrix is:

$$\mathbf{\Omega}_{plane} = \begin{pmatrix} 1/\sigma_{plane}^2 & 0 & 0 \\ 0 & 1/\sigma_{plane}^2 & 0 \\ 0 & 0 & 1/\sigma_{plane}^2 \end{pmatrix} \tag{3.9}$$

For odometry constraints and loop detection constraints, the information matrix can be either fixed and user-defined or computed evaluating the quality of the scan matching, using the custom class *information_matrix_calculator*. Given two keyframes, which g2o nodes are connected in the pose graph via an odometry constraint or a loop constraint, the corresponding point clouds are passed as inputs to the information_matrix_calculator. The oldest keyframe point cloud is called *source point cloud* and the current keyframe point cloud is called *target point cloud*. information_matrix_calculator starts computing the *fitness score* associated with the alignment between the source point cloud and the target point cloud, obtained through scan matching. The fitness score tells how much the relative pose is good at making the previous keyframe point cloud align with the current keyframe point cloud. If it is low, then the relative pose transformation is good and the point clouds overlap well, and vice versa.

First, the source point cloud is transformed via the relative pose transformation, so that in this way it maximally overlaps with the target point cloud. Secondly, for each point in the transformed source point cloud, the calculator searches for the closest point in the target point cloud and computes the Euclidean squared distance between them. The fitness score is the mean over all these Euclidean squared distances.

The fitness score is used to compute two weights, one for the translational components, w_x, and one for the rotational parts, w_q. These weights act as variances that will be used to compute the information matrix. First, $weight\_percentage$ is computed:

$$weight\_percentage = \frac{1.0 - e^{-var\_gain * fitness\_score}}{1.0 - e^{-var\_gain * fitness\_score\_threshold}} \tag{3.10}$$

where var_gain tells the step with which variance varies, $fitness\_score$ is the computed fitness score and $fitness\_score\_threshold$ is the maximum expected fitness score. $weight\_percentage$ is the same for w_x and w_q and tells a number which represents where the current fitness score stands on a scale between 0 (lowest fitness score equal to 0) and 1 (highest fitness score equal to $fitness\_score\_threshold$). The weights are computed as:

$$w_i = min\_var_i + (max\_var_i - min\_var_i) * weight\_percentage \quad (3.11)$$

with i equal to x or q. min_var_i is the desired minimum variance and max_var_i is the desired maximum variance.

The information matrix is computed as:

$$\mathbf{\Omega}_{odometry} = \begin{pmatrix} 1/w_x & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/w_x & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/w_x & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/w_q & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/w_q & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/w_q \end{pmatrix} \quad (3.12)$$

We already explained that if the fitness score is low it means the odometry relative pose is good. If the fitness score is low, the weight is also low, meaning the corresponding information matrix value is high. A higher value in the information matrix means that the measure will have a major impact on the optimization than the measures with corresponding low values in the information matrix.

Recall Equation 2.8: we need to find the $\mathbf{x}$ that minimize the negative log-likelihood function $F(\mathbf{x})$. Following its definition in Equation 2.7, if the $\mathbf{\Omega}$ of a term of the sum is high, then the error $\mathbf{e}_{ij}$ has to be kept low, to try to keep the term to the minimum possible. Given the definition of $\mathbf{e}_{ij}$ in Equation 2.5, to keep the error low, the expected measurement has to be close to the real one. The real measurement is provided by the edge. The expected measurement depends on the $\mathbf{x}$ chosen by the optimization. Thus, the optimization chooses the values of the final poses $\mathbf{x}$ that have high information matrices to be near the measures, to try to keep the error low. This means that the measures with high information matrices have more weight on the optimization than the ones with low information matrices.

The system is giving more importance in the optimization to the odometry edges that are believed to be correct because of the low fitness score, and vice versa. The final values of the robot poses are nearer the measures suggested by the odometry edges that are believed to be correct than the others that are believed to be more imprecise.

This way of computing the information matrix has been described in detail because it will be a central concept in Chapters 4 and 5.

4 GRAPH SLAM WITH MAP PRIORS

The work proposed in this thesis is hdl_graph_slam_with_map_priors, a ROS package based on the hdl_graph_slam ROS package described in Chapter 3, which adds maps priors into the system, following the ideas explained in Subsection 2.5. hdl_graph_slam has been chosen because it is accurate and graph-based. Its structure with independent flush functions for each type of data allows to easily include new constraints in the pose graph. The proposed system keeps most of the features and characteristics of hdl_graph_slam, as it is built upon its structure: scan matching for odometry estimation, loop detection, pose graph optimization, GPS and IMU constraints.

hdl_graph_slam_with_map_priors adds a new type of constraint in the pose graph, based on the information associated to buildings from publicly available map services, such as Google Maps or OpenStreetMap. The system has been designed to work with OpenStreetMap, but it can be easily adapted to work with other map services. We chose OpenStreetMap because it is publicly available, free to use, and well documented.

Buildings are inserted into the pose graph as nodes, linked with the keyframe nodes through the aforementioned constraints. From an optimization point of view, building nodes are of the same type of keyframe nodes, thus the optimization finds the most likely configuration of keyframe nodes and building nodes given the constraints in the graph. This is different from what explained in Subsection 2.5.1, because, in that case, the transformation derived from the alignment was simply a measurement attached to the keyframe (a g2o prior, such as GPS and IMU priors) while here buildings are g2o nodes, linked to other nodes through g2o edges, whose position can be modified by the optimization. Because of that, we can modify the position of building estimates in order to be more coherent with the LiDAR scans, and thus with the environment. This has the advantage of making the system to be quickly adapted to changes in the environment. We may even use the system with sole purpose of mapping and correcting the buildings. At the same time, the robot positioning benefit of the building constraints even when other types of data (such as IMU or GPS) are not present or unreliable.

4.1 SYSTEM OVERVIEW

The working of hdl_graph_slam_with_map_priors is explained in Figure 4.1. Given the pose associated to a keyframe (red dots in the figure), the buildings surrounding its location are downloaded from OpenStreetMap. The data

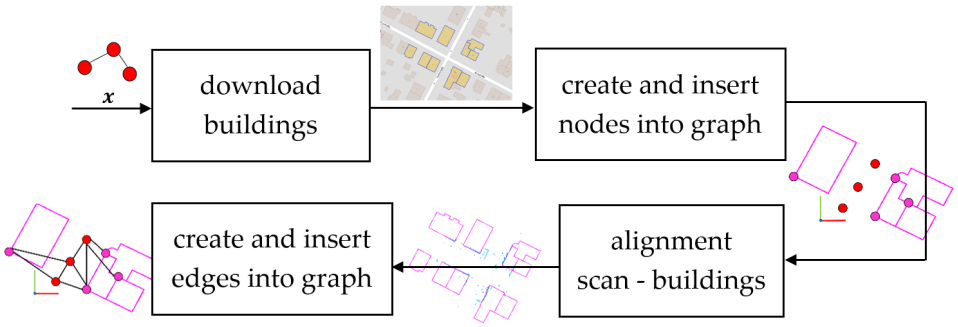


Figure 4.1: Workflow of hdl_graph_slam_with_map_priors

downloaded are parsed and each building is inserted into the system as a node in the pose graph (pink dots in the figure). Then, the relative pose between a keyframe node and a building node is computed through the alignment of the keyframe point cloud with the map containing all the buildings seen by that keyframe. This pose is used to insert an edge into the pose graph between the building nodes seen by the keyframe and the keyframe node. After that, the system goes on with optimization and loop detection.

The work proposed in this thesis stands in the back-end of hdl_graph_slam: before optimization, all the flush functions, included the one associated to buildings, are executed. The flush function of buildings downloads the buildings for each new keyframe inserted into the system, and inserts them and the corresponding edges into the pose graph as explained above. The execution of flush functions is inside the optimization callback, as explained in Subsection 3.3.8, which is executed every few seconds: thus, if there are new keyframes, the download of buildings and the corresponding insertion into the pose graph happen continuously every few seconds.

4.2 GEOGRAPHIC POSITION OF THE ROBOT

The download of buildings is associated to the keyframe of which we want to see the surrounding environment. In particular, it requires a geographic position in longitude and latitude coordinates. Since the GPS priors are not always present, the position estimate of the keyframe may be used as a geographic position to download buildings. Since buildings are ENU coordinates, but to be downloaded require latitude and longitude coordinates, at least one GPS coordinate is required. All the other ENU coordinates, including keyframe poses, may be referred to this single GPS, and thus used to download buildings, without the need for other GPS coordinates.

A problem arises when robot poses drift over time, and thus the robot poses are not anymore representative of the real geographic position of the robot. In this case, the download of buildings using the position estimate of the keyframe may led to the download of the wrong buildings. The robot poses drift when there are only odometry and/or IMU constraints, since the corresponding measures are imprecise because of noise in the data, modelling errors, integration errors and others.

When more precise constraints are added to the system, e.g. GPS priors, the keyframe position estimate is representative of the real geographic position of the robot, so it can be used to download buildings. The GPS is not always present or not always reliable. We could use a low frequency GPS to download buildings; such type of GPS is unreliable for correcting keyframe poses, but may be useful to loosely download the correct buildings.

Buildings are georeferenced, so, if their position is known with a sufficient accuracy, they constrain the keyframe position to be a correct geographic position. If the accuracy on buildings position is not high and there are no other accurate constraints in the pose graph, then the keyframe poses may drift and the download of buildings may be wrong. The accuracy on buildings may be low because they are usually mapped by a GPS, which itself is not perfect but has a defined accuracy, and because of human errors in mapping and submitting data to OpenStreetMap. The "sufficient accuracy" depends on the accuracy of the other constraints in the system. For example, if the accuracy on odometry constraints is lower than the accuracy on buildings constraints, and there are no other constraints in the system, then the robot pose may drift significantly while the buildings poses will be more stable.

As explained in Subsection 3.3.5, all the GPS coordinates in the system are ENU coordinates, which means they are defined with respect zero_utm, the first GPS coordinate stored in the system. Buildings positions, since they are geographic coordinates, are also ENU coordinates referred to zero_utm. When GPS priors are not used, zero_utm is not defined, but buildings require zero_utm to be downloaded and inserted into the system. Thus, at least the GPS and IMU coordinates of one keyframe are required. It may be specified manually when starting the system, or through a GPS and an IMU.

4.3 CONVERSION TO 2D

We encountered difficulties in the correct functioning of the system in 3D when adding building nodes as a g2o $SE(3)$ nodes and corresponding edges. Depending on which type of first node was fixed (a keyframe node, a floor node or a building node) and on the accuracy of the constraints, the optimization would correctly estimate keyframe poses in their real geographic position or not, leading to incorrect estimates of the poses, denying the possibility of downloading buildings in the correct location.

We explained before that odometry constraint are subjected to drift, which makes their exclusive usage not suitable to download buildings. Floor plane constraints do not add any information about the geographic position of the keyframes (only elevation and some adjustments in roll and pitch), thus their addition into the system does not improve the situation. When adding building constraints on top of odometry and floor constraints, the situation for the optimization becomes complicated. In fact, the system in 6 Degrees of Freedom requires two nodes of different types to be fixed. Fixed means that their position is set as the initial estimate and never modified again by the optimization, such that they can be used as "anchors" for the following poses to be referred to (the x_0 described in Section 2.1). When only two different constraints are present (i.e., odometry and floor constraints), we fix one node for each type and the optimization works well. When we add buildings constraint, we do not need to fix a building node, since we already have two. The optimization seems to struggle with three different types of constraints, having just two of them with a fixed node. The result is that buildings constraints are not taken in consideration, thus the keyframe poses do not represent a correct geographic position. Changing the fixed nodes, e.g. fixing a building node and a keyframe node, seems to not improve the situation, apart from one case (fixed the floor node and a building node) which still presents inaccuracies. Another factor to take in consideration is the accuracy of the constraints. Floor constraints are generally more precise than buildings ones, so the optimization gives naturally more weight to them, but since they do not provide geographic information, the keyframe poses will be more likely to drift.

Converting the system from 3D to 2D means passing from a system in 6 Degrees of Freedom (translation over x, y, z and rotation around x, y, z) to a system in 3 Degrees of Freedom (translation over x, y and rotation around z). This brings many advantages. First, the overall complexity of the workflow is reduced, as algorithms (e.g., scan matching) are dependent on the Degrees of Freedom considered, and passing from 6 to 3 greatly improve performances. Second, it is a more natural way to contextualize the problem, as maps are usually not given the elevation, hence falling into a planar representation.

In the following subsections, we explain in detail how all the components of the system have been converted to 2D.

4.3.1 *Keyframe Pose Nodes*

Keyframe pose nodes were converted from g2o $SE(3)$ nodes to g2o $SE(2)$ nodes.

4.3.2 *Floor Nodes*

Floor nodes were removed, since now the system is constrained to work on the plane $z = 0$.

4.3.3 *GPS*

GPS priors were converted from g2o *EdgeSE3PriorXYZ* edges to g2o *EdgeSE2PriorXY* edges. *EdgeSE2PriorXY* is a custom class adapted from *EdgeSE3PriorXY* custom class. GPS positions were converted from a 3D vector (easting, northing, altitude) to a 2D vector (easting, northing). In places where the altitude is still required, such as point clouds and the download of buildings, the altitude is set to zero.

4.3.4 *IMU*

IMU priors were converted from g2o *EdgeSE3PriorQuat* edges to *EdgeSE2PriorQuat* edges. *EdgeSE2PriorQuat* is a custom class adapted from *EdgeSE3PriorQuat*. Orientations were converted from quaternions to a single angle, representing the rotation around the z axis. This angle has a corresponding 2×2 rotation matrix.

4.3.5 *Point Clouds*

Point clouds have been left in 3D, with a 3D point type. Just for the point clouds involved in the data association with maps, the z coordinate of all points in all point clouds is set to zero. Point clouds has been left in 3D since the algorithms provided by PCL only works with 3D point clouds.

4.3.6 *Registration Algorithms*

The scan matching algorithms only works in 3D, with 3D point clouds as input. Since the point clouds in input to the algorithms have all the points with z coordinates equal to zero, the returned transformation is in 2D.

4.4 BUILDINGS

Buildings are at the core of the creation of the map that will be aligned with the LiDAR scan in order to insert map priors into the pose graph. We obtain them from OpenStreetMap: we first create a query asking for building in a certain area, then we send it to the OSM server and, lastly, we receive a response as an OSM XML file. Inside the XML, buildings are represented

as *way*, one of the main elements underlying the OpenStreetMap structure, as described in Section 2.4. The XML is parsed and buildings are stored as objects into the system.

In the following subsections we describe this particular workflow.

4.4.1 *OSM Query*

The query used to retrieve data from OpenStreetMap is of the following form:

```
(
  way["building"](around: radius, latitude, longitude);
);
(._;>;);
out;
```

The query retrieves all buildings in a certain area specified by a radius centered in a given position defined by latitude and longitude.

The rounded brackets opening at line 1 and closing at line 3 represents the union operator: the result set will be the union of all the results from the statements inside. In this case inside there is only one query statement, at line 2. The union itself, from lines 1 to 3, is a block statement. The query statement on line 2 retrieves all the buildings in an area of a given centered in a position with given latitude and longitude. The keywords `way["building"]` on line 2 specify that the statement retrieves all the ways that have a tag with key "building". The remaining part of the statement is a filter in the form `(around: radius, latitude, longitude)` which specifies that buildings have to be taken in a radius centered in the given position.

On line 4 there is the recursion statement, that is itself a union of two statements. The first statement is ._; that is an item standalone query. . (dot) indicates to read all items from the input set specified by the next character, in this case _, the default input set. The second statement is >; that is a recurse down standalone query. Given an input set, it produces a result set that is composed of: all nodes that are part of a way which appears in the input set, plus all nodes and ways that are members of a relation which appears in the input set, plus all nodes that are part of way which appears in the result set. Recursion is needed to specify whether to insert, in the result set, nodes outside the radius, which are members of ways that lie partly inside the considered area (the same for relations). The entire line 4 statement joins all items in the default input set with the recurse down result over the default input set, and then outputs the result back to the input set, overwriting it.

On line 5 there is the output statement, that outputs the content of a set; in this case, because no input set is specified, it will output the default input set _, that in this case contains the results from line 4.

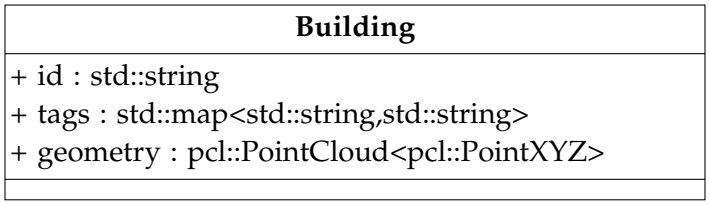


Figure 4.2: Class diagram of Building

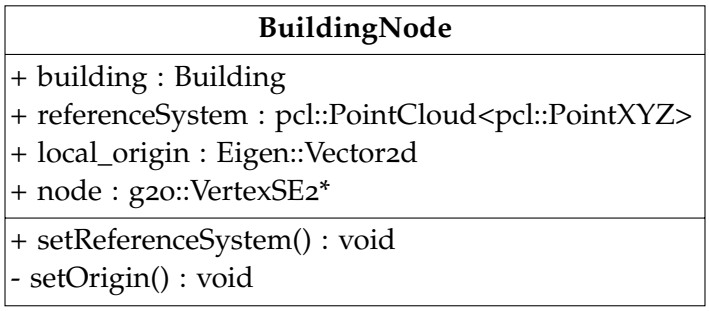


Figure 4.3: Class diagram of BuildingNode

4.4.2 *OSM Response*

The response is in XML form, as explained in Subsection 2.4.5. A building is represented as a *way*. Inside each *way* there are the *nd* references to the nodes: each *node* correspond to a vertex of the shape of the building seen from above. These nodes are represented and stored in the system as points of a point cloud. Inside each way there are also *tags* regarding characteristics and information of the building.

4.4.3 *Buildings Classes*

We describe here the three custom classes that have been created to handle buildings:

- *Building*: represents the building entity; it has the same structure as a *way* element in the XML returned by OSM; as showed in Figure 4.2
  - *id*: id of the building returned by OSM
  - *tags*: tags associated to the building, represented as a *std::map*
  - *geometry*: point cloud representing the shape of the building, already interpolated and referred to zero_utm
- *BuildingNode* : represent the building as an entity in the graph; it contains a reference to a corresponding Building, a reference to a g2o node and other useful data, as showed in Figure 4.3

  - *building*: corresponding Building object
  - *local_origin*: point in the *building.geometry* that acts as a reference for the building
  - *referenceSystem*: *building.geometry* point cloud referred to *local_origin*
  - *node*: g2o SE2 node representing the building
  - *setOrigin()*: set *local_origin*
  - *setReferenceSystem()*: set *referenceSystem* point cloud
- *BuildingTools*: contains static functions to download and parse buildings, as shown Figure 4.4
  - *Node*: internal structure used to represent an *OSM Node*
  - *getBuildings()*: public method that sequentially calls *downloadBuildings()* and *parseBuildings()*
  - *downloadBuildings()*: performs the request to download buildings and return the response given by OSM
  - *parseBuildings()*: takes the response from OSM and parse it into a Building vector
    * *buildPointCloud()*: builds and returns the point cloud for a *Building*, already interpolated and referred to *zero_utm*
      · *getNode()*: given a node reference and the list of all *Node*s in input, returns the corresponding *Node*
      · *toUtm()*: refers a point to *zero_utm*
      · *interpolate()*: given an initial and final points, returns the point cloud containing the linear interpolation between them

4.4.4 *Download of Buildings*

The download of buildings is done by the function *downloadBuildings*. This function takes in input *latitude*, *longitude*, *radius* and *host*. It builds the query given the parameters in input and sends the request to OpenStreetMap. The request is sent through the use of cURLpp library [36]. The host parameter is used to choose the OpenStreetMap server to which send the request. The request is a string in the following form:

```
host + "/api/interpreter?data=query"
```

where the query is the one explained in Subsection 4.4.1. The output is a string, representing the response from OpenStreetMap. The response from OpenStreetMap may be:

BuildingTools
+ struct Node {id : std::string, lat : double, lon : double}
+ getBuildings(lat : double, lon : double, rad : double, zero_utm : Eigen::Vector2d, host : std::string) : std::vector<Building> - downloadBuildings(lat : double, lon : double, rad : double, host : std::string) : std::string - parseBuildings(result : std::string, zero_utm : Eigen::Vector3d) : std::vector<Building> - buildPointCloud(nd_refs : std::vector<std::string>, nodes : std::vector<Node>, zero_utm : Eigen::VEctor3d) : pcl::PointCloud<pcl::PointXYZ> - getNode(nd_ref : std::string, nodes : std::vector<Node>) : Node - toUtm(pt : Eigen::Vector3d, zero_utm : Eigen::Vector3d) : pcl::PointXYZ - interpolate(a : PointXYZ, b : PointXYZ) : pcl::PointCloud<PointXYZ>

Figure 4.4: Class diagram of BuildingTools

- Data about the buildings
- Empty building set (i.e., there is no building in the specified radius)
- Error

In any case, the response is always in Overpass XML format, as explained in Subsection 2.4.6. If cURLpp encounters any error (e.g., no internet connection or timeout), the returned string will be empty.

4.4.5 *Parsing of Buildings*

The parsing of the response returned from OpenStreetMap is done by the function *parseBuildings*. This function takes in input the string representing the XML response from OSM and a 3D vector representing the *zero_utm* coordinate. The XML response is parsed through the use of the Boost Property Tree library [37]: first each OSM *Node* element is parsed and put in a vector of *Node* structs. Then each OSM *Way* element is parsed and the corresponding *Building* object is constructed. In output, there is a vector of *Building* objects. If there are errors in the parsing or there are no buildings, the returned vector is empty.

Each Building is built by initializing an empty object. The *geometry* point cloud is initialized by calling the function *buildPointCloud*, that parses the *nd* elements associated to the *Way* and creates a point cloud in which each point is given by a *Node* referenced by a *nd*. The function *toUtm* uses *zero_utm* to automatically refers to *zero_utm* a point of the point cloud. Since OSM returns

KeyFrame
stamp : ros::Time odom : Eigen::Isometry2d accum_distance : double cloud : pcl::PointCloud<pcl::PointXYZI> utm_coord : Eigen::Vector2d acceleration : Eigen::Vector2d orientation : Eigen::Rotation2D<double> node : g2o::VertexSE2* buildings_nodes : std::vector<BuildingNode> buildings_check : bool
id() : long = node→id() estimate() : Eigen::Isometry3d = node→estimate()

Figure 4.5: New class diagram of a keyframe

only the vertices of the polygon of a building, the function *interpolate* allows to create a point cloud of the outline of the buildings.

4.4.6 *Buildings in the Keyframe*

A keyframe is modified to contain a reference to the vector of buildings associated to it, *buildings_nodes*, and a boolean to check if the keyframe have already been checked for the presence of buildings in its surroundings, *buildings_check*. The new keyframe structure is represented in Figure 4.5.

4.5 INSERTING BUILDINGS INTO GRAPH SLAM

The management of buildings is done inside hdl_graph_slam_nodelet by a function called *update_buildings_nodes*.

This function does not have input arguments and has a bool as output, true if new buildings edges and/or nodes have been inserted into the graph, false otherwise. It is called by the optimization function when all the others flush functions are also called (see Chapter 3 for more information about flush functions and optimization).

update_buildings_nodes iterates over all the keyframes that do not have buildings associated with them. For each keyframe, it performs the operations explained in the following subsections in the order they are presented.

4.5.1 *Inserting Buildings Nodes*

update_buildings_nodes calls *getBuildings()* in order to obtain the buildings which are seen by the current keyframe. This function takes in input *latitude*, *longitude*, *radius*, *zero_utm* and *host*, and gives in output a vector of *Building* objects.

The vector of *Building* objects is used to build a corresponding vector of *BuildingNode* objects. hdl_graph_slam_nodelet uses this last vector to internally store a reference to all the buildings that it has ever seen since the start of the system, to avoid duplicates.

The query downloads all the buildings surrounding the keyframe, and the system parses all of them into Building objects. Then, for each *Building* that is new and has never been seen by the system, a *BuildingNode* is created and initialized with the reference to the *Building* itself. To identify the building in the pose graph, a reference point is chosen from *building.geometry* as the point with the lowest x coordinate, which is a corner. This reference vertex is *local_origin*, that is initialized by the method *setOrigin()*. After creating the *BuildingNode* object, we call the method *setReferenceSystem()*, that calls itself *setOrigin()*, to initialize *local_origin*, and then creates the point cloud *referenceSystem*, that is *building.geometry* but referred to *local_origin*. The building g2o node initial estimate is set with a position equal to *local_origin* and rotation as the identity matrix. Its reference is stored inside the *BuildingNode* object.

local_origin is fixed and it is always the initial estimate of the position of the g2o building node. After the optimization, the building node estimate position may not be equal anymore to *local_origin*, thus the system always takes as a reference for the building the building node estimate and not *local_origin*. The point cloud *referenceSystem* is also fixed and always refers to *local_origin*, thus it may be needed to transform it with the building node estimate to obtain an up to date point cloud.

4.5.2 *Point Clouds*

Two point clouds are needed: one representing all the buildings seen by the keyframe, called *buildingsCloud*, and one that is the scan given by the LiDAR, called *odomCloud*.

The *buildingsCloud* is built during the construction of the vector of *Building Node* objects, by merging together all the single *building.geometry* point clouds of the buildings surrounding the keyframe. The method *buildPointCloud()* of class *BuildingTools* is in charge of constructing the *building.geometry* for a *Building* object. It takes in input the list of *nd* associated to the way. These represent the corners of the outline of the buildings and are listed in consecutive order. For each couple of consecutive *nd*, the corresponding *nodes*

are retrieved and the function *interpolate()* is called with them in input. This function computes the linear interpolation between the two points in input, and return a point cloud containing all the interpolated points. All the point clouds coming from the interpolation are merged together in a unique point cloud, which is the final shape of the building, *building.geometry*.

Since *nodes* contain latitude and longitude coordinates, before calling interpolation, these are converted to UTM and then to ENU. When merging together the point clouds of all buildings, every single *building.geometry* is transformed using the current g2o node estimate of the building, such that the alignment is done with the buildings in the position estimated by the optimization, which may differ from the position downloaded from OpenStreetMap (see Section 4.6 for more information). This is a really important aspect: because the optimization may change the position of buildings following the constraints into the graph, always aligning to the OpenStreetMap buildings would mean to discard the job of the optimization and make the building to stay in its original OpenStreetMap position. The *odomCloud* is built starting from the point cloud stored in the keyframe. This point cloud passes through the following stages:

1. PassThrough: to remove the floor
2. Downsampling: to reduce the number of points in the cloud
3. Radius Outlier Removal: to filter out noise (like trees and cars)
4. Projection on the plane $z = 0$: since the point cloud is in 3D but we are operating in 2D
5. Transformation from *base_link* to *odom*

The resulting cloud is *odomCloud*.

4.5.3 *NDT-OMP Alignment*

The next step is to align *odomCloud* with *buildingsCloud*: as can be seen by Figure 4.6, *buildingsCloud* is the target point cloud, so it is kept fixed, while *odomCloud* is the source point cloud, so it is rotated and translated to match the target. In order to do this we use the NDT-OMP algorithm, because it is able to find a good alignment even in case of a large gap between the two initial point clouds.

The algorithm needs an initial guess to operate properly. The initial guess for the first keyframe is given by its GPS position and by its IMU orientation. The initial guess for the following keyframes is given by the transformation estimated by NDT-OMP for the previous keyframe.

The output of the algorithm is a transformation that aligns *odomCloud* to *buildingsCloud*. The algorithm operates only in 3D, although the z coordinate

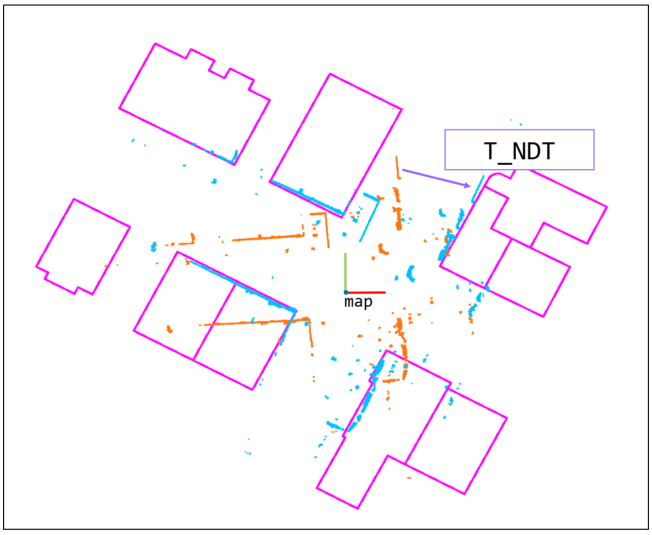


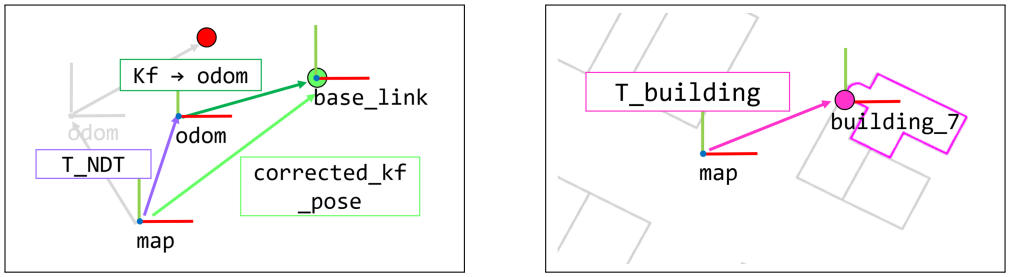
Figure 4.6: Alignment between *odomCloud* and *buildingsCloud*. In orange *odomCloud* in *odom* reference (frame not reported for clarity); in pink *buldingsCloud* in *map* frame; in light blue aligned *odomCloud*; the purple arrow is T_NDT.

of all point clouds is set to zero, so that a 2D alignment is performed. Because of this, the returned transformation is a $SE(3)$ transformation with the translation on z always equal to zero, and with the third column and row with only zero elements, except for their third element, which is one. For convenience, this transformation is converted to a $SE(2)$ transformation, called T_NDT.

4.5.4 *Computing Transformations*

To insert the edge between the keyframe pose node and the building node, we need to compute the relative pose between them.

First, we do an intermediate step to compute the pose of the keyframe with the information from the alignment. This pose defines a new transformation between *map* frame and *base_link* frame, and it is called corrected_kf_pose, since we obtain a new keyframe pose, which may differ from the current keyframe pose and eventually be more correct, because of the information from the alignment with the buildings. We start by examining the transformation returned from NDT-OMP, called T_NDT. This is a transformation between *odomCloud*, that is in *odom* frame, and *buildingsCloud*, that is in *map* frame (by definition, as it is composed by points that are geographic coordinates). So T_NDT defines a transformation between *map* and *odom*, such as odom2map, but given only by the alignment between the LiDAR scan and the buildings.



(a) (b)

Figure 4.7: Intermediate transforms used to compute T_scan_building. (a) corrected_kf_pose in light green, obtained by the composition of T_NDT in purple and kf → odom in dark green; the red dot is the keyframe g2o node; the green dot is the corrected keyframe pose (not a g2o node); (b) T_building, the pink arrow, is the transform between *map* frame and the building frame, building_7; the pink dot is the building g2o node.

Then, we take a look at the odometry pose stored in the keyframe: this is a transformation between *odom* and *base_link*. As showed in Figure 4.7a, we build corrected_kf_pose, which is a transformation between *map* and *base_link*, by composing T_NDT and keyframe → odom:

$$\text{corrected\_kf\_pose} = \text{T\_NDT} * \text{keyframe} \rightarrow \text{odom} \tag{4.1}$$

This pose may be inserted into the pose graph as a g2o prior attached to keyframe node, obtaining exactly what explained in Subsection 2.5.1. We now loop over all the *BuildingsNode* objects associated to the keyframe, in order to compute a relative transformation between the keyframe and each building and then insert into the pose graph the corresponding edge. Given a BuildingsNode object, we retrieve the current pose of the building from the associated g2o node and we call it T_building, as represented in Figure 4.7b. It is a transformation between the *map* frame and *building_{id}* frame, a frame positioned on the building node, where *{id}* is substituted with the id number of the building. Given corrected_kf_pose and T_building, both in *map* frame, we can compute the relative pose between the keyframe node and the building node, called T_scan_building as:

$$\text{T\_scan\_building} = (\text{T\_building})^{-1} * \text{corrected\_kf\_pose} \tag{4.2}$$

The composition of transformations is show in Figure 4.8.

4.5.5 *Inserting Edges*

T_scan_building is used to insert an edge between the starting keyframe node (not the corrected keyframe, which serves only as a virtual step) and

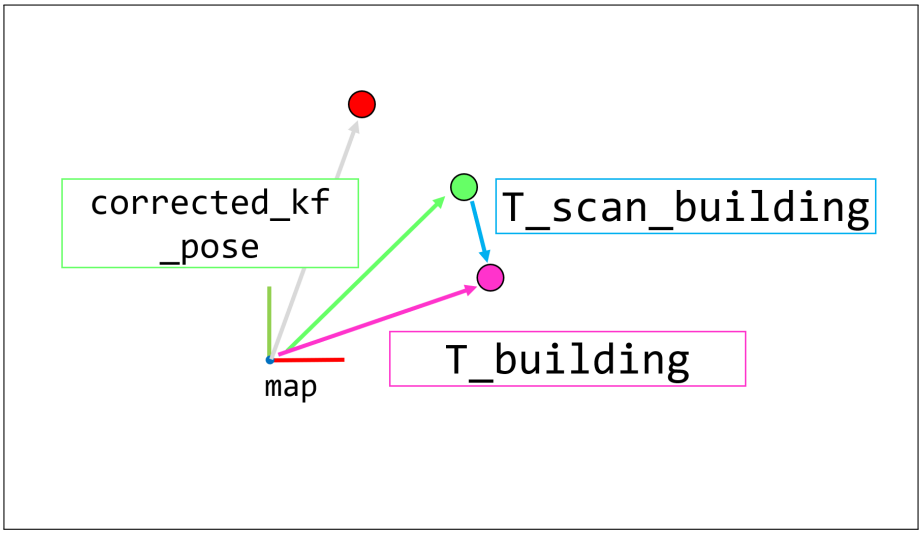


Figure 4.8: T_scan_building as a composition of transformations. The light green arrow is the corrected_kf_pose transformation and the light green dot is the corrected keyframe pose (not a g2o node); the pink arrow is the T_building transformation and the pink dot is the building g2o node; the light blue arrow is T_scan_building transformation; the red dot is the keyframe g2o node.

the building node. The information matrix is computed as explained in Subsection 3.3.9 for odometry and loop detection constraints. We use as target point cloud *buildingCloud*, as source point cloud *odomCloud*, and as transformation T_NDT. Given a keyframe, the information matrix is the same for all the edges connecting the keyframe node to a building node. By computing the information matrix in this way, the edges corresponding to a good NDT-OMP alignment have a major weight in optimization with respect to the ones stemming from a worse alignment.

4.6 WHAT HAPPENS DURING OPTIMIZATION

In this section, we explain how buildings constraints influence optimization and how the motion of the position of the buildings is achieved. In Subsection 4.6.1 we explain how a keyframe and a building, linked by a building constraint, influence themselves, while in Subsection 4.6.2 we describe how multiple buildings constraints influence a single building.

When talking about buildings, we always refer to their position, because this is where the most evident motion occurs. In reality, the optimization may decide to change also the orientation of the building.

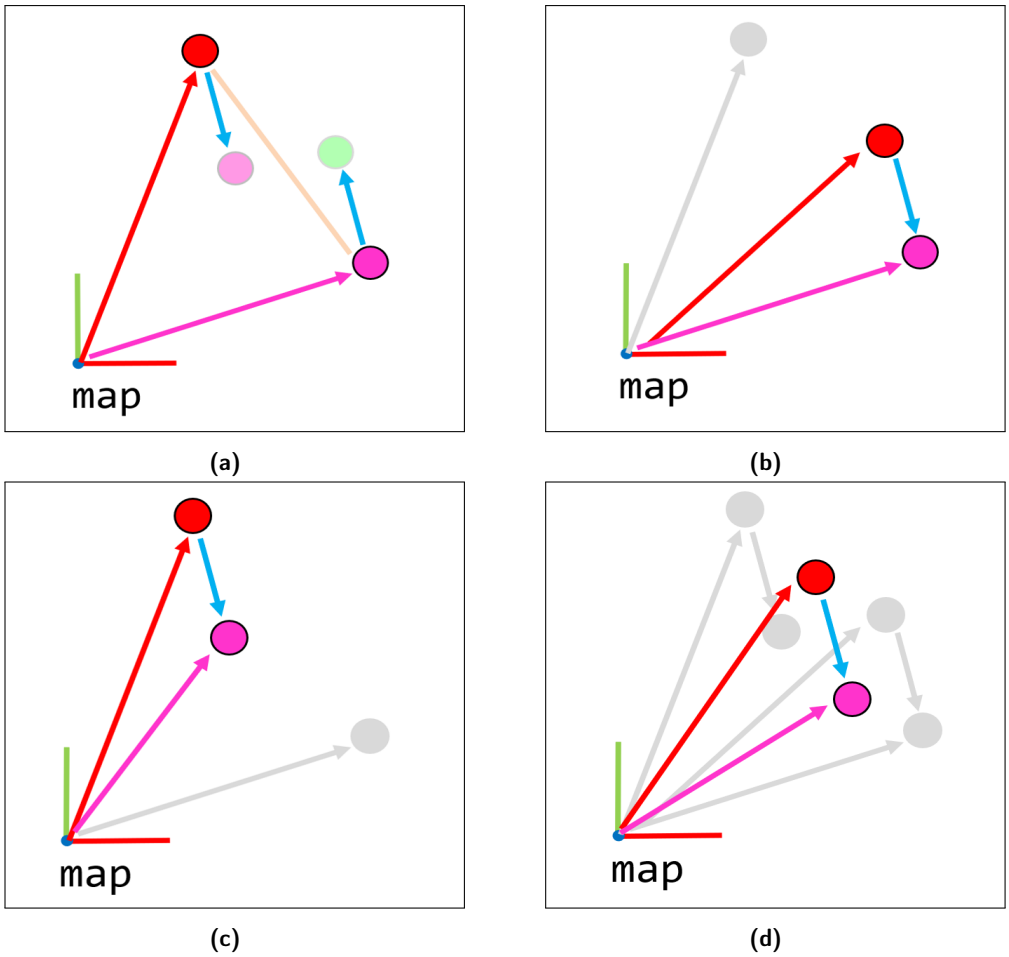


Figure 4.9: Results from optimization given a single keyframe-building edge. (a) Situation before optimization: in red the keyframe estimate transformation (arrow) and keyframe g2o node (dot); in pink the building estimate transformation (arrow) and building g2o node (dot); in orange the g2o edge between the keyframe node and building node; in light blue the T_scan_building transformation; the transparent pink dot is the building pose seen by the keyframe node, as inferred by T_scan_building; the transparent light green dot is corrected_keyframe_pose, which is the keyframe pose seen by the building node, as inferred by T_scan_building; (b) first limit case after optimization: the g2o building node is kept fixed and the g2o keyframe node is moved to the corrected keyframe pose (c) second limit case after optimization: the g2o keyframe node is kept fixed and the building g2o node is moved to the position indicated by T_scan_building; (d) intermediate case after optimization: both the g2o nodes are moved in new intermediate positions; in (b), (c), (d) the orange edge is still present but overridden by the blue arrow.

4.6.1 *Building - Keyframe Interaction*

When optimization is performed, the optimizer faces the situation described by Figure 4.9a: The g2o node corresponding to a keyframe, which is the red dot, is attached to a g2o node associated to a building, the pink dot, with an edge, the orange line, which relative transformation is T_scan_building, represented as the light blue arrow.

In Figure 4.9a, the transparent pink dot is the pose of the building seen from the keyframe, as inferred by T_scan_building (light blue arrow). There is obviously a mismatch between where the keyframe sees the building (transparent pink dot) and where the building really is (pink dot).

The situation can be seen also from the other side: the transparent light green dot is where the building node sees the keyframe, as inferred by T_scan_building (light blue arrow). Also here, there is a mismatch between where the building thinks the keyframe is (transparent light green dot) and where the keyframe is in reality (red dot).

The optimization has to solve an apparent paradox due to this mismatch and, thus, has to decide where to position the keyframe g2o node and the building g2o node to make the relative transformation between their corresponding poses coherent with T_scan_building.

In Figure 4.9b is described a first limit case: the building g2o node is kept fixed and the keyframe g2o node is moved to be coincident with the corrected keyframe pose. In Figure 4.9c is described a second limit case: the keyframe g2o node is kept fixed and the building g2o node is moved to the position told by T_scan_building. What is most likely to happen is that both g2o nodes will be moved in some intermediate position, as it can be seen in Figure 4.9d.

The limit case showed in Figure 4.9b is useful when we have very precise maps: in this case the SLAM system will be guided by these maps, in order to increase the accuracy on the poses of the robot. The second limit case of Figure 4.9c, instead, can be used when we want to correct maps that are known to be imprecise: assuming that the SLAM system is a lot more precise than the map itself, it uses the keyframes as anchors to eventually move the buildings. An example of usage may be the re-mapping of an area after many years. In case of comparable precision between maps and SLAM system, the intermediate case of Figure 4.9d is more suited. These cases highlight the flexibility of hdl_graph_slam_with_map_priors, so that the Graph SLAM system adapts to all possible situations depending on the context.

We conclude that the optimization has the power to correct the keyframe poses, thus eliminating the drift, but also to correct building positions, to make them more coherent with what was detected by the LiDAR scans. The results indicates that, while the drifting of the keyframe is correctly compensated for, the building position are not always correctly moved,

because the T_scan_building is computed using the global alignment T_NDT, which may not be perfectly fitting for each single building. In the next chapter, we propose a way to overcome this problem and compute the relative transformations and the information matrices, used to insert edges into the graph, in a way which is more tailored to each building.

Figure 4.9 is just a simple example with only two nodes and one edge: when there are more nodes and more edges of different types, the situation becomes more complex and the optimization may not be able to exactly satisfy all the relative transformations between nodes. In the particular cases, after optimization, the chosen poses are able to satisfy the relative transformations of the edges, such that the orange lines are coincident with the light blue arrows.

4.6.2 *Multiple Keyframes - Single Building Interaction*

We examine the situation when a single building is seen by many keyframes, which means there are many buildings constraints with the same building node. For convenience in the explanation, we assume that keyframes nodes are fixed, so that only the building node is moved.

In Figure 4.10a is presented the situation before optimization: we have a building, the pink dot, and its initial estimate, the pink arrow. A first keyframe node, the red dot, is linked with the building node through the orange edge, with relative transformation given by the light blue arrow. In a similar way, a second keyframe node, the brown dot, is linked with the building node through another orange edge (different from the previous one, just the colors are the same for clarity), with relative transformation given by the dark blue arrow. In both keyframes, we can clearly see the mismatch between the real relative position between the keyframe-building and the one told by the edge (see the previous subsection for more details). The two keyframes, because of their relative transformation, see two different building positions (the transparent pink dots), both different also from the initial estimate position (pink dot). Given the initial estimate of the node (pink node) and the two positions seen by the keyframes (transparent pink nodes), the optimization decides where to eventually move the building node in order to maximally satisfy the constraints. This decision is based on the information matrices of the two constraints. As explained in Subsection 4.5.5, if a constraint has an information matrix with high values, then it will have more importance in the optimization with respect to another constraint with an information matrix with low values.

In Figure 4.10b, the constraint between the first keyframe and the building (red - light blue) has an information matrix higher in values with respect to the second keyframe - building constraint (brown - dark blue). Thus the constraint related to the first keyframe has more weight in the optimization.

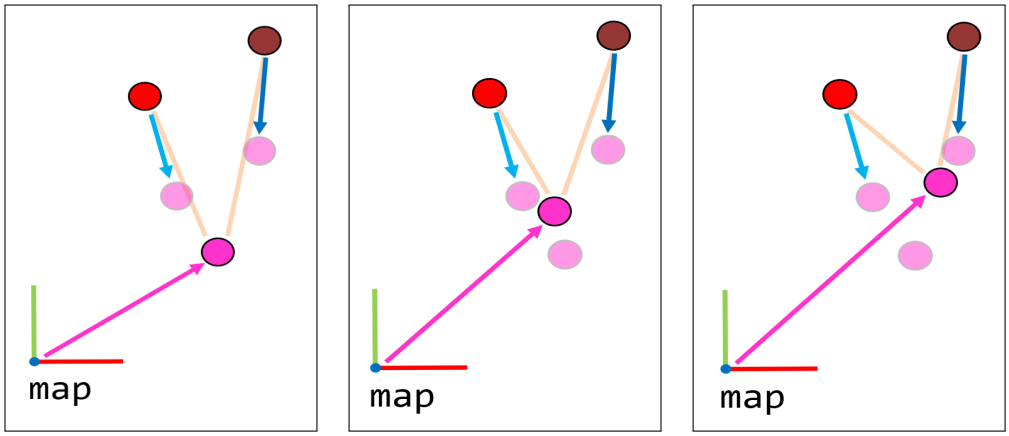


Figure 4.10: Results from optimization given multiple keyframes-building edges. (a) Situation before the optimization: the pink dot is a building node and the pink arrow is its initial estimate; the red dot is a first keyframe node that is linked with the pink building node through the orange edge, with relative transformation given by the light blue arrow; the brown dot is a second keyframe linked with the same pink building node through another orange edge, with relative transformation as the dark blue arrow. The transparent pink nodes are where the keyrframes see the building node, because of their relative transformations; (b) first case after optimization: the edge from the first keyframe has more weight in the optimization than the second one; (c) second case after optimization: the edge from the second keyframe has more weight in the optimization that the first one.

Consequently, after the optimization, the position of the building node is near the position of the building seen by the first keyframe. In Figure 4.10c the opposite case is explained: the information matrix of the constraint between the second keyframe node and the building node is higher in value with respect to the constraint stemming from the first keyframe. Thus, after optimization, the position of the building node is near the pose of the building seen by the second keyframe. Note how, in both cases, the position of the initial estimate of the building node (pink dot in Figure 4.10a) is taken in the consideration by the optimization, such that the position of the building node after optimization slightly hangs towards it. Also note how the optimization is not able to perfectly satisfy the relative transformations $T\_scan\_building$ associated to the edges, such that in both cases the orange edges does not coincide with the light blue and dark blue arrows.

The figures presents a simplified example of what may happen. When multiple constraints with different information matrices are present, the situation is much more complicated.

We now describe why different keyframes may see the same building in different positions and some problems related to it.

Figure 4.11: NDT-OMP alignment on wrongly positioned buildings. The NDT-OMP alignment between the LiDAR point cloud (light blue) and the buildings point cloud (pink) is correct, but since the buildings are wrongly positioned, the LiDAR point cloud is aligned correctly on the left side but not on the right side.

When all the keyframes see the same building in the same position, then the building position is not modified, even if the building is wrongly positioned. This may happen because when buildings are wrongly positioned, what happens is that the point cloud of a first keyframe is positioned correctly with respect to a group of lines in the buildings point cloud, but not with respect to other lines. This behavior can be seen in Figure 4.11: the "corridor" in the LiDAR point cloud (light blue) is aligned correctly with the buildings (pink) on the left, but not with the ones on the right.

If the registration algorithm aligns the point cloud of a second keyframe (temporally after the first) to the second group of lines (buildings on the right in Figure 4.11), then the two keyframes will see the building each one in a different position. If there is not this "jump" in the alignment, then the system is not able to understand that the building is in the wrong position, because it is always seen in the same place by different keyframes.

This highlights the behavior of the optimization with a pose graph without using g2o priors: because there are not measures directly associated with a node, each estimate of a node is computed with respect to the other nodes using the relative transformations of edges. If all keyframes see a building in a position, then the building will remain there, even if the position is wrong. The system may be able to move buildings but is not able to tell if a building is in a wrong or correct position.

We are also relying on the fact that the registration algorithm is correctly aligning the point clouds. If the registration fails and there significant differences between one keyframe and another, then we may get that more

keyframes see a building in different positions, but not because the building is wrongly positioned.

5

BUILDING SLAM WITH BUILDINGS PRIORS

The objective of hdl_graph_slam_with_map_priors, as explained in Chapter 4, is to estimate the poses of the robot given the information about buildings, taken from publicly available map services like OpenStreetMap. Buildings are inserted into the pose graph as $SE(2)$ nodes, as keyframe nodes are. After optimization, keyframe poses may have been changed following the constraints inside the pose graph. In the same way, also building poses may have changed.

Since the buildings are aligned with the keyframe LiDAR point cloud, a motion in the poses of buildings would make them to be more coherent with the LiDAR point cloud.

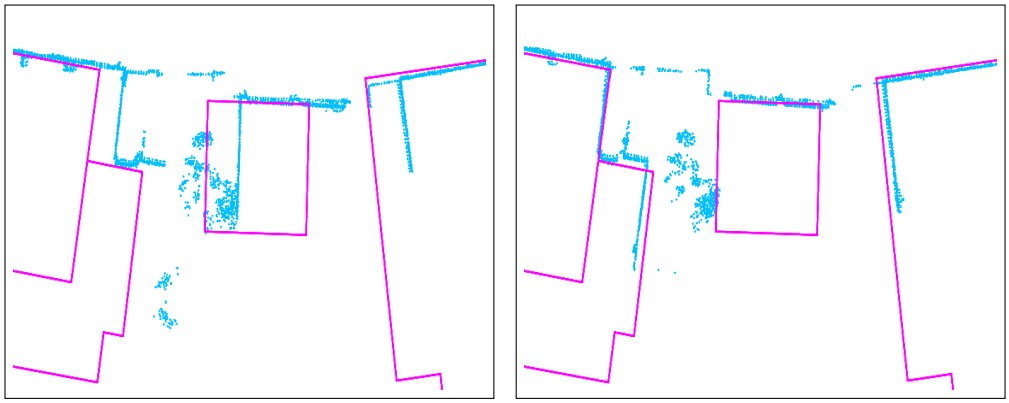
In this way, the position of buildings that originally were wrongly positioned in OpenStreetMap is now corrected. The robot obtains a map which is more coherent with the real environment, facilitating the correction of the drifting of the robot. In general, a better map eases tasks in which precision is required, such as autonomous driving. The system may be even used not for the specific task of navigating a mobile robot in the environment, but with the focus on solely correcting the map and the buildings. The advantage is that, instead of re-mapping every single building with a GPS, it is enough to send out a mobile robot equipped with a LiDAR and possibly, but not mandatory, a GPS on the main streets, and automatically all the buildings which can be seen should be eventually corrected.

We are assuming that the LiDAR point cloud is correct. Like any other sensor, also a LiDAR suffers from sensor noise and other errors. LiDARs particularly suffer from occlusions, which make parts of the environment not observable, and a LiDAR measurement is influenced by the ego-motion of the mobile platform on which it is mounted, causing distortions in the point clouds acquired.

OpenStreetMap is a collaborative and open source project carried out by different people with different tools. This means that it is prone to human errors, errors due to the accuracy of tools (GPS accuracy), and errors due to the redundancy of data (the same place mapped by different people with different tools and the consequent merging of information).

Buildings may also have been demolished, newly built, or changed in shape and OpenStreetMap may not be updated. The building SLAM is robust to these kinds of issues and may be useful to identify them.

As explained in Section 4.6, hdl_graph_slam_with_map_priors already achieves a correction of the positions of buildings that is not perfect. In the



(a) (b)

Figure 5.1: Wrong NDT-OMP alignment and correction. (a) The LiDAR scan (light blue) is wrongly aligned with the buildings (pink), because of an error of the NDT-OMP algorithm; (b) the correct alignment has been recovered.

following sections, we describe two methods that have been followed to finely tune the eventual motion of the position of buildings. We call the resulting systems building_slam_with_building_priors Rigid SLAM and building-_slam_with_building_priors Non-rigid SLAM. Rigid SLAM is like hdl_graph-_slam_with_map_priors described in Chapter 4, but the computation of information matrices has been improved to better reflect how much the alignment is fitting for each building. Non-rigid SLAM modifies the way the alignment is done to also get better transformations, other than better information matrices, by performing local alignments with each building, independently one from the other.

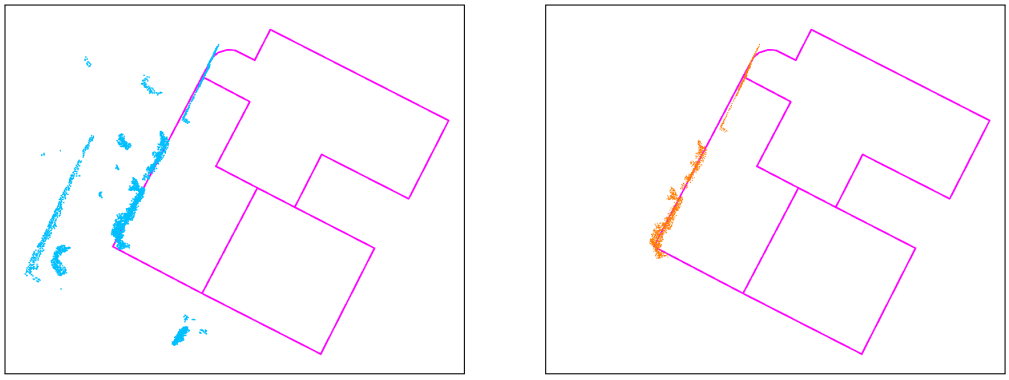
5.1 RIGID SLAM

In hdl_graph_slam_with_map_priors, the information matrices associated to buildings constraints are computed by evaluating the quality of the alignment between the LiDAR point cloud and the buildings point cloud, as explained in Subsection 4.5.5. We call the alignment between the LiDAR scan and the buildings point cloud, which contains all the buildings surrounding the keyframe, *global alignment*, and the corresponding transformation, *global transformation*. The naming derives from the fact that the alignment is done with respect to the buildings as a whole, unified map. Given a keyframe, the information matrix is computed one time and it is the same for all the building constraints associated with that keyframe. Because the global alignment may not be precise for all the buildings, computing the same information matrix for all of them is not advantageous. A possible cause of the global alignment not being precise may be due to NDT-OMP, which encounters difficulties

Figure 5.2: Misplaced building and corresponding NDT-OMP alignment. The LiDAR scan (light blue) is correctly aligned with the buildings (pink), as can be seen from the building on the right. The buildings on the left are clearly wrongly positioned.

in getting a correct global alignment, as shown in Figure 5.1. In this case, the global alignment is not precise in the same way for all the buildings, although there may still be buildings aligned a little bit better than others. Buildings may also have been demolished or newly built, but OpenStreetMap may have not been updated about it. Another possible cause may be due to imprecision in the positioning of the buildings in OpenStreetMap, as represented in Figure 5.2. Even if the NDT-OMP is correctly aligning, as it can be seen from the building on the right, on the left, there is still a mismatch between the LiDAR point cloud and the buildings point cloud, which is evidently caused by the wrong positions of those buildings. Using the same information matrices for edges linked to the buildings on the left and on the right means that we are assuming the alignment is of the same quality, for all of them. This implies that all the buildings edges linked with the same pose of the robot have the same weight in the optimization. Instead, a building associated to different poses of the robot, may have a different information matrix for each edge, thus a different weight in the optimization.

The information matrices are linked to the quality of the alignment through the notion of fitness score. Remember that information matrices are based on the fitness score, as explained in Subsection 3.3.9, which is a measure of how good the transformation estimated by the alignment algorithm is. In our case, the transformation is good, and the fitness score is low if the aligned LiDAR point cloud (light blue in Figure 5.2) maximally overlaps with the building points cloud (pink in Figure 5.2), and vice versa. If the fitness score is low, we already know the information matrix has high values and vice versa. So, if



(a) (b)

Figure 5.3: Example of correspondences computed for a building. (a) In light blue the entire LiDAR point cloud and in pink the buildings point cloud; (b) in orange the points of the LiDAR point cloud that are part of the correspondences used to compute the fitness score, for the leftmost building.

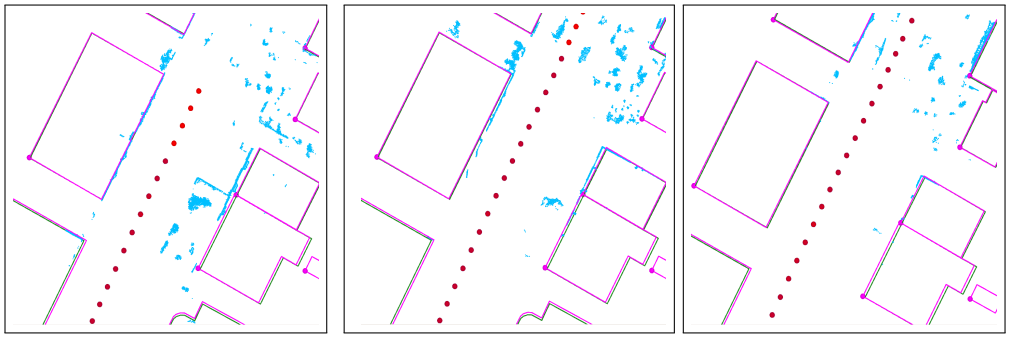
the information matrix has high values, it means the quality of the alignment is good, and vice versa.

From figure 5.2, we can clearly see that, in reality, for the building on the right, the quality of the alignment is really good, while for the buildings on the left, the quality is bad, because the error is in the map and not in the keyframe point cloud. Consequently, we would like to give to the edges associated with the buildings on the left a low values information matrix, such that their weight in the optimization is low. On the other hand, we would like to give to the edge associated with the building on the right a high values information matrix, such that it will more important in the optimization.

By doing as explained, we are trying to keep the correctly positioned buildings in their position, while giving the wrongly positioned ones more freedom to move (see Subsection 3.3.9 for more details). Then, buildings will be moved following the mechanism explained in Subsection 4.6.2.

Indeed, a possible way of improving the positioning of buildings lies in a better calculation of the information matrix. In particular, a different information matrix is computed for each single building constraint associated with a keyframe.

The computation is still based on the fitness score. The only difference is that, instead of computing it using all the correspondences between the LiDAR point cloud and the buildings point cloud (thus computing the *global fitness score*), for each building we use just the correspondences associated with it (the *local fitness score*). First, the LiDAR point cloud is transformed via the global transformation estimated by NDT-OMP, so that in this way it maximally overlaps with the buildings point cloud. Then we retrieve all



(a) (b) (c)

Figure 5.4: Results from Rigid SLAM. (a) (b) (c) in green the buildings as downloaded from OpenStreetMap, in pink the buildings as estimated by the optimization, in light blue the LiDAR scan. Pink dots are the buildings nodes, red dots are keyframe nodes. In the progression of images, we can see how the Rigid SLAM progressively moves buildings to align correctly to the LiDAR scan of the "corridor", although there is still imprecision.

the correspondences that would be used to compute the *global fitness score* as a vector. For each point in the transformed LiDAR point cloud, the closest point in the buildings point cloud is found: the pair formed by these points is a *correspondence*. Correspondences which distance between elements is higher than a user-defined threshold are discarded. Then, we cycle over buildings for which we need to compute an edge, and for each of them, we extract, from the vector of correspondences, only the ones related to the building, by comparison with *building.geometry*. Once obtained the building-specific correspondences, we compute the fitness score in the usual way: for each pair of points, we compute the Euclidean distance between them, and, at the end, we average all the calculated distances. This average is the local fitness score. Since the correspondences are tied to a single building, as it can be seen from Figure 5.3, the resulting fitness score is representative of how much the global alignment is good for that single building.

Figure 5.4 shows the results of Rigid SLAM on a "corridor"-like environment. The LiDAR point cloud is in light blue, the green buildings are the fixed buildings as downloaded from OpenStreetMap and the pink buildings are the ones after optimization. We see that going from (a) to (b) to (c), the buildings on the right move to the left in order to satisfy the LiDAR scan. There is still a lot of imprecision, for example, the buildings on the left are moved to the left, while there is no need for it. The building on the right, although moved towards the correct position, still not matches perfectly the LiDAR scan.

Information matrices customized for each building constraint lead to an improvement on the eventual correction of the positions of buildings,

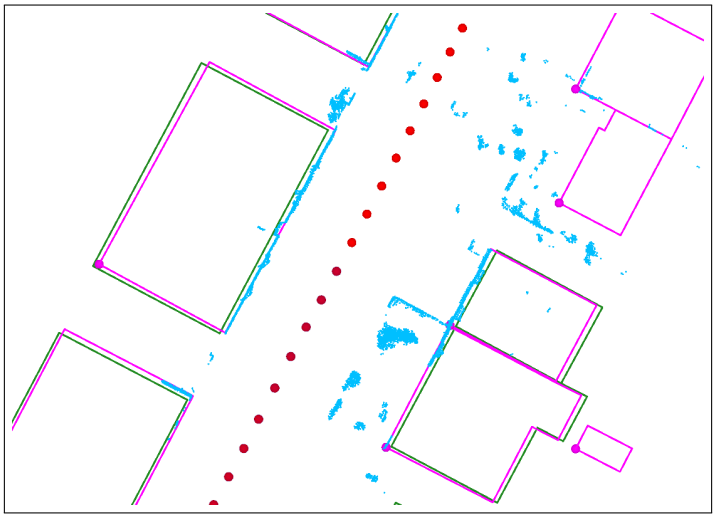


Figure 5.5: Results from Non-rigid SLAM. In green the buildings as downloaded from OpenStreetMap, in pink the buildings as estimated by the optimization, in light blue the LiDAR scan. Pink dots are the buildings nodes, red dots are keyframe nodes. We can see how the pink buildings perfectly coincide with the LiDAR scan.

since now each edge is weighted in a more appropriate manner than in hdl_graph_slam_with_map_priors, although there are still inaccuracies.

5.2 NON-RIGID SLAM

Both hdl_graph_slam_with_map_priors and Rigid SLAM suffers from the problems exposed in Section 4.6.2, about the fact that a building is corrected only when more keyframes see it in different positions. The building_slam-_with_buildings_priors Non-rigid SLAM system, here explained, implements a solution to that kind of problems.

Instead of using a global alignment and a local fitness score to compute information matrices, we use a local alignment and a local fitness score. *local alignment* means that we align the entire LiDAR point cloud to the point cloud representing a single building. The fitness score is computed for each local alignment, so, given a keyframe, there is a different information matrix for each edge linking it to the buildings surrounding the keyframe, as explained in the previous subsection. However, since the alignment is local, the fitness score is not anymore the measure of how much the global transformation fits that single building, but it is the measure of how much the local transformation fits the single building.

First, we compute the global alignment as usual, since it is used as an initial guess for the local alignment. Then, we cycle all over the buildings for which we need to insert an edge with the current keyframe: for each building we perform the local alignment, compute the information matrix and insert

the edge, using a relative transformation computed starting from the local alignment transformation, as explained in Subsection 4.5.4.

In this way, a building constraint directly tells the correct position of the building, without the need for abrupt jumps in the alignment and without the need for a building to be seen in different positions from different keyframes. Nevertheless, the phenomena for which a building may be seen in different positions from different keyframes is still present, although its negative sides are not anymore a problem. If it happens, it will be treated as usual.

Since we are not relying anymore on the global alignment but on local matchings, each building may move independently from the others: that is the reason for the name "Non-rigid SLAM".

In Figure 5.5 we can see the results from the Non-rigid SLAM on a "corridor"-like environment. Also here, the LiDAR point cloud is in light blue, the green buildings are the fixed buildings as downloaded from OpenStreetMap and the pink buildings are the ones after optimization. Now the pink buildings perfectly coincide with the LiDAR scan, thus obtaining the desired behavior of moving the buildings so that they are coherent with the LiDAR scan, and thus with the environment.

6

RESULTS

In this chapter, we present the results of running the proposed systems with a set of pre-collected data, namely a ROS bag. We evaluate the performance of the systems through two types of tests: numeric errors on the final pose of the robot and a visual evaluation of the correction of wrongly positioned buildings. To check the accuracy of our work, we use typical metrics for benchmarking SLAM systems, explained in Subsection 6.1.2 and Subsection 6.1.3: ATE (Absolute Trajectory Error) which is the error on the final pose of the robot with respect to the ground truth, and RPE (Relative Pose Error) which measures the error on the relative transformation between two consecutive poses of the robot with respect to the ground truth. By providing these errors, which use is widespread in the SLAM community, we provide numeric values which can be easily used for comparison with different algorithms and systems.

The evaluation on the correction of the position of buildings is done visually, by comparing the visual representation of point clouds and trajectories offered by rviz. In particular, we check how much the internal map of the robot is aligned with the point cloud of estimated buildings. rviz is a 3D visualization tool for ROS [53], which allows to visualize various types of data, such as reference frames, markers and point clouds. We do a visual evaluation since we do not have the ground truth on buildings.

The data used comes from the KITTI dataset [50]. This dataset is recorded from a moving vehicle collecting data from different sensors, such as high-resolution color and grayscale stereo cameras, a Velodyne HDL-64e laser scanner, and a GPS/IMU inertial navigation system. Data is gathered from different scenarios, such as countryside, city or campus. We focus on the urban context, as it is strictly correlated to the availability of 2D maps of the environment.

In the previous chapters, we have presented three different types of systems: the basic hdl_graph_slam_with_map_priors, the building_slam_with_building-_priors Rigid SLAM, and the building_slam_with_building_priors Non-rigid SLAM.

We run the hdl_graph_slam_with_map_priors without buildings to account for results for the basic hdl_graph_slam, because they represent the same system and our proposal also includes scripts for error computation, to immediately evaluate the results. We compare data from hdl_graph_slam, hdl_graph_slam_with_map_priors and building_slam_with_building_priors, highlighting the differences between the proposed systems. Numerical data

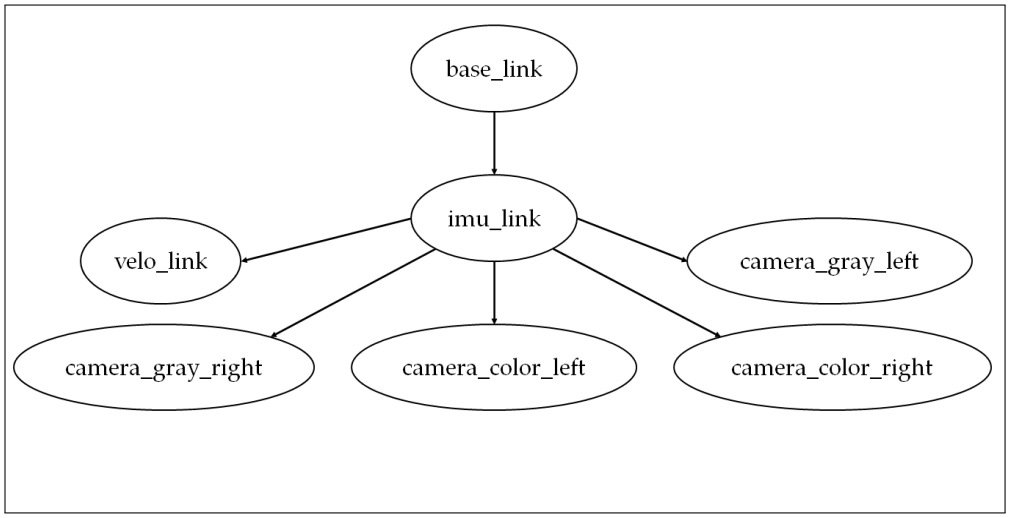


Figure 6.1: Transform tree of a KITTI bag

about building_slam_with_buildings_priors Rigid SLAM are not provided since it is just an intermediate system towards the Non-rigid SLAM. Only the visual evaluation is provided for it.

6.1 BACKGROUND

In the following subsections, we describe the types of data that are used for the evaluation and the types of errors computed, to better understand our experiments and evaluation.

6.1.1 *KITTI*

The KITTI dataset has been recorded from a moving platform while driving in Karlsruhe, Germany [50]. It includes camera images, laser scans, high-precision GPS measurements, and IMU accelerations from a combined GPS/IMU system. The dataset is mainly used in computer vision and robotic algorithms targeted to autonomous driving. The sensors from which data are provided are 2 PointGray Flea2 grayscale cameras, 2 PointGray Flea2 color cameras, 4 Edmund Optics lenses, one Velodyne HDL-64e rotating 3D laser scanner, and an OXTS RT3003 inertial GPS navigation system. Sequences are provided in the categories "Road", "City", "Residential", "Campus" and "Person". For each sequence, raw data, 3D bounding box tracklets, and a calibration file are provided. Data has been calibrated, synchronized, and timestamped. Data from cameras are provided as .png images with a .txt file containing timestamps. Data from OXTS are stored in a .txt file from each frame, plus another .txt containing timestamps. Scans from Velodyne are

Figure 6.2: Ground truth and OpenStreetMap buildings. A square represent a 10×10 m^2 area. The trajectory is distributed on a total area of approximately 230×180 m^2, for a total length, estimated by the scan matching, of 687.116 m.

provided as floating-point binaries, plus a .txt file for timestamps. For some sequences, the ground truth is provided as a .txt file.

To convert these data into a ROS bag, to be easily integrated with the ROS system, we use a tool called *kitti2bag* [51]. This tool converts KITTI data into ROS bag and uses the KITTI calibration files to build the transform tree in Figure 6.1. *imu_link* is the reference frame of the GPS and IMU messages, *velo_link* is the reference frame of the Velodyne scans, and *camera_..._..* are the reference frames of the various camera images. The *base_link* is assumed to be coincident with the *camera_gray_left* KITTI reference frame, but on the ground level, thus with height equal to zero.

GPS messages are of type *sensor_msgs/NavSatFix*, published on the topic */kitti/oxts/gps/fix*. IMU messages are of type *sensor_msgs/Imu*, published on the topic */kitti/oxts/imu*. Velodyne scan messages are of type *sensor_msgs/PointCloud2*, published on the topic */kitti/velo/pointcloud*. Image messages are of type *sensor_msgs/Image*, published on the topic */kitti/camera_..._../image_raw*. The examples presented here are done with the sequence *2011_09_30_drive_0027* from the category "residential". It has a length of 1112 frames for a total of 01:51 minutes. It has a unique loop closure near the end of the bag. Since it is a residential zone, there are buildings all over the trajectory. The shape

of the trajectory can be seen in Figure 6.2, where the light green dots are the ground truth and the green buildings are the buildings from OpenStreetMap.

6.1.2 *RPE*

As already stated, the RPE, *Relative Pose Error*, measures the local accuracy of the trajectory between consecutive robot poses [52].

Given two consecutive robot poses, which are both $SE(3)$ transformations with respect to a global frame of coordinates, namely T_{i-1} and T_i, it computes the relative pose transformation between them, $T_{x\_rel\_i}$, as:

$$T_{x\_rel\_i} = (T_{i-1})^{-1}(T_i) \tag{6.1}$$

In the same way, it computes $T_{gt\_rel\_i}$ for each couple of consecutive ground truth poses. Given $T_{x\_rel\_i}$ and $T_{gt\_rel\_i}$ corresponding to the same couple of poses in the sequence, we compute the relative pose between them, as:

$$T_{delta\_i} = (T_{gt\_rel\_i})^{-1}(T_{x\_rel\_i}) \tag{6.2}$$

T_{delta_i} indicates how much the robot pose relative transformation differs from the ground truth one. High values in its translational and/or rotational components correspond to a high error between the pair of poses considered. The RPE is divided into two parts: translational and rotational. The translational RPE is computed as:

$$t\_RPE = \frac{1}{n}\sum_{i=1}^{n}(trans(T_{delta\_i})) \tag{6.3}$$

where $trans(T_{delta\_i})$ is the translational component of T_{delta_i} and n is the number of couples of robot poses in the sequence. The rotational RPE is computed as:

$$r\_RPE = \frac{1}{n}\sum_{i=1}^{n}(rot(T_{delta\_i})) \tag{6.4}$$

where $rot(T_{delta\_i})$ is the rotational component of T_{delta_i}.

6.1.3 *ATE*

The ATE, *Absolute Trajectory Error*, measures the global consistency of the estimated trajectory, by comparing the absolute poses between the estimated and ground truth trajectories [52].

Generally, the ground truth is not aligned in the same way as the estimated trajectory, thus, before being used to compute ATE, it needs to be aligned in the same reference frame as the estimated trajectory. KITTI provides the

ground truth in *camera_gray_left* reference frame, thus we need to convert it to *base_link* in order to be compared with the estimated trajectory. This can be done easily by retrieving the $T_{base\_link \to camera\_gray\_left}$ from the ROS network and applying it to the ground truth poses.

The KITTI ground truth poses are referred to the first one, in the same way as the estimated trajectory is referred to zero_utm, thus the origin of the estimated trajectory and the origin of the ground truth trajectory referred to the first pose, coincide.

The original orientation of the reference frame of the ground truth is not provided, thus we need to manually orient the ground truth in order to overlap with the estimated trajectory. To do this we use a mechanism that takes the last estimated robot pose, takes the corresponding ground truth pose, and computes the difference in angle between them (with respect to the origin). This angle is used to compute a specific transformation that, applied to the ground truth poses, makes them align with the estimated trajectory. In the following, we assume the ground truth has been correctly aligned.

Given a robot pose T_{x_i} and the corresponding ground truth pose T_{gt_i} in the sequence, we compute the relative transformation between them as:

$$T_{delta\_i} = (T_{gt\_i})^{-1} T_{x\_i} \tag{6.5}$$

The ATE is computed only on the translational component, and it is:

$$ATE = \frac{1}{m} \sum_{i=1}^{m} (trans(T_{delta\_i})) \tag{6.6}$$

where $trans(T_{delta\_i})$ is the translational component of T_{delta_i} and m is the number of poses in the sequence.

6.2 RESULTS

In the following subsections, we present the results. The first subsection presents the numerical error data. The second subsection presents a visual comparison of the alignment between the map point cloud and the estimated buildings.

6.2.1 *Parameters*

All the parameters can be set in the ROS launch file called *kitti.launch*. The parameters for the computation of the information matrices of the buildings edges in all cases are $min\_var_x = 0.01$ m, $max\_var_x = 1$ m, $min\_var_q = 0.05$ m and $max\_var_q = 0.2$ m. All of the cases use FAST GICP for the scan matching.

	ATE [m ± m]	t_RPE [m]	r_RPE [rad]
hdl	2.05932 ± 1.12551	1.69395	0.23148
hdl_odom_gps_imu	1.19747 ± 0.46963	1.70285	0.18073
hdl_map_priors	1.54614 ± 1.14073	1.70185	0.18350
buildings_priors _non_rigid	1.86219 ± 1.07568	1.70273	0.18031

Table 6.1: ATE and RPE errors

hdl_graph_slam_with_map_priors uses NDT-OMP for the alignment between LiDAR point clouds and buildings clouds. All the parameters for NDT-OMP are set to default apart from $\texttt{ndt\_reg\_resolution} = 2.0$ m. The fitness score threshold for the computation of the information matrices in this case is $\texttt{b\_fitness\_score\_thresh} = 2.5$.

building_slam_with_buildings_priors Non-rigid SLAM uses NDT-OMP for the global alignment and GICP-OMP for the local one. This setup is due to the fact that GICP-OMP can be tuned to be really precise (see Section 7.1 for a more in-depth explanation). All the parameters for these algorithms are left to default values apart from $\texttt{ndt\_reg\_resolution} = 2.5$ m and $\texttt{gicp\_max\_correspondence\_distance} = 0.5$ m. The fitness score threshold for the computation of the information matrices in this case is $\texttt{b\_fitness\_score\_thresh} = 0.5$.

6.2.2 *Errors*

We present the numerical errors, namely ATE and RPE, computed for various scenarios. The cases presented are:

- hdl_graph_slam with only odometry and loop closure (*hdl*) and hdl_graph_slam with odometry, loop closure, GPS and IMU (*hdl_gps_imu*), as seen in Chapter 3
- hdl_graph_slam_with_map_priors (*hdl_map_priors*), without GPS and IMU, as explained in Chapter 4
- building_slam_with_buildings_priors (*building_priors_non_rigid*) non rigid SLAM, without GPS and IMU, as explained in Chapter 5

Numerical data about building_slam_with_buildings_priors Rigid SLAM are not provided since it is just an intermediate system towards the Non-rigid SLAM. The results are presented in Table 6.1. The values for ATE and t_RPE are expressed in meters, while r_RPE is expressed in radians. ATE is expressed as $\texttt{mean} \pm \texttt{standard\_deviation}$ (confidence interval at 68%). The

Figure 6.3: Resulting buildings, trajectory and ground truth of four selected cases. Light green dots are the ground truth poses; red dots are the estimated robot poses; in pink the estimated buildings; in dark green the buildings from OpenStreetMap; (a) *hdl*; (b) *hdl_gps_imu*; (c) *hdl_map_priors*; (d) *building_priors_non_rigid*.

four cases here proposed can be seen in Figure 6.3. The green dots are the ground truth poses, while the red dots are the estimated final poses of the robot. Note that we printed all the ground truth poses, even the ones that do not have a corresponding keyframe. In all four cases, the loop closure at the end has been correctly detected.

Before starting the evaluation of the results, let's examine the sequence. The sequence proposed is a quite linear and simple trajectory, without abrupt changes and particular geometries. Additionally, all along the path, there are distinctive features, the buildings themselves. Because of these characteristics, the scan matching, which is already good, works even well, and the estimated odometry does not drift much.

The errors are low and quite similar one to the other. The t_RPE is about 1.7 m for all of them and the r_RPE is about 0.18 m, apart for the *hdl* case in which is about 0.23 m. Given the length of the trajectory (687.116 m), these values are good because they are low. They tell us that, over time, there is not a significant local drift. Additionally, by comparing the different cases, we see that the local accuracy cannot be improved that much, at least with the techniques proposed. Note how, because of the IMU or because of the buildings, the r_RPE of all the other cases is slightly reduced with respect to *hdl*.

A little bit more variability is present in the ATE, although it is still low. Remember, from Subsection 2.5.1, that the two experiments by Vysotska and Stachniss [9], run without map priors, yielded errors on the final poses of 5 m and 22 m, while with map priors, they measured 1 m and 0.5 m, respectively. An ATE in the range $1.19 - 2.05$ m, with or without map priors, is definitely more accurate, even as result of the base system. The lowest error is the case of hdl_graph_slam with GPS and IMU because the accuracy given by the GPS positions greatly contributes to the estimation of the trajectory of the robot. *hdl_map_priors* and *building_priors_non_rigid* errors are higher than *hdl_gps_imu*, but still manage to remain under the values associated to *hdl*. The ATE from *building_priors_non_rigid* is higher than *hdl_map_priors* because the former is still not perfectly tuned. Thus, while generally the buildings are moved in the correct position, there are some unwanted motions of the buildings (see Section 7.1 for more details). Being that *hdl_map_priors* do not move buildings that much, the NDT-OMP alignment is a bit more stable so the ATE is low.

With a better tuning of the registration algorithm between the LiDAR scan and buildings and better tuning of the information matrices parameters, more precise results may be obtained. It is worth noticing that, by using buildings priors only, we obtain results comparable with the GPS/IMU ones. This is a great advantage since, for buildings priors, the system does not need any additional equipment, i.e., the GPS/IMU unit, which is expensive and energy-consuming, but only a LiDAR, which is already included because it

is used for scan matching, and an internet connection to download buildings (the system is built to directly download data about the buildings from OpenStreetMap, but it can be easily adapted to work with offline maps).

From Figure 6.3a we can see how the odometry does not represent, in reality, a correct geographic position. Thus, when introducing the GPS or the buildings, as in Figures 6.3b, 6.3c and 6.3d, the trajectory and the ground truth are aligned with the geographic map. It is worth noticing that, before the loop closure in the end, the *hdl* case was slightly drifting, with an ATE of about 3.40 m, and then, because of the loop closure, the error decreases.

6.2.3 *Visual Evaluation*

We provide a visual evaluation of how the estimated buildings are aligned to the internal map of the robot estimated by SLAM. This map is built by merging together all the point clouds of the keyframes. We discuss three cases: hdl_graph_slam_with_map_priors, referred to as *hdl_map_priors*, visible in Figure 6.4a; building_slam_with_buildings_priors Rigid SLAM referred to as *building_priors_rigid*, visible in Figure 6.4b and building_slam_with-_buildings_priors Non-rigid SLAM, referred to as *building_priors_non_rigid*, visible in Figure 6.5. In these figures and in the following, we make a distinction between the OpenStreetMap buildings (dark green) and the buildings estimated by the system (pink). OpenStreetMap buildings are fixed and do not move, they just represent the buildings as they were downloaded. Estimated buildings may change position and orientation if the optimization decides to change the pose of the corresponding node in the pose graph.

Figures 6.7a, 6.7b and 6.7c present a detail from the whole trajectory, showing a "corridor"-like environment, already seen in Figure 5.5. As we can see, there is a discrepancy between the map (black) and the OpenStreet-Map buildings (dark green), while in pink we have the estimated buildings. The corridor from the map is narrower than the one formed by the OSM buildings. Assuming the map is correct, this means that the buildings are wrongly positioned.

Figure 6.7a is the *hdl_map_priors* case. There is just a small sign of improvement. We can see that the right lower building is slightly moved toward the scan, but not enough to perfectly coincide with it. The right upper building is not moved, while the left building is moved toward the left, further worsening the situation.

Figure 6.7b is the *building_priors_rigid* case. With respect to the previous case, there is a remarkable refinement. The right buildings are both moved toward the left and both nearly coincident with the map. The right building is kept still. Thus, with Rigid SLAM, we get a not perfect but good alignment. Figure 6.7c is the *building_priors_non_rigid* case. The correction achieved here is the best one among the three. The two right buildings are moved to the left

(a)

(b)

Figure 6.4: (a) *hdl_map_priors*; (b) *building_priors_rigid*. Light green dots are the ground truth poses; red dots are the estimated robot poses; in pink the estimated buildings; in dark green the OpenStreetMap buildings; in black the map.

Figure 6.5: *building_priors_non_rigid*. Light green dots are the ground truth poses; red dots are the estimated robot poses; in pink the estimated buildings; in dark green the OpenStreetMap buildings; in black the map.

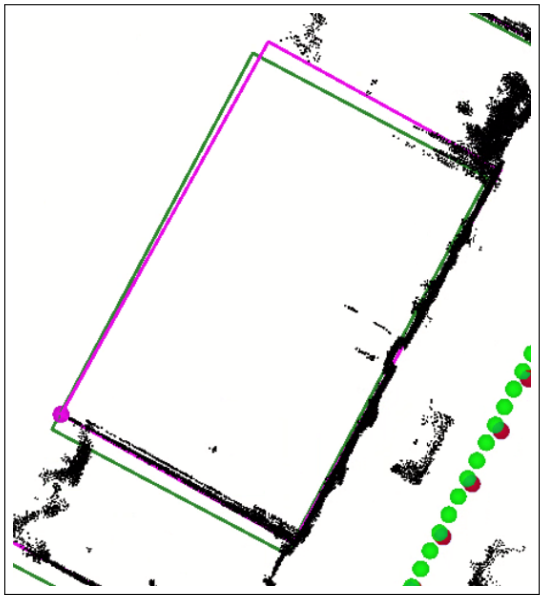


Figure 6.6: Detail about the orientation of a building in *building_priors_non_rigid*

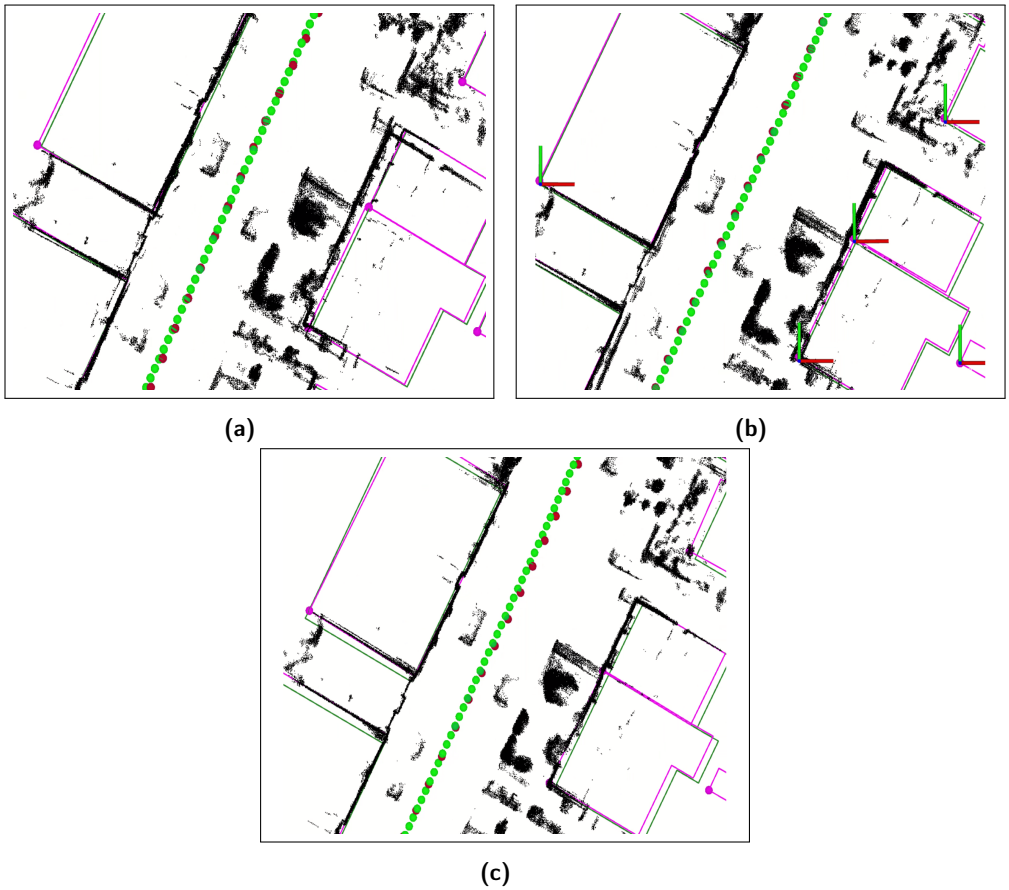


Figure 6.7: "corridor" details on three different cases. Details about a "corridor"-like environment; light green dots are the ground truth poses; red dots are the estimated robot poses; in pink the estimated buildings; in dark green the buildings from OpenStreetMap; in black the map estimated by the robot; (a) *hdl_map_priors*; (b) *building_priors_rigid*; (c) *building_priors_non_rigid*.

and perfectly aligned with the map. The building on the left is just moved upward but still coincides with the scan. Note that, of the building on the left, also the orientation has been corrected with respect to its original position, represented in Figure 6.6. From the previous subsection, we saw that the ATE for *building_priors_non_rigid* is higher than the one for *hdl_map_priors*, altough the positioning of building is better in the *building_priors_non_rigid* case. This happens because *building_priors_non_rigid* is still not tuned perfectly, thus there may be some imprecision in the correction of buildings that makes its ATE to be higher (see section 7.1 for more details).

Figures 6.8a, 6.8b and 6.8c show details of four buildings. Figure 6.8a presents the situation for *hdl_map_priors*: no buildings have been moved. The center buildings are clearly aligned with the map on the right and upper sides, while the upper buildings are misaligned with the scan, thus at least

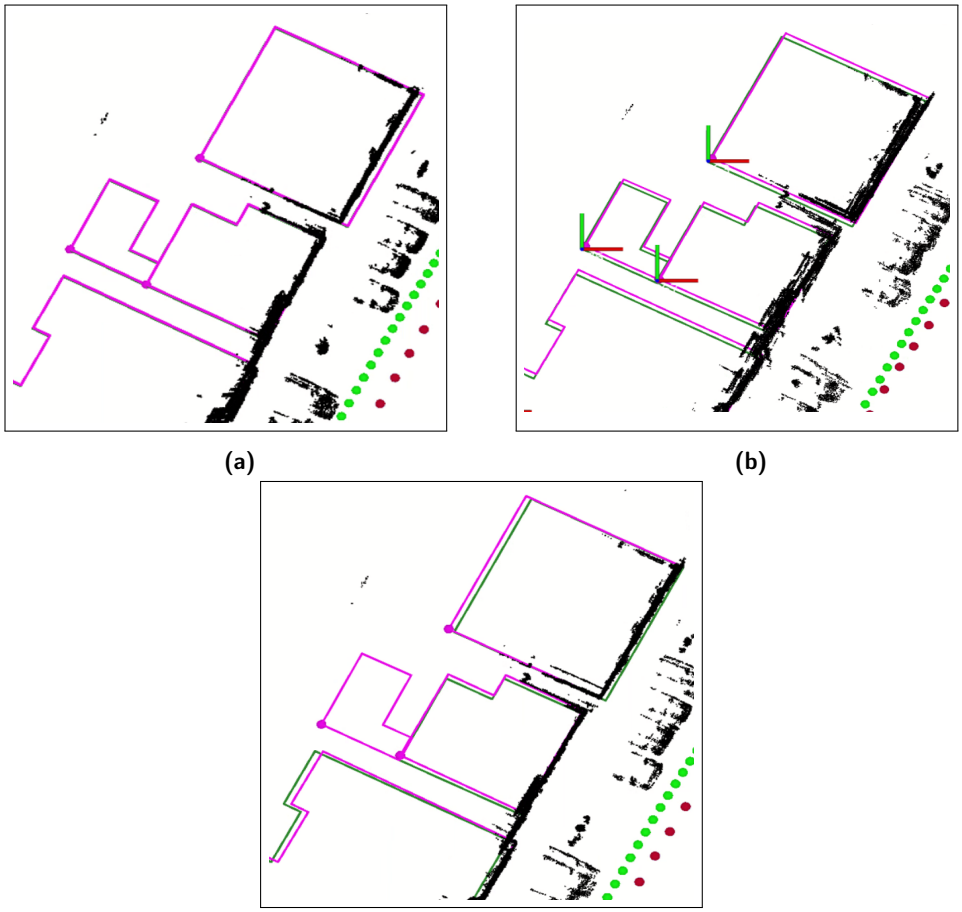


Figure 6.8: Details about 4 buildings on three different cases. Details about a 4 buildings; light green dots are the ground truth poses; red dots are the estimated robot poses; in pink the estimated buildings; in dark green the buildings from OpenStreetMap; in black the map estimated by the robot; (a) *hdl_map_priors*; (b) *building_priors_rigid*; (c) *building_priors_non_rigid*.

one of them is wrongly aligned. This was expected, as hdl_graph_slam_with-_map_priors is not able to do a correction.

In Figure 6.8b, we see the result from building_slam_with_buildings_priors Rigid SLAM. Buildings are all displaced of the same amount, corresponding to a common rigid roto-translation. This is coherent with the fact that just a global alignment is used. Note also how the map is much more cluttered with respect to Figures 6.8a and 6.8c. This is due to the highly variable alignments done by NDT-OMP, which make the point clouds constituting the map not to overlap perfectly. In Figure 6.8c, we see the result from building_slam_with_buildings_priors Non-rigid SLAM. Here the map is clear and the buildings are perfectly aligned with it: the upper building has

been moved to the left, the center building has been moved slightly upward and the lower one has been moved to the right. These last two considerations are not visible from Figures 6.8a and 6.8b, where there is so much clutter in the map that the lower and center buildings seem to be clearly aligned, while in reality, as we can see from Figure 6.8c, they are not. The more precise map is actually due to the good positioning of buildings, for which the global alignment is less variable. If all the buildings are in a position coherent with the scans, NDT-OMP does not need to continuously adapt (and thus change) the alignment to try to maximally match wrongly positioned buildings.

Remarking what we have seen with these experiments, hdl_graph_slam-_with_map_priors is not really able to do the correction of buildings, due to its global nature, which is not able to solve local conflicts about misaligned buildings. Rigid SLAM works a little bit better, in the sense that wrongly positioned buildings are generally moved towards the correct position, but still not perfectly coherent with the map of the robot. Note how its map seems to be the more cluttered, due to the fact that the NDT-OMP alignment is highly variable. This is caused by the imprecise displacement of the buildings and the NDT-OMP tries to continuously improve its alignment without success, since, because of the buildings not perfectly positioned, the alignment itself will never be perfect.

This is also the reason of why the obtained map is more precise in the case of Non-rigid SLAM: since buildings coincide perfectly with the map, NDT-OMP does not need to continuously change the alignment to try to improve it, since it is already in the best case. The use of a local alignment allows for each single buildings to be precisely positioned with respect to the map. Non-rigid SLAM is able to obtain a good estimate on the final pose of the robot, while simultaneously correcting the positions of buildings badly positioned in OpenStreetMap.

We just presented a single example and some details of the trajectory. By looking closely at Figure 6.5, we see how, although there are still some inaccuracies, we are able to correct the final pose of the robot from drifting and simultaneously correct the poses of buildings to make them more coherent with the scans.

7

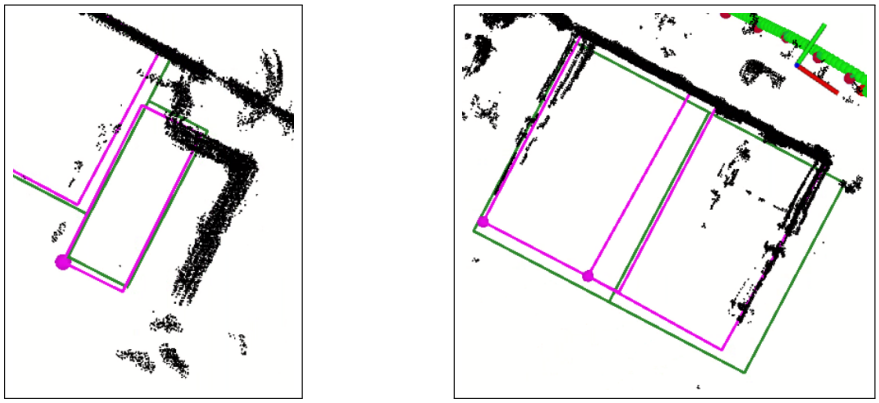
CONCLUSIONS

In this thesis, we introduced a new LiDAR Graph SLAM system, which is able to precisely localize the robot, and creates an accurate map of the environment, by using the information associated to buildings obtained from already available maps. The system is also able to correct the position and orientation of buildings that are misplaced into the map.

Three variants of the system were presented, each improving from the previous one. These are based on an existing LiDAR Graph SLAM system called hdl_graph_slam. It presents all the features of a typical Graph SLAM system, such as graph construction, data association, optimization, and loop closure. Since it based on a graph structure, it is easy to introduce new types of data and new types of constraints into the graph. The whole hdl_graph_slam has been described and in particular the component called hdl_graph_slam_nodelet, because it is the one that we modified to develop our work.

The first system we presented is called hdl_graph_slam_with_map_priors. It introduces a new type of constraint in the pose graph, based on the alignment between the map of buildings and the LiDAR scan. It is in charge of downloading the buildings, inserting them into the pose graph as nodes, and inserting new edges into the pose graph between robot poses nodes and buildings nodes, based on their alignment. From the results, we can see how it is able to correct the pose of the robot from drifting but not to correct the poses of the buildings.

The second proposed system is called building_slam_with_buildings_priors Rigid SLAM and it is the first tentative of improving hdl_graph_slam-_with_map_priors, in order to better correct the poses of the buildings. The computation of information matrices has been improved in order to be targeted to every single building, instead of being the same for all the buildings associated to a single pose of the robot. The results indicate that the Rigid SLAM version is able to correct the poses of the buildings, but it is not really precise. The third variant of the described system, building_slam-_with_buildings_priors Non-rigid SLAM, constitutes another improvement based on Rigid SLAM. The computation of information matrices for every single building is retained. What changes is that, instead of using a global alignment between all the buildings and the LiDAR scan, as it was done in the hdl_graph_slam_with_map_priors and in the Rigid SLAM version, a local alignment is used. This is an alignment between the LiDAR scan and every single building independently one from the other (so if we have N buildings,



(a) (b)

Figure 7.1: Errors in non rigid SLAM 1. (a) high misalignments are not corrected; (b) moved buildings may overlap.

there would be N local alignments). Thus, not only the information matrices are targeted towards the single building, but also the transformations used to build the edges to be inserted into the pose graph between a keyframe node and a building node. This allows to reach a good accuracy in the correction of the positions of the buildings, such that most of the time they are perfectly aligned with the LiDAR scans.

We evaluated the proposed systems on a real world scenario, proving that the system is able to correct the drifting on the final pose of the robot and that also the poses of buildings can be corrected.

The system may be adapted to work in different manners depending on the context. If we have really accurate poses of the robot (e.g., because we have a very precise GPS) then we may use the system to correct the poses of the buildings only. If we know the map is really accurate, then we may use the system to just correct the drift on the poses of the robot. Otherwise we can use the system to both correct the drift of the poses of the robot and the positions of buildings.

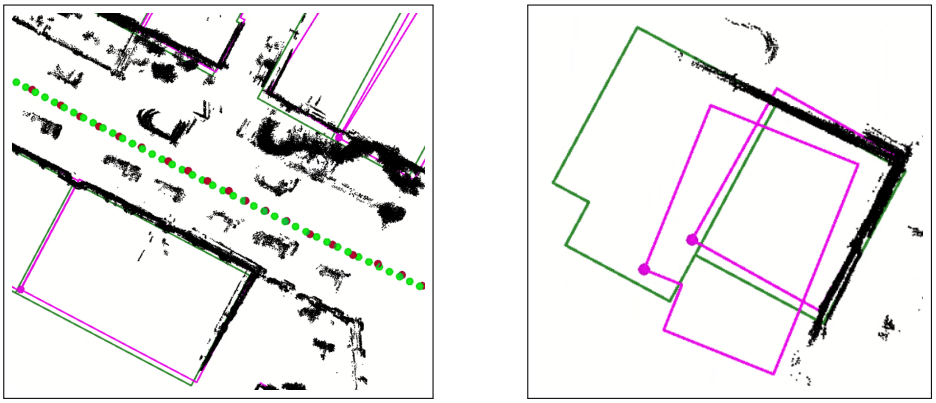
7.1 FUTURE IMPROVEMENTS

Here, we describe problems of which the Non-rigid SLAM version of the proposed system still suffers, along with possible solutions. A general improvement is to better tune parameters for better global and local alignments and for better computation of the information matrices. Being able to remove outliers and objects (e.g., cars and trees) from LiDAR scans, would allow for a better alignment too. This also would improve the pose tracking, leading to more accurate poses to start with, before the alignment with the OSM map. Other more specific problems are reported in Figures 7.1 and 7.2.

In Figure 7.1a we can see how objects that are highly misplaced are not corrected. The reason of this is the fact that the local alignment is tuned to be really precise. To perform an accurate local alignment, Non-rigid SLAM is using GICP-OMP with a really low *max_correspondence_distance* parameter. This parameter makes GICP-OMP discard all points correspondences between the LiDAR point cloud and the building point cloud whose distance is higher than the parameter value. Thus, if there are large displacements, the correspondences will be discarded and the alignment fails or it is really imprecise. A possible improvement would be to find a way to get high precision on the local alignment even for highly misplaced buildings. To obtain it, we may tune more precisely the GICP-OMP algorithm.

In Figure 7.1b we see two buildings correctly re-positioned to match the map. The problem is that now the two buildings geometrically overlap one with the other. A possible solution may be to introduce a new g2o prior or constraint in the system, which accounts for the relative pose between one building and the other so that this new information is directly inserted into optimization. This may hinder the optimization, since there would be a high number of edges. Another possible solution may be to keep the buildings as they are in 7.1b, but redefine their dimensions in order to not overlap.

In Figure 7.2a we can see how the buildings have been all more or less moved by the same amount upward, on both sides of the road. We are not able to discern if there is a real misplacement of all the buildings, or there is a general drift that affects all the buildings together. Remember, from Subsection 4.6.1, that the optimization has always to make a compromise between the robot poses and buildings. The buildings anchor the robot poses in order to reduce the drift. The robot poses may still drift a little, and thus, since buildings are linked with them, also the buildings may drift together. Alternatively, it may just be an error of the global alignment between the LiDAR point cloud and the buildings point cloud: if the alignment at a certain time is wrong, then it is not able anymore to recover the correct alignment, since buildings are moved to match the map created by the wrong alignment. In Figure 7.2b we see the effects of a problem of missing information. The leftmost building has been moved in a totally wrong position. This is due to the fact that the points of the map associated with such building are just a short line. The registration algorithm struggles to correctly associate a short line with an entire building, and the result is a bad positioning of the building. This happens frequently when there are a lot of buildings attached together, whose facades are all on the same line, thus the LiDAR just sees a straight line. The optimization is able to correct buildings whose facade is not in line with the others, but is not able to do something for the other sides of the buildings. It may even be that the registration moves buildings in a direction parallel to facades, without any reason, leading to a wrong local alignment. In general, if at least two sides of the building are seen by the



(a) (b)

Figure 7.2: Errors in non rigid SLAM 2. (a) all buildings drift together; (b) some buildings are difficult to locally align, since there is corresponding information into the map, but not enough.

LiDAR, then the registration is correct, otherwise, there may be ambiguities. To solve it we may perform the local alignment by using GICP-OMP but with a variable *max_correspondence_distance*: if there are a low number of points in the LiDAR scan, then increase the parameter so that the alignment is more coarse (because an accurate alignment may led to the wrong position since there are not enough points), otherwise, keep the parameter low to get a really precise alignment.

An example of the problem can be seen in Figure 7.3: the two buildings are positioned in line one after the other, and thus the LiDAR sees a straight line. The lower building has been moved upward, but on the map, there is no info about the other sides of the building, thus that is an arbitrary movement decided by the optimization. We do not know if it is correct or not.

Usually, we can distinguish if a building is misplaced because it is not coherent with the map, while the other buildings around are aligned. There may be some cases in which, even if there is enough information, the alignment and the subsequent motion of buildings are ambiguous. Take the corridor of Figures 6.7a, 6.7b and 6.7c. Assume there are only the upper building on the left and the two on the right, and nothing else around. It would not be possible to understand if it is the building on the left to be misplaced or the buildings on the right. Thus the decision of which buildings to move is up entirely to where the registration algorithm decides to align to. This is a rare situation, which happens only where there are few buildings. In general, without the ground truth on buildings, there is no way to understand if the buildings have been correctly positioned or not, even if they are coherent with the scan.

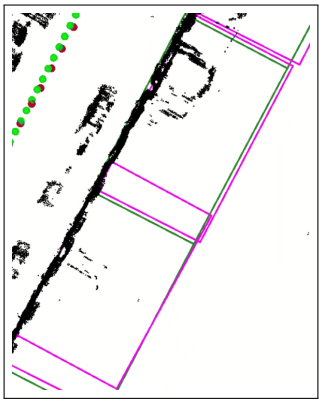


Figure 7.3: Wrongly positioned buildings in line

Another problem present in the computation of the local transformation is that having to do an alignment for each building slows down the performance of the system. Also here, the solution lies in a better tuning of the GICP-OMP algorithm or in the choice of a faster algorithm, such as FAST GICP.

Another improvement of the system is to separate the download of buildings from their parsing and insertion into the graph. If we are working with online maps and the internet connection is slow or the OSM server is congested or offline, then the system performance may be critically reduced because it waits for the response from the server. The optimization would be performed rarely, since, before it, the system has to wait for the download of buildings. A possible solution may be to implement a structure similar to the callback - flush queue structure of hdl_graph_slam described in Chapter 3: the download of buildings (performed in a sort of callback function) would be separated from the parsing and insertion into the graph (done the flush queue function). The buildings callback function would not be a real ROS callback function, because data about buildings are not read from a ROS topic, but the functionalities would be the same. The buildings callback function would be executed as a ROS timer (like the optimization callback) that periodically would check, eventually download and parse buildings for the newly inserted keyframes. These buildings would be inserted in a buildings queue. Then the optimization would call the buildings flush queue function, which would process the buildings queue, inserting them into the pose graph. We may even decide to create a new nodelet for the purpose of downloading and parsing buildings. The buildings would be published on a ROS topic with a custom message and subsequently read by the buildings callback function into hdl_graph_slam_nodelet.

BIBLIOGRAPHY

[1] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, and Wolfram Burgard. "A tutorial on graph-based SLAM." In: *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), pp. 31–43. ISSN: 1939-1390 (cit. on pp. 10, 12, 14, 40).

[2] OpenStreetMap contributors. *Planet.osm*. https://wiki.openstreetmap.org/wiki/Planet.osm (cit. on p. 20).

[3] OpenStreetMap contributors. *Overpass QL*. https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL (cit. on p. 21).

[4] OpenStreetMap contributors. *Overpass API/Language Guide*. https://wiki.openstreetmap.org/wiki/Overpass_API/Language_Guide (cit. on pp. 21, 22).

[5] OpenStreetMap contributors. *Elements*. https://wiki.openstreetmap.org/wiki/Elements (cit. on p. 19).

[6] OpenStreetMap contributors. *Map Features*. https://wiki.openstreetmap.org/wiki/Map_features (cit. on p. 20).

[7] OpenStreetMap contributors. *Overpass API*. https://wiki.openstreetmap.org/wiki/Overpass_API (cit. on p. 21).

[8] Wikipedia contributors. *OpenStreetMap — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/OpenStreetMap. [Online; accessed 17-June-2021]. 2021 (cit. on p. 19).

[9] Olga Vysotska and Cyrill Stachniss. "Improving SLAM by Exploiting Building Information from Publicly Available Maps and Localization Priors." In: *Photogrammetrie, Fernerkundung, Geoinformation* 85.1 (2017), pp. 53–65. ISSN: 14328364 (cit. on pp. 25, 27, 82).

[10] Kenji Koide. *hdl_graph_slam*. https://github.com/koide3/hdl_graph_slam (cit. on p. 29).

[11] OpenStreetMap contributors. *API*. https://wiki.openstreetmap.org/wiki/API (cit. on p. 20).

[12] OpenStreetMap contributors. *Downloading data*. https://wiki.openstreetmap.org/wiki/Downloading_data (cit. on pp. 20, 22).

[13] OpenStreetMap contributors. *OSM XML*. https://wiki.openstreetmap.org/wiki/OSM_XML (cit. on p. 22).

[14] Overpass API. *Output formats*. http://overpass-api.de/output_formats.html (cit. on pp. 21, 23).

[15] Rainer Kümmerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. "G2o: A general framework for graph optimization." In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3607–3613 (cit. on p. 17).

[16] Kenji Koide. *ndt_omp*. https://github.com/koide3/ndt_omp (cit. on p. 18).

[17] Kenji Koide. *fast_gicp*. https://github.com/SMRT-AIST/fast_gicp (cit. on p. 19).

[18] Kenji Koide, Masashi Yokozuka, Shuji Oishi, and Atsuhiko Banno. *Voxelized GICP for Fast and Accurate 3D Point Cloud Registration*. EasyChair Preprint no. 2703. EasyChair, 2020 (cit. on p. 19).

[19] Dirk Holz, Alexandru E. Ichim, Federico Tombari, Radu B. Rusu, and Sven Behnke. "Registration with the Point Cloud Library: A Modular Framework for Aligning in 3-D." In: *IEEE Robotics Automation Magazine* 22.4 (2015), pp. 110–124 (cit. on pp. 17, 18).

[20] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)." In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, 2011 (cit. on p. 18).

[21] Martin Magnusson. "The three-dimensional normal-distributions transform : an efficient representation for registration, surface analysis, and loop detection." In: 2009 (cit. on p. 18).

[22] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006 (cit. on pp. 10, 11, 32).

[23] Kenji Koide, Jun Miura, and Emanuele Menegatti. "A portable three-dimensional LIDAR-based system for long-term and wide-area people behavior measurement." In: *International Journal of Advanced Robotic Systems* 16 (Feb. 2019) (cit. on pp. 32, 38–40).

[24] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. "ORB-SLAM: a Versatile and Accurate Monocular SLAM System." In: *CoRR* abs/1502.00956 (2015). arXiv: 1502.00956 (cit. on p. 34).

[25] Henrik Kretzschmar and Cyrill Stachniss. "Information-theoretic compression of pose graphs for laser-based SLAM." In: *The International Journal of Robotics Research* 31.11 (2012), pp. 1219–1230. eprint: https://doi.org/10.1177/0278364912455072 (cit. on p. 34).

[26] Wim Meeussen. *REP 105 – Coordinate Frames for Mobile Platforms*. https://www.ros.org/reps/rep-0105.html (cit. on p. 35).

[27] Wikipedia contributors. *Universal Transverse Mercator coordinate system — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system. [Online; accessed 21-June-2021]. 2021 (cit. on p. 38).

[28] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Alexander Kleiner, Giorgio Grisetti, and Wolfram Burgard. "Large Scale Graph-based SLAM using Aerial Images as Prior Information." In: *Autonomous Robots* 30 (Jan. 2009), pp. 25–39 (cit. on p. 25).

[29] Matthias Hentschel and Bernardo Wagner. "Autonomous robot navigation based on OpenStreetMap geodata." In: *13th International IEEE Conference on Intelligent Transportation Systems*. 2010, pp. 1645–1650 (cit. on p. 25).

[30] Georgios Floros, Benito van der Zander, and Bastian Leibe. "OpenStreetSLAM: Global vehicle localization using OpenStreetMaps." In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 1054–1059 (cit. on p. 25).

[31] ROS Wiki. *ROS/Introduction*. http://wiki.ros.org/ROS/Introduction (cit. on p. 29).

[32] ROS Wiki. *ROS/Concepts*. http://wiki.ros.org/ROS/Concepts (cit. on p. 29).

[33] ROS Wiki. *nodelet*. http://wiki.ros.org/nodelet (cit. on p. 30).

[34] Wikipedia contributors. *Covariance matrix — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Covariance_matrix. [Online; accessed 27-June-2021]. 2021 (cit. on p. 40).

[35] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623 (cit. on pp. xvii, 1, 9, 40).

[36] Jean-Philippe Barrette-LaPierre. *cURLpp | C++ wrapper around libcURL*. http://www.curlpp.org/ (cit. on p. 52).

[37] Kalicinski, Marcin and Redl, Sebastian. *Chapter 32. Boost.PropertyTree*. https://www.boost.org/doc/libs/1_76_0/doc/html/property_tree.html (cit. on p. 53).

[38] Frank Dellaert and Michael Kaess. "Square Root SAM: Simultaneous Localization and Mapping via Square Root Information Smoothing." In: *I. J. Robotic Res.* 25 (Dec. 2006), pp. 1181–1203 (cit. on p. 17).

[39] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, Benson Limketkai, and R. Vincent. "Efficient Sparse Pose Adjustment for 2D mapping." In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2010), pp. 22–29 (cit. on p. 17).

[40] K. Konolige. "Sparse Sparse Bundle Adjustment." In: *BMVC*. 2010 (cit. on p. 17).

[41] Hauke Strasdat, J. Montiel, and Andrew Davison. "Scale drift-aware large scale monocular SLAM." In: vol. 2. June 2010 (cit. on p. 17).

[42] Wikipedia contributors. *OpenMP — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/OpenMP. [Online; accessed 30-June-2021]. 2021 (cit. on p. 19).

[43] Wikipedia contributors. *Streaming SIMD Extensions — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions. [Online; accessed 30-June-2021]. 2021 (cit. on p. 19).

[44] ROS Wiki. *nodelet*. http://wiki.ros.org/Names (cit. on p. 30).

[45] R. Rusu, Zoltán-Csaba Márton, Nico Blodow, Mihai Emanuel Dolha, and M. Beetz. "Towards 3D Point cloud based object maps for household environments." In: *Robotics Auton. Syst.* 56 (2008), pp. 927–941 (cit. on p. 32).

[46] M. Fischler and R. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography." In: *Commun. ACM* 24 (1981), pp. 381–395 (cit. on p. 32).

[47] www.dirsig.org. *Coordinate Systems*. http://www.dirsig.org/docs/new/coordinates.html (cit. on p. 38).

[48] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. "Real-time loop closure in 2D LIDAR SLAM." In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278 (cit. on p. 34).

[49] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J. Leonard. "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age." In: *IEEE Transactions on Robotics* 32.6 (2016), pp. 1309–1332 (cit. on p. 13).

[50] Andreas Geiger, P Lenz, Christoph Stiller, and Raquel Urtasun. "Vision meets robotics: the KITTI dataset." In: *The International Journal of Robotics Research* 32 (Sept. 2013), pp. 1231–1237 (cit. on pp. 75, 76).

[51] Tomáš Krejčí. *ros2bag*. https://github.com/tomas789/kitti2bag (cit. on p. 77).

[52] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. "A benchmark for the evaluation of RGB-D SLAM systems." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 573–580 (cit. on p. 78).

[53] ROS Wiki. *rviz*. http://wiki.ros.org/rviz (cit. on p. 75).

[54] Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. *Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks*. 2013. arXiv: 1301.3853 [cs.LG] (cit. on pp. xvii, 1).

[55] Giorgio Grisetti, Slawomir Grzonka, Cyrill Stachniss, Patrick Pfaff, and Wolfram Burgard. "Efficient estimation of accurate maximum likelihood maps in 3D." In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2007, pp. 3472–3478 (cit. on pp. xvii, 1).