UCLouvain

École polytechnique de Louvain

Simultaneous Localization and Mapping and Autonomous Navigation on SPOT robot from Boston Dynamics

Authors: **Achille Morenville, Lucas Vermeulen**
Supervisors: **Pierre Schaus, Siegfried Nijssen**
Readers: **Jean Gillain, Sébastien Jodogne, Timothée Lonfils**
Academic year 2021–2022
Master [120] in Computer Science and Engineering / Master [120] in Electro-mechanical Engineering

Abstract

Achieving autonomous navigation of robots can open doors to many applications. Boston Dynamics has developed the Spot robot to enable robust, safe, and agile movement over many types of terrain. However, Spot's navigation is only semi-autonomous – it needs an operator to create the path that the robot can then repeat on its own. This navigation method is not adaptive as it requires the operator to repeat the initialization step should Spot need to take another path. In a very dynamic and changing environment such as a construction site, this can become a problem. Indeed, it is necessary to change very regularly the predefined paths to maintain efficiency in the navigation, if not to avoid the robot to be lost. In addition, Spot doesn't have a solution for mapping environments in a precise way and the methods available today for that purpose are slow and cumbersome.

To improve the navigation and the possible applications of Spot (e.g., mapping, monitoring), we have developed a Simultaneous Localization and Mapping (SLAM) algorithm based on a graph optimization technique. It effectively uses the point clouds features received from the Spot's LiDAR and detects loop-closures in the robot's path to allow precise localization. This enables the mapping of the robot's environment in a precise and real-time way. In addition, we have also developed a path-planning algorithm based on an improvement of A*: Jump Point Search. A filtering method of the computed path has also been built to transform the output of the navigation into an efficient set of instructions for the robot.

Tests carried out on Spot in buildings demonstrate the effectiveness of the proposed solutions. Environments can be precisely reconstructed and the robot can navigate autonomously inside them. The need for an operator to construct predefined paths that Spot can take is removed.

Acknowledgments

Five years have led us to this master thesis. Probably the most important years of our life. These formative years, rich in discoveries and encounters, have allowed us to develop as engineers, but also as humans. For these valuable years, we would like to thank several people.

In the context of this master thesis, we would like to thank the following people :

- Firstly, we would like to thank Prof. Pierre Schaus and Prof. Siegfried Nijssen for their supervision. Prof. Pierre Schaus has been very patient with us and continuous in his support and advice.
- Secondly, Jean Gillain, for all the help, for his good humor, his kindness, and the precious advice he gave us.
- Finally, the BBRI for granting us access to Spot and their infrastructures. Special thanks to Timothée Lonfils and Angelo Buttafuoco for their support, advice, kindness, understanding, and flexibility. A small nod to Timothée's amazing sarcasm that often baffled us.

Then, for the many years spent at the University, and all the experiences we have had, we would like to thank :

- All the excellent teachers we have had, both at Bachelor's and Master's levels. They have trained us and participated greatly in our personal development.
- All our friends, who sometimes had to endure "meaningless engineering discussions" and who made the years of study the best of our lives.
- Our families for their unwavering support, their eternal support, and the faith and trust they have given us.

Achille Morenville, Lucas Vermeulen

Contents

Introduction

The autonomous movement of robots enables nowadays many applications; we all have seen viral videos of robots performing tasks that are usually done by humans. Robots can also be used to perform mapping of environments that are hazardous to humans, inspect industrial areas, or monitor the progress of buildings under construction. In this context, robots need to be mobile and robust to changing environments.

The Spot quadruped robot developed by Boston Dynamics offers mobility, robustness, safety, and agility on many types of terrain. The Belgian Building Research Institute (BBRI) has procured a Spot robot with the objective to explore its potential applications in the building construction domain. Spot's navigation is semi-autonomous, meaning that the robot can only follow paths predefined by an operator. This thesis aims to develop a more autonomous navigation and a better mapping ability. More precisely, the objective is to allow Spot to move from point A to point B without any prior information about its environment. This is achieved in two parts: Firstly, through a Simultaneous Localization and Mapping (SLAM) algorithm that allows Spot to locate itself in an unknown environment and map it accurately; secondly, a navigation algorithm that directs Spot intelligently to its target. The proposed solutions allow Spot to move autonomously through real-time information processing.

A possible application can be found in the context of monitoring the progress of a building under construction. Solutions already exist to map the environment using fixed laser towers. However, their usage is cumbersome and time-consuming. Our solution could provide an alternative that is both easier to deploy and faster in map creation.

Chapter 1

Spot

1.1 General description

Spot is a mobile quadruped robot with 12 degrees of freedom, 3 per leg [1]. A visual description taken from the Spot SDK provided by Boston Dynamics is shown in Fig. 1.1. Its dimensions are: 1100 mm long, 500 mm wide and 840 mm high when standing. Its net weight is 32.5 kg without payload. Its maximum speed is 1.6 m/s. Spot is capable of straddling obstacles with a maximum height of 30 cm. The time of use of the battery in activity is approximately 90 min. Moreover, it is possible to attach to Spot one or two payloads, with a total mass of a maximum of 14kg.

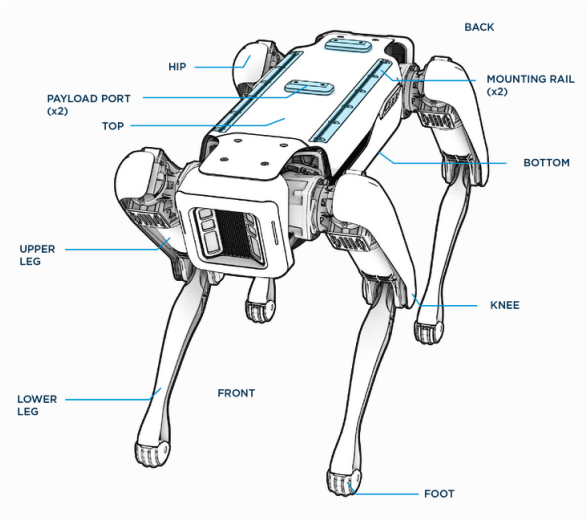


Figure 1.1: Visual representation of the main characteristics of Spot [2].

1.2 Communication with Spot

To communicate with Spot, there are several solutions, depending on the application. The different possibilities are listed below [1]:

- **Communication connected peer-to-peer** : It is possible to connect to Spot physically through its RJ-45 port, through the DB-25 port of the payload or an RJ-45 port connected to the GXP payload.
- **Spot as Wifi access point** : Customers can connect to Spot via Wi-Fi, with Spot directly as an access point. This allows for a wireless connection, with a limited range.
- **Spot as a Wifi client** : Spot can connect to a shared Wifi, which the client can also connect to communicate with the robot. This increases the range of a direct Wifi connection. However, care must be taken in dead zones where the WiFi connection is weak or non-existent.
- **Custom communications** : Other communication solutions also exist, such as mobile data. These solutions increase the possibilities offered by the previous solutions.

1.2.1 gRPC

gRPC is an application layer protocol that is used by much of the Spot API [3]. With gRPC, a client application can call a function on a server application, even if the two applications are coded in different languages. This protocol was chosen by Boston Dynamics for Spot because of the security and performance it offers, with compatibility with a large number of programming languages.

gRPC offers the possibility to create services that will use Remote Procedure Calls (RPC) to transmit data. This means that we will define a service, with its methods that can be called as well as its parameters and the type of response that it returns. On one side we have the server that implements an interface that will manage the client calls, and on the other side a client, potentially coded in another language, that has a version of the interface with the same functions, and that it can call. Fig. 1.2 provides a visual explanation.

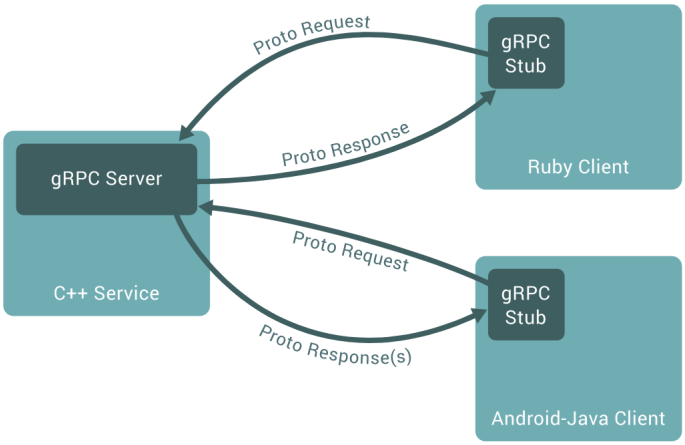


Figure 1.2: Visual description of a gPRC use case. Here, a C++ server communicates with a Ruby and Android-Java clients through Proto requests and responses.[3]

In the example shown in following piece of code [4], a service is defined in a `.proto` file. The services use the Protocol Buffer data structure and the arguments given to the Remote Procedure Call are the Protocol buffers messages. These are also defined in a `.proto` file and are structured as messages that contain a series of variables. These variables consist of a type, a name and an associated value. In the example, `HelloRequest` and `HelloReply` are protocol buffers messages.

```

// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

An example with Spot is the following: the API has an authentication service which is the following:

```
service AuthService {
    rpc GetAuthToken(GetAuthTokenRequest) returns (GetAuthTokenResponse) {}
}
```

Here, the RPC is `GetToken`, and its function is to check the username and password, and it returns a token if it is valid. `GetAuthTokenRequest` and `GetAuthTokenResponse` messages are Protocol Buffer messages. For example, `GetAuthTokenRequest` is defined as :

```
message GetAuthTokenRequest {
    // Common request header.
    RequestHeader header = 1;
    // Username to authenticate with. Must be set if password is set.
    string username = 2;
    // Password to authenticate with. Not necessary if token is set.
    string password = 3;
    // Token to authenticate with. Can be used in place of the password, to re-mint a
    token.
    string token = 4;
}
```

Then, we can use this service with a python function of the API :

```
 def auth(self, username, password, app_token=None, **kwargs):
        """Authenticate to the robot with a username/password combo.

        Args:
            username: username on the robot.
            password: password for the username on the robot.
            app_token: Deprecated.  Only include for robots with old software.
            kwargs: extra arguments for controlling RPC details.

        Returns:
            User token from the server as a string.

        Raises:
            InvalidLoginError: If username and/or password are not valid.
        """
        req = _build_auth_request(username, password, app_token)
        return self.call(self._stub.GetAuthToken, req, _token_from_response,
    _error_from_response,
                         **kwargs)
```

`_build_auth_request()` creates a message `GetAuthTokenRequest` that is sent by RPC to the `AuthService` that returns a `GetAuthTokenResponse` message.

1.3 Services

1.3.1 Stereo cameras

Spot is equipped with 5 stereo cameras [1], capable of rendering a black and white image with a range of 360°. It is also possible to calculate an image with depth information through mathematical operations on the images returned by the stereo cameras. An example of the depth estimation is shown in Fig. 1.3. The data given by the cameras can be returned in different formats (JPEG, RLE, RAW) depending on the application.

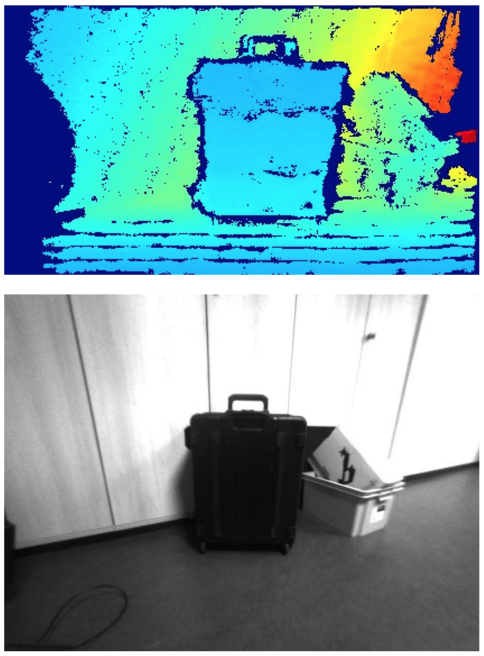


Figure 1.3: Example of images taken from one of the front cameras of Spot. The image on the top is the estimated depth computed by Spot.

Finally, the developer cannot retrieve the result of a single camera, only the result of the reconstructed images can be used.

1.3.2 Robot State

Spot keeps track of different frames to help it describe itself and the objects around it [5] . Everything is represented using a frame. As an example, a frame is used to represent the body of the robot as

shown in Fig. 1.4. The spatial relationships between the frames are defined as transformations that are described using a translation vector $[x, y, z]$ and a rotation quaternion $[x, y, z, w]$. The translation vector is the difference between the origin of the two frames and the rotation quaternion is the difference between the axis orientations. As an example, these transformations between frames can be used to know the location of sensors on Spot or to compute the position and orientation of Spot.

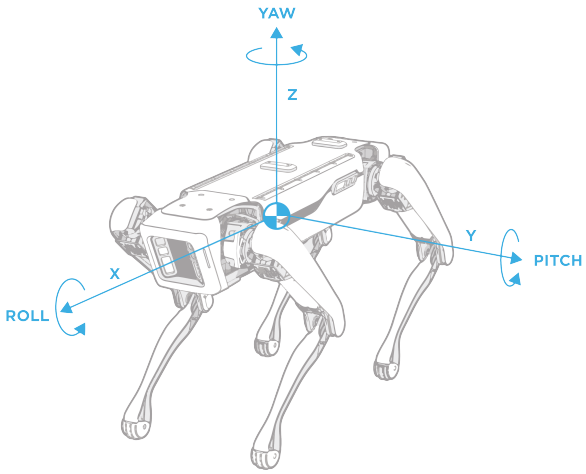


Figure 1.4: The frame representing the body of the robot. It is centered at the geometric center of the front and rear hips, and the geometric center of the side legs. The x-axis is directed towards the front of the robot and the z-axis towards the top [5].

.

Spot keeps track of multiple frames and here are some of the most important ones:

- The "**odom**" and "**vision**" frames are inertial frames, or reference frames, that estimate the fixed location and orientation of where the robot boots up.
- The "**body**" frame represent Spot's body.
- The sensors frames represent the sensors.
- The object frames are associated with objects detected by Spot.

The transformations between the frames are computed by Spot and are updated at every time step. As they can be related to each other, they are kept inside a tree structure and can be obtained at any time. The transformations can either be static or dynamic. A transformation is static when its two frames remain in the same place relative to each other. For example, this is the case for the transformations between the "body" frame and the sensors frames as every sensor is fixed on Spot.

The transformations that change with time are dynamic. As an example, the transformation between an object frame and the "body" frame might change when the object is moved with respect to Spot.

The most important transformations are the transformation that relates the position and orientation of Spot to its boot-up place, which is the world origin. These are the transformation between the "odom" or the "visual" frame and the "body" frame. Both are updated as the robot moves but they use different techniques to update the transformation. The transformation between the "odom" frame and the "body" frame is computed using the kinematics of the robot and is commonly called odometry. The transformation between the "vision" frame and the "body" frame uses the cameras on top of the kinematics to refine the movement estimation. They can be used to track the overall displacement of Spot.

Transformations can also be represented using homogenous coordinates [6]. Homogenous coordinates represent a 3D space in a 4D space to ease computations of translations and rotations. A point $[x, y, z]^T$ is represented as $[x, y, z, 1]^T$. This system is useful because it allows for computing a translation and a rotation with a single matrix:

$$T = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{pmatrix}, \tag{1.1}$$

where $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ is a 3D rotation matrix and $\boldsymbol{t} = [t_x, t_y, t_z]^T$ the translation vector.

This representation of the transformation is also useful to change the reference frame of an object. If an object A is represented in the "body" frame with the transformation matrix $T_{body\to A}$ and we have the transformation $T_{odom\to body}$ between the "body" frame and the "odom" frame, then the object A can be expressed in the "odom" frame using the matrix multiplication $T_{odom\to A} = T_{odom\to body}T_{body\to A}$.

Sometimes, it is needed to invert a transformation. It can be done using the formula:

$$T_{B\to A} = T^{-1}_{A\to B} = \begin{pmatrix} \boldsymbol{R}^{-1} & -\boldsymbol{R}^{-1}\boldsymbol{t} \\ \boldsymbol{0} & 1 \end{pmatrix}, \tag{1.2}$$

where $\boldsymbol{R}$ and $\boldsymbol{t}$ are the original rotation matrix and translation vector of the transformation.

1.3.3 LiDAR

Spot is equipped with a LiDAR provided by the autonomy payload. A LiDAR is a sensor that measures distances to nearby objects and terrain. It estimates the distances by firing a laser in a direction and by measuring the time needed for the light to reflect on the surface and travel back to the sensor. The distance to the nearest object in the direction of the laser can easily be computed with the formula $d = \frac{c\cdot t}{2}$, where c is the speed of light and t is the time measured. As the LiDAR uses its own light

source it doesn't need a good lighting conditions to work properly. It can even work in the dark.

The LiDAR sensors usually don't fire the laser in a single direction but the laser is rotated to be able to estimate the ranges in several directions. The LiDARs are also not limited to a single laser and can be equipped with multiple ones arranged perpendicularly to the plane of rotation. This allows the estimations of the distances to not be limited to a plane. Each laser is called a channel. Fig. 1.5 represent a LiDAR with 8 channels which are represented in blue.

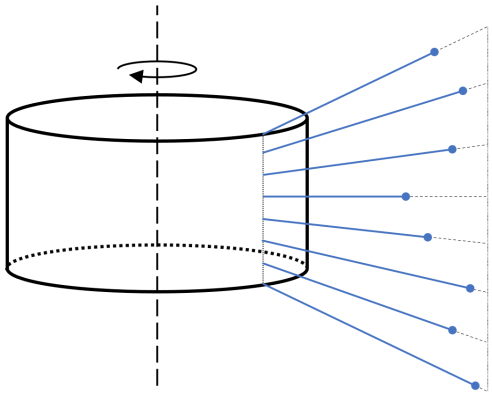


Figure 1.5: Schema of a LiDAR.

Using the distance and the direction of each beam, we can compute the points where the light reflections append. This becomes a set of 3D points also called a point cloud. An example of the point cloud of a room is visualized in Fig. 1.6. Most of the time it is in this format that the data from the LiDAR is handled.

Spot's LiDAR is a Velodyne VLP-16 with 16 channels [7]. Its vertical field of view is 30° (from $-15°$ to 15°) and its horizontal field of view is 360°. Its vertical angular resolution is 2° and its horizontal angular resolution depends on its rotation rate but ranges from 0.1° to 0.4°. Its rotation rate can be set from 5 Hz to 20 Hz. The LiDAR can measure a distance of up to 100m with an accuracy of +- 3cm.

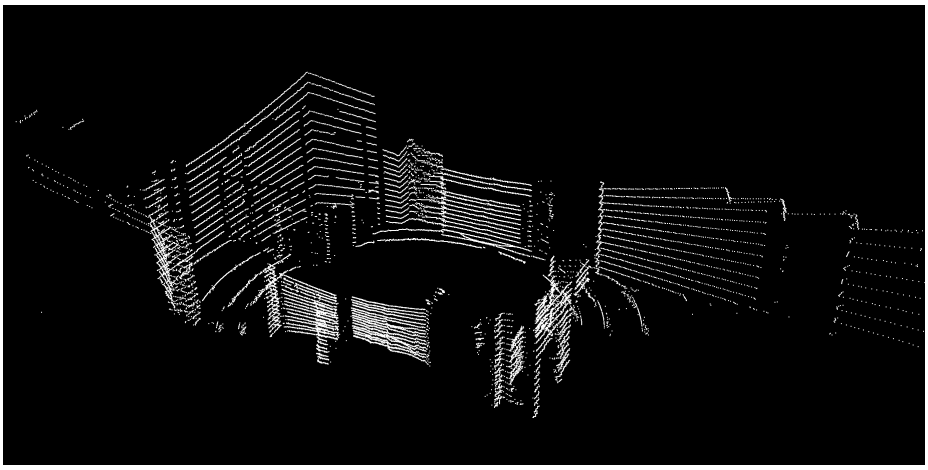


Figure 1.6: Point cloud of a room acquired from the LiDAR mounted on Spot.

One problem with mounting a LiDAR on a moving robot is that some distortion can appear in the point clouds if it is taken when the robot is moving. This is because the point cloud capture is not instantaneous. As the LiDAR is moving, ranges are not taken from the same center and thus the measurements are not consistent. Fortunately, the robot corrects this distortion using its estimated displacement during the capture. The idea is to use Spot's IMU (Intertial Measurement Unit) velocity estimate and the acquisition time of each point to estimate and correct the deformation of each point.

1.3.4 Local grids

Spot can locally map its environment at a range of 3.84 m represented as a 128×128 grid, thanks to its various cameras and sensors [1]. The robot then creates, at each instance of time, a local grid around him. This grid can be of different types:

- **Terrain** : The robot evaluates its environment and classifies it according to its height. Each cell of the grid will have a value corresponding to its height.
- **Terrain-valid** : Grid that can be combined with the Terrain type grid, and adds a filter to eliminate invalid estimates, i.e. non-sense values.
- **Intensity** : The values in each cell are mapped to a gray value. This type of map can then be used to colour and represent the grid visually.
- **Obstacle distance** : Local grid where the values of each cell correspond to the respective distance to the nearest obstacle.

- **No-step** : Local grid that determines where the robot can move, depending on the distance to obstacles and their height. An example of a no-step grid is shown in Fig. 1.7

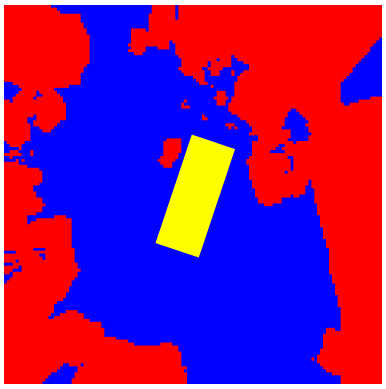


Figure 1.7: Example of a no-step local grid acquired with Spot. Spot is represented in yellow. The red cells represent places where the robot cannot walk on.

1.3.5 World objects

Spot can recognize certain objects thanks to its various sensors [1]. These can be of different types such as stairs, a door handle, April Tag or even objects that a developer can add. Spot detects these objects and stores a certain amount of information depending on the object in question. However, they share common information, such as a human-readable name, a time of detection or a frame that locates them in the environment and with respect to Spot.

Currently, the most widely used detectable object is the AprilTag [8]. It is used to locate accurately because it represents a known reference in the environment. An example of AprilTag is shown in Fig. 1.8. Spot can use the information that the detection of an April Tag offers him concerning his localization to locate himself precisely in the environment.

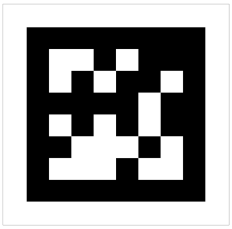


Figure 1.8: Example of AprilTag detectable by Spot [9].

1.3.6 GraphNav

The Spot SDK provided by Boston Dynamics provides APIs that allow the customer to implement several autonomous navigation behaviours for Spot [1]. The set of mapping, localization and movement services of Spot is called GraphNav. The robot is controllable via a tablet, and a client can use an Auto-Walk function to implement a path that can be executed by Spot automatically. The path that the robot must take must first be done manually by the client for Spot to save the map and the path it must take. Then, this saved data is used by Spot to locate itself and repeat the predefined path autonomously. When Spot is on a prerecorded path, it can be given missions to perform.

For Spot, the environment is represented as a graph, with nodes and edges connecting them. The nodes are called waypoints. Fig. 1.9 represents this structure.

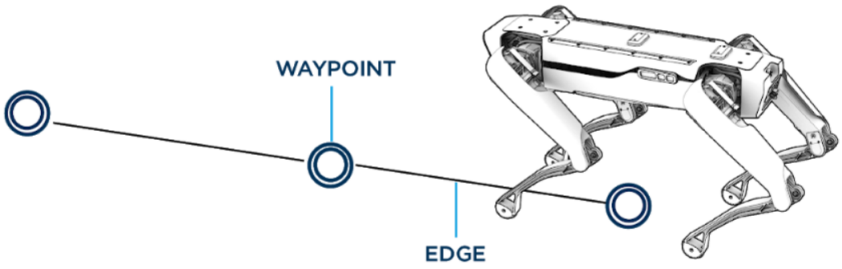


Figure 1.9: Representation of the environment as a graph of waypoints and edges [10].

Waypoints are locations in the environment where Spot records data locally. At each waypoint, the robot will use the sensors it has (stereo cameras, LiDAR, ...) to record information around it. It is therefore important to have as much information as possible to locate the robot. A neutral location will not offer much information and therefore the localization will be less accurate. The ideal is to have an AprilTag for a maximum of information. A waypoint is created automatically every two meters, or manually by the operator.

The **edges** link the waypoints together and contain information about the transformations between the waypoints (position, orientation). In addition, edges also contain data about the environment between waypoints. For example, if 2 waypoints are separated by a staircase, then the edge will contain information about the staircase, as shown in Fig. 1.10. The robot then knows the behaviour it has to adopt when it goes from waypoint A to waypoint B.

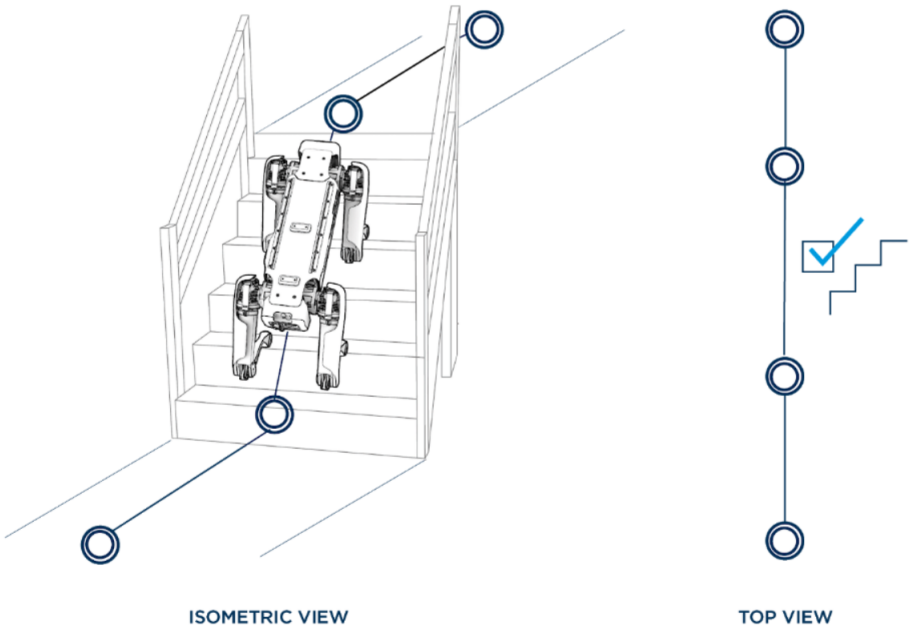


Figure 1.10: The edges contain data describing the topology of the route between two waypoints, as staircases [10].

When the robot is going to use a stored map, it must first find its way around the map. It is then important to give the robot a piece of location information at its **initialization**, thanks to an AprilTag for example, as shown in Fig. 1.11. This AprilTag must of course be present in the registered map to give Spot a transformation between its current location and the location of the nearest waypoint. It is also possible to manually give a transformation between its position and a waypoint. After this initialization, the robot will continue to locate itself in relation to the different waypoints thanks to the information given by its sensors.

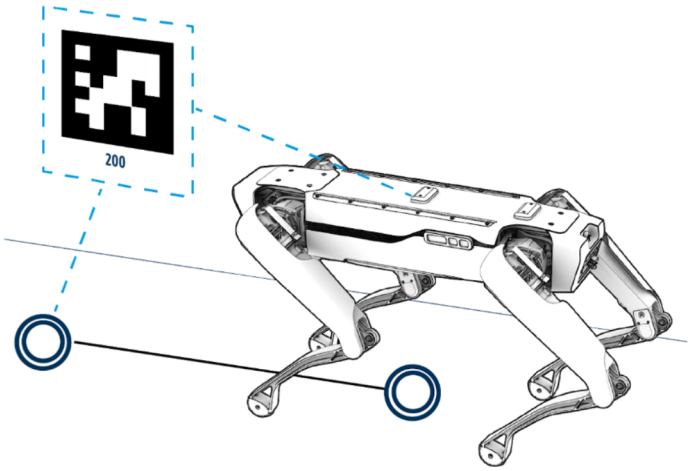


Figure 1.11: At initialization, Spot must find its way in the environment. The solution offered by the AprilTag system is robust and efficient [10].

As said before, for Spot the environment is represented by a graph, with waypoints and edges. The topology of this graph can be diverse, with a simple chain structure, a tree structure or even loops, as shown in Fig. 1.12.

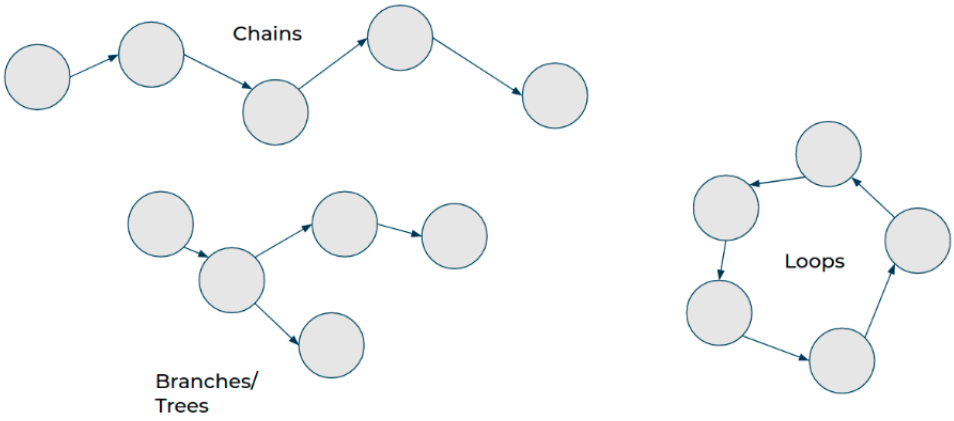


Figure 1.12: The different toplogies that the graph representing the world for Spot can have [10].

The **loop structure** is interesting because when the robot makes a path and returns to a place it has already passed, it is important for Spot to build a loop to know that there is another path to a waypoint in that loop. This concept is called loop-closure. The value of loop-closure can be

explained by the following example: if Spot travels 100 meters before returning to its starting point, it is interesting to know that it can reach its end point without having to travel the 100 meters again. The loop closures are closed only with the use of the relative position estimations, no external measurements are used.

Finally, Spot can sometimes get **lost**. It will inform the operator that it is lost and stop all manoeuvres. It must then be initialized again by the operator because it cannot get out of its lost condition alone.

1.4 Why Spot has to be augmented ?

In its current state, Spot faces limitations when the environment is dynamic. Indeed, the robot is capable of following predefined paths but needs an operator each time it needs to change these paths. In a context such as construction, where the environment evolves rapidly, the possibilities offered by Spot are limited. It is therefore interesting to develop new solutions to facilitate the creation of new paths and thus increase the robot's autonomy. In addition, the robot's ability to create maps is quite limited. The maps created by Spot are made to allow the robot to move around and not to accurately map a complex environment. On the one hand, precise mapping of a potentially unknown environment is necessary to locate and understand where Spot should evolve. On the other hand, moving from a new point A to a new point B in a more autonomous way, i.e. without the need for an operator, is an improvement offering possibilities in many applications. These two parts are complementary and together constitute a more autonomous navigation of Spot that we will develop in the next chapters.

Chapter 2

Simultaneous Localization and Mapping

Our goal is to create a precise map of an environment and localize Spot within it to autonomously navigate the robot through the environment. These are two common problems in robotics, the localization and the mapping problems. The first one is the problem of estimating the robot's pose, position and orientation, while the second is to create a map of the environment in which the robot is moving. In this context, the two must be solved simultaneously, which is also known as the Simultaneous Localization and Mapping (SLAM) problem.

While this problem is easy to describe, it is challenging to solve. A map is usually used to estimate the robot's pose, and the poses are often needed to build an accurate map [11]. While some solutions have been found for specific environments, it is still an open research problem. As you can imagine, finding a robust and general solution can open doors to many different new applications. They range from autonomous driving to autonomous exploration of dangerous environments. So, it is commonly considered one of the most significant problems in the autonomous mobile robot field [12].

This chapter will first define more formally the SLAM problem. It will then explore its characteristics and the different approaches used to solve it. Finally, it will explain the chosen technique given the context.

2.1 Definition of the SLAM problem

As explained before, the SLAM problem can be defined as the problem of constructing a map of an unknown environment while simultaneously localizing a robot within it using imperfect sensors.

2.1.1 Formal definition

More formally the problem can be defined using a probabilistic approach [12]. Let denote x_t the position and orientation, pose, of the robot at time t and $x_{0:T} = \{x_0, x_1, x_2, \ldots, x_T\}$ the set of poses, the path, from time 0 to time T. Here, x_0 is the known initial position of the robot.

The robot can give some information about its displacement. Let u_t be the displacement of the robot between time $t-1$ and t, this quantity is also called the odometry [12]. It can be calculated in many different ways but it is often calculated by integrating inertial measurements. The sequence $u_{1:T} = \{u_1, u_2, \ldots, u_T\}$ describes the overall displacement of the robot. If the odometry was perfect, then there would be no challenge in finding the robot's position as the initial position is known. But in reality, the odometry is noisy, and the error in the poses estimated from it will increase with the length of the trajectory [12].

Let us define m as the true map of the environment. The robot can sense this environment through sensors e.g. LiDARs, cameras, ... The measurement of the environment got from the sensors can be defined as z_t. As for the odometry, the sequence of measurements can be defined as $z_{1:T} = \{z_1, z_2, \ldots, z_T\}$. While the measurements are less noisy than the odometry, there are still some uncertainties. For example, there is noise in the camera images or the reflection of lasers can cause wrong estimations of the LiDAR's ranges. And it is because of the uncertainties in the measurements and the odometries that the SLAM problem is hard.

The SLAM problem becomes the problem to recover the true map m and the sequence of poses $x_{0:T}$ from the set of measurements $z_{1:T}$ and the odometries $u_{1:T}$. This is equivalent to estimating the posterior of the posterior probability distribution

$$p(x_{1:T}, m \mid u_{1:T}, z_{1:T}, x_0). \tag{2.1}$$

In other word, it is finding the $x_{1:T}$ and m that maximize this probability. This formulation, is called *full* SLAM [12] as it recovers the full path of the robot.

If recovering the last pose is enough then it becomes the *online* SLAM [12] problem and can be described as estimating the posterior of

$$p(x_T, m \mid u_{1:T}, z_{1:T}, x_0). \tag{2.2}$$

2.1.2 SLAM properties

The description of the SLAM problem done above is still a general one. There exist many different versions of this problem that depend on the type of map, the type of measurements, or the properties of the environment. Therefore, before exploring the different ways this problem can be solved, the version of the SLAM problem we have to solve must be detailed.

The environment can either be static or dynamic. A dynamic environment can change over time as opposed to a static one. Ideally, the algorithm must be able to handle dynamic environments but this makes the problem much harder as dynamic objects must be detected and processed accordingly. So for now, the environment will be assumed as a static one.

The map can either be described using a feature-based representation or a volumetric representation [12]. The first one uses features, also called landmarks, extracted from the measurements of the environment to represent the map. This type of representation is quite sparse as the map is represented using only a few specific aspects of the environment. The second representation, also called dense representation, describes the map with the highest resolution possible. In this application, it would be preferable if the map could be used afterward to extract data on the construction site. This limits it to be a volumetric map as a feature-based one is quite abstract. An example of a volumetric map is shown in Fig. 2.1.

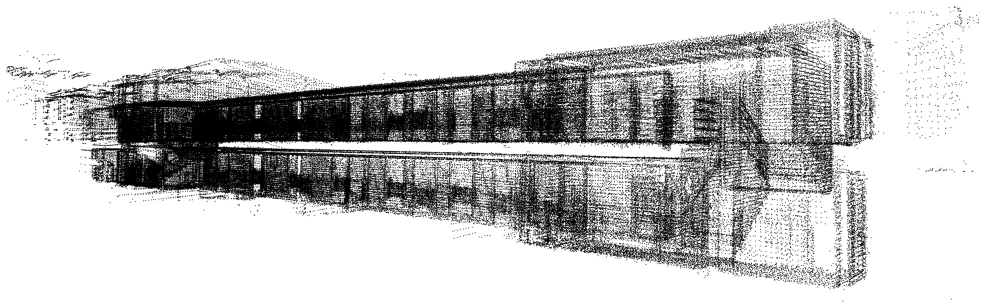


Figure 2.1: Example of a volumetric map of two floors with a staircase. The map is a concatenation of many point clouds retrieved from the LiDAR of Spot and assembled using a SLAM algorithm.

For autonomous navigation, the robot must be able to update the map and give a pose estimate in real-time during the robot exploration. This will limit the computational time possible for the computation of the map.

2.2 SLAM approaches

There are a lot of different solutions to the SLAM problem but most of them can be categorized into two categories, the filtering and the smoothing techniques [13]. The filtering approach models the problem using the online formulation (2.2) where they estimate a state composed of the current robot's pose and the map. Most of the solutions that follow this technique use either a Kalman filter or a particle filter [13]. The smoothing approach uses the full formulation of the problem (2.1) as they estimate the full path of the robot using all the acquired measurements. Usually, the smoothing solutions use a Graph-based optimization approach [13].

This section will introduce the three methods cited and expose the specific aspects that helped us choose the Graph-based methods over the other ones.

2.2.1 Kalman filters

The Kalman filter is a technique that is used to estimate a variable, a state, that can't be measured directly. The main idea is to estimate the current state of the system by fusing multiple measurements that are linked to it [14].

The Kalman filter is a recursive algorithm that alternates between two steps, the prediction steps and then the update step. It starts with an initial estimation of the state. Then the predict step updates the previous state estimation with a state transition model to predict what the current state is. During the update step, the algorithm compares the measurements received and the current state using an observation model and updates the state estimation to better match the measurements. On top of that, the Kalman filter uses the known uncertainty of each measurement to put more weight on the more certain ones.

This principle is visualized in Fig. 2.2 using a simple example in one dimension that uses only one measurement [11]. In this example, the state is the position of the robot along a single dimension. At first, the filter has an initial estimation of the position of the robot and its uncertainty (in red). When the robot moves to the right, the prediction step updates the estimation and the uncertainty of the position (in orange) using the state estimation model. As this is not a perfect model, the uncertainty increases. Then a sensor gets a position measurement and its associated uncertainty (in blue) and the update step corrects the position estimation and obtains a better value for the position (in green).

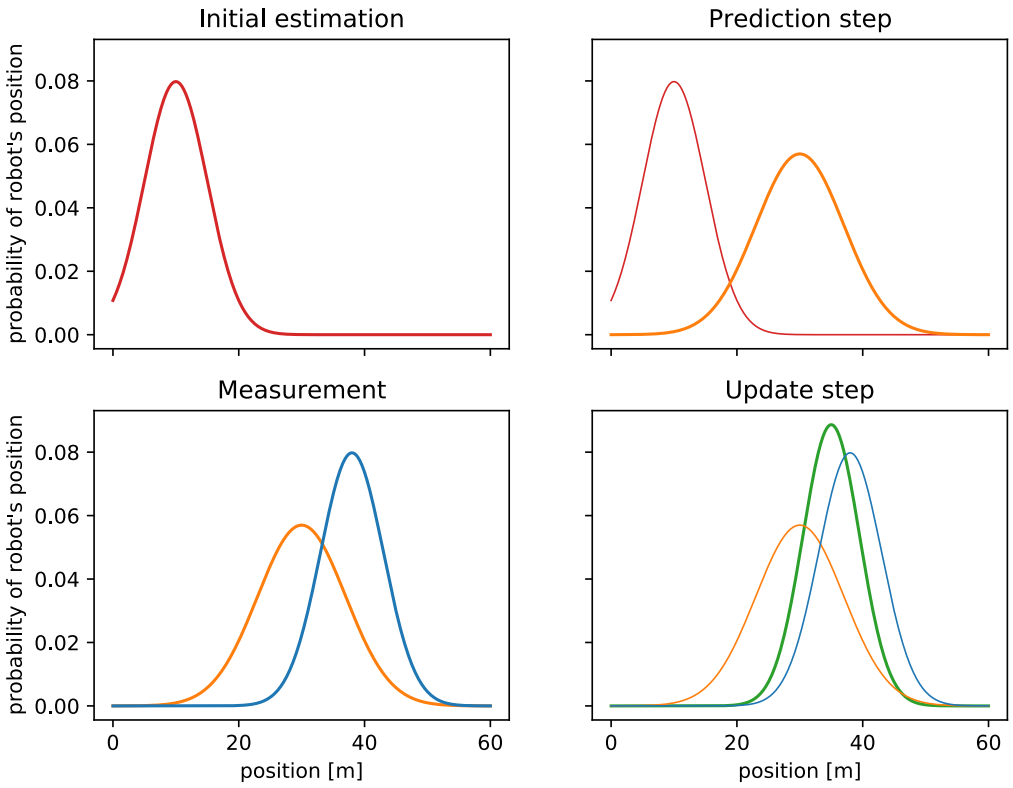


Figure 2.2: Example of the Kalman filter in one dimension with one measurement. The filter estimates the position of a robot in a single direction. The Kalman filter estimates a final position (in green) by fusing the measurements (in blue) and the predicted position (in orange). [11]

In the regular Kalman filters, the state estimation and observation models are both linear but in some applications, like for the SLAM problem, non-linear assumptions are needed [11]. To resolve this issue some variants of the Kalman filter were developed. The most used variant is the Extended Kalman filter which was the first technique used to solve the SLAM problem and the most influential one [12].

In the SLAM problem, the state of the filter is the pose of the robot and the map as they are the variables that need to be estimated. Unfortunately, it is more adapted to feature-based maps as it is a finite list of landmarks. While it can be modified to work with dense maps there are other methods more suited for this application.

2.2.2 Particle filters

The particle filter is another well-known state estimation technique. The principle is to represent the state of a system as a set of possible hypotheses, or particles [15]. At the start, a set of particles is drawn randomly. Then when the state changes the particles can be modified according to the estimated change and keep only the particles that are the most consistent with the measurements. Finally, some other particles can be generated according to the distribution of the particles kept so that the right number of particles is reached. As one can expect, the more particles used the better the estimation becomes.

One problem in the context of the SLAM problem is that the estimated state is the combination of the pose of the robot and the entire map as these are the variables of the problem. And each particle contains its estimation of the state. But the maps can take up a lot of computational space, especially for 3D volumetric maps. This limits the total amount of particles that could be used. Furthermore, as this method uses randomness, many hypotheses are needed to obtain enough precision.

2.2.3 Graph-based optimization

The idea behind the graph-based optimization techniques, also called Graph-SLAM, is to represent the problem under a graph [13]. The nodes in the graph represent the robot's poses and each one of them is associated with their corresponding measurements. The edges in the graph are the spatial constraints between the poses that can be computed using the odometry and optionally from the measurements. This graph is also called a pose graph. The graph can then be optimized to update the pose of each node so that they satisfy the constraints the most.

To give more sense and bring more understanding to the optimization process, the problem can be visualized in another way. The nodes can be represented as masses and the edges as springs. The optimization step becomes releasing the system and letting the springs pull and push on it to find the equilibrium.

This technique solves the *full* SLAM problem as it optimizes the entire path during each optimization. At first, this technique was used for offline optimization, when all the data had already been collected. But in recent years some efficient algorithms, like ISAM2 [16], that use efficient data structures and that reuse past optimizations, were developed which allowed this technique to be used during the robot exploration.

This technique is also quite flexible as it can incorporate multiple types of measurements as long as they can be represented using a graph. For example, landmarks can easily be added to the graph to increase the robustness of the method.

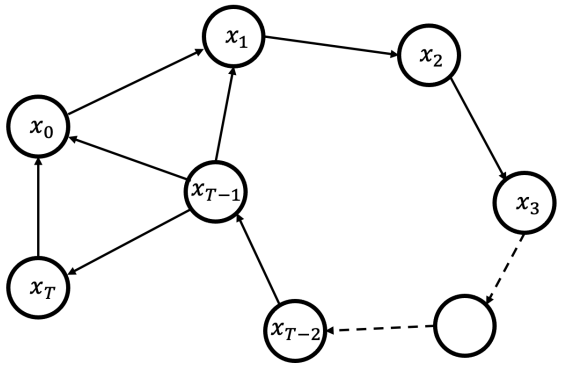


Figure 2.3: Visualization of a graph used for the graph-based optimization approach. Each node is associated to a pose x_t. The nodes are linked following the sequence of the path. Some links are added between non-sequential nodes to add spatial constraints which help for the optimization.

With all these features, we chose this approach over the Kalman filter and particle filter. Furthermore, this technique is now considered the state of the art [12]. It is often even the first approach used when facing a new problem as there are many efficient implementations and because it is easy to set up and comprehends.

2.3 Graph-SLAM

This section will explore the commonly used solution's architecture in the Graph-SLAM technique. Then the graph generation will be explained and finally, the mathematical formulation of the optimization problem will be given.

2.3.1 Architecture of Graph-based SLAM

As explained in section 2.2.3, the Graph-based SLAM technique uses a graph representation to optimize the robot's path. The approach can be split into two distinct parts. The first one is the graph generation part, also called the front-end [13]. It generates the nodes and the edges of the graph and sends them to the graph optimization part, also known as the back-end [13]. The pose associated with each node is then optimized to satisfy the spatial constraints. The graph generation uses the optimized poses so that the graph stays consistent through the robot's execution. The architecture of a general Graph-SLAM solution is depicted in Fig. 2.4

The front-end is more problem-specific as a graph can be constructed using many different mea-

surements. On the contrary, the graph optimization part is more independent as the graph's structure is the same whatever the front-end.

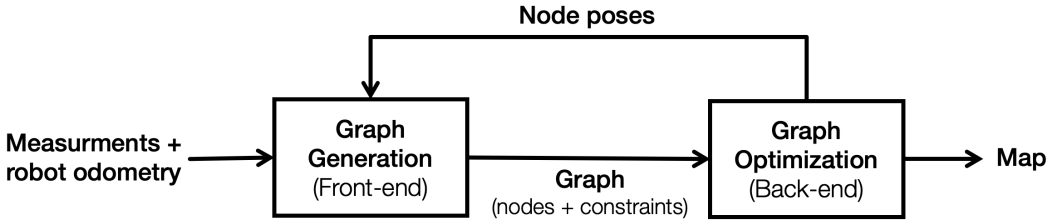


Figure 2.4: Architecture of the Graph-SLAM technique. The front-end takes the measurements and the odometry coming from the robot to generate a graph. The graph is then sent to the back-end to be optimized. A map can be constructed using the optimized graph by assembling the measurements of each node into the proper configuration. [6]

2.3.2 Graph generation

The basic graph generation uses the sequence of robot odometries to generate the graph. The robot odometry at the time step t can be expressed as a transformation matrix $\boldsymbol{X}^R_{W,t} \in \mathbb{R}^{4\times4}$. It represents the transformation between the robot's body position and the world origin frame $\{W\}$ of the robot.

The graph generation starts by creating a first node with pose $\boldsymbol{X}_0$ set as the identity matrix $\boldsymbol{X}_0 = \boldsymbol{I} \in \mathbb{R}^{4\times4}$. This node represents the initial pose of the algorithm and it will be used as the reference frame for the remaining of the execution.

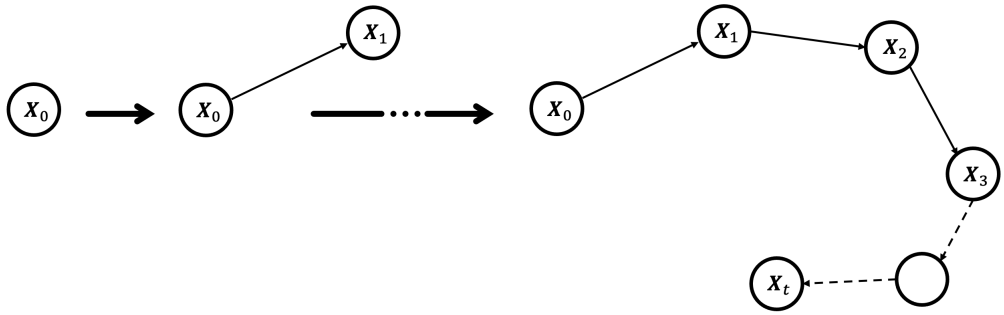


Figure 2.5: Visualization of the sequential graph generation. It starts with an initial node. This node is used as reference for the rest of the graph generation. The nodes are added one after the other and a spatial constraint is added between them.

When the robot moves, a new node can be added to the graph with an edge linking it to the

preceding node. As the frame of reference of the graph is not the same one as the odometry, the odometry can't be used directly to compute the new node position. For that, the robot's displacement must be computed between the last node and the new node and add it to the last node. This displacement can be computed using the odometry of the robot at both poses. The displacement between node $t-1$ and t is $((\boldsymbol{X}^R_{W,t-1})^{-1}\boldsymbol{X}_{W,t})$. Using this displacement the pose of the node t can be set as $\boldsymbol{X}_t = \boldsymbol{X}_{t-1}((\boldsymbol{X}^R_{W,t-1})^{-1}\boldsymbol{X}_{W,t})$. This process is shown in Fig. 2.5.

But optimizing the graph with only sequential edges won't change anything. To be able to optimize the path, non-sequential nodes have to be linked together. A solution to this problem is to link nodes that represent the same location. This is called loop-closure as it forms loops in the graph [13]. For example, if the robot moves again one step in the examples in Fig. 2.5, the new robot's pose $\boldsymbol{X}_{t+1}$ could be linked to nodes 0 and 1 as they are close to each other and they might cover the same part of the environment. The loop-closure edges are represented in blue in the Fig. 2.6.

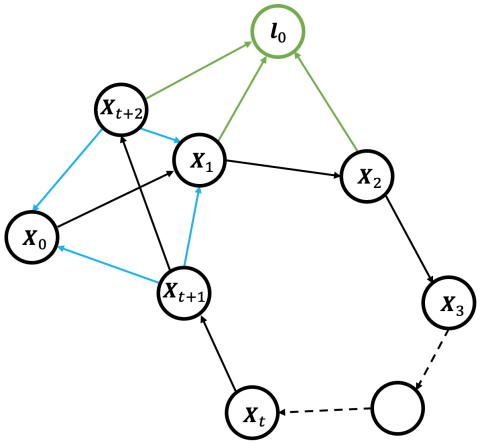


Figure 2.6: Visualization of the loop closure edges and the landmark edges. The edges in blue represent the loop-closure edges that can be added to non-sequential nodes close to each other. Landmarks (shown in green) can also be added to the graph.

As motioned in section 2.2.3, other types of edges and nodes can be added to the graph. For example, nodes that represent landmarks can be added and they can be linked to the nodes that could detect them as shown in green in the Fig. 2.6.

2.3.3 Mathematical formulation of the optimization problem

An example can be used to help formalize the graph-based SLAM technique as an optimization problem [13]. Let $\boldsymbol{X} = \{\boldsymbol{X}_0, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_t\}$ be the nodes' poses in a graph and $\mathcal{C}$ the set of edges. The graph

generation can add an edge that links node i and j. From the measurements of node i, the pose of j relative to i can be estimated as $\boldsymbol{Z}_{ij}$. Therefore, the error $\boldsymbol{e}_{ij}$ can be defined as the difference between the estimated pose $\boldsymbol{Z}_{ij}$ and the pose $\boldsymbol{X}_j$ in the graph. It is visualized on Fig. 2.7. This error function depends on the poses $\boldsymbol{X}_i$ and $\boldsymbol{X}_j$. The goal is to find the poses that satisfy the spatial constraints between the nodes. These poses can be computed by minimizing the error $\boldsymbol{e}_{ij}$. Going back to a more general view, there is an error for each edge in the graph. So, the sum of the squared error over all the edges in the graph can be minimized to find the set of poses that satisfy the constraints the bests.

$$\boldsymbol{X}^* = \underset{\boldsymbol{X}}{\operatorname{argmin}} \sum_{i,j \in \mathcal{C}} \boldsymbol{e}_{ij}^T \boldsymbol{e}_{ij} \tag{2.3}$$

But there are uncertainties in all measurements, and they can be taken into account to put more weight on spatial constraints for which there is more confidence. The uncertainties associated with an estimated pose $\boldsymbol{z}_{ij}$ can be represented using an information matrix $\boldsymbol{\Omega}_{ij}$. On the contrary to the uncertainty, more weight should be put on the edges for which the information matrix is large. The problem can thus be weighted using this information matrix and the problem can be expressed as

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \sum_{i,j \in \mathcal{C}} \boldsymbol{e}_{ij}^T \boldsymbol{\Omega}_{ij} \boldsymbol{e}_{ij}. \tag{2.4}$$

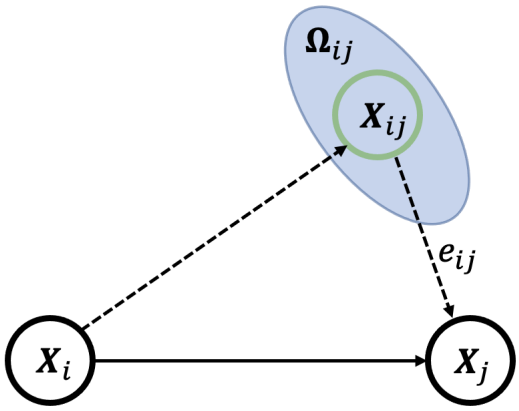


Figure 2.7: Visualization of a sub-problem in the graph optimization. When an edge is added between node i and j, an error e_{ij} can be computed using the measurements of both nodes [13].

Going back to the springs and masses representation of section 2.2.3, the edges' weight can be

associated with the spring's stiffness. The more stiffness, the more the spring will impact the equilibrium.

2.3.4 Spatial Constraints

The spatial constraints between the nodes must be computed to build a graph. This section will explore how sensors can be used to improve the robot's odometry.

For the sequential edges, the odometry returned by Spot can be used. But its uncertainty is significant. The odometry only uses the kinematics of the robot [5]. It only considers the internal measurements of the velocity, acceleration, or legs' positions. But it doesn't take into account the external events that could append. For example, a robot's leg could slip or the terrain could be uneven. Moreover, all the measurements used are estimates. Hence, an error accumulates in the movement estimation. As equation 2.4 shows, the final solution's precision depends on the uncertainties of the edges. One way the odometry can be improved is by using external information from sensors. This process is called sensor odometry.

The robot can already provide a visual odometry. It computes it by analyzing the environment with the five stereo cameras [5]. The exact way Spot computes it is unknown, as are all its internal computations. Usually, visual odometry computes the displacement in two steps [17]. First, the algorithm detects feature points in the successive images. Then it tries to match pairs of points in successive images to estimate the displacement. There exist a lot of different methods to detect feature points in an image, and each one comes with its advantages. Two of the most popular ones are ORB [18] and SIFT [19] algorithms. But there are limitations to the visual odometry. As it is based on cameras, it needs enough light to work. And feature detection can be hard on textureless surfaces [20]. Both those conditions can occur on construction sites. They might not be properly lit and walls are often featureless in empty buildings.

Those problems can be resolved by using the LiDAR as it doesn't need external light to work. Let's imagine two point clouds taken by the LiDAR from poses that are close to each other. Both will contain the same spatial features. As a result, they can be aligned to recover the transformation between them. The problem of aligning point clouds is named point cloud registration and it will be developed in the next chapter. The process can be used to improve the robot's odometry by aligning the new point cloud to the saved past ones and recovering the transformation between them.

Unlike the sequential edges, computations are needed to find the pairs of points that must be linked by loop-closure edges. But as for the sequential edges, a point cloud registration algorithm can be useful. If the point clouds of two nodes can be successfully aligned then a loop-closure edge can be put between the two nodes.

Chapter 3

Point cloud registration

Before explaining the implementation of the Graph-SLAM, this chapter will develop the point cloud registration problem. As explained previously in section 2.3.4, this technique can be used to improve the robot's odometry and consequently improve the precision of the overall SLAM.

This chapter will first define what is the point cloud registration problem. Then it will explore its common solutions. And finally, a general optimization method will be described to help solve variants of the common solutions.

3.1 Formulation of the problem

A general formulation of the point cloud registration problem can be expressed as [21]:

Given two point clouds $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ and $Y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_M\}$, find the transformation T^* that minimizes a distance function between the two. In other words, that aligns X to Y. They are respectively referred as the source and the target point cloud.

$$T^* = \underset{T}{\operatorname{argmin}}\ dist(T(X), Y)$$

The basic form of point cloud registration is when the correspondences between the points of the two point clouds are known. The distance function can be expressed as the sum of the squared euclidean distances between the pairs of points of the correspondences set. In our context, a transformation T can be represented by a rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ and a translation vector $\boldsymbol{t} \in \mathbb{R}^3$. Indeed, it can be assumed that both point clouds are at the same scale and so a scaling factor is not needed in the

transformation. The point cloud registration problem with a known set of correspondences $\mathcal{C}$ can be formulated as:

$$\boldsymbol{R}^*, \boldsymbol{t}^* = \underset{\boldsymbol{R},\boldsymbol{t}}{\operatorname{argmin}} \sum_{i,j \in \mathcal{C}} \|\boldsymbol{y}_j - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2 \tag{3.1}$$

This problem is often solved directly in two steps. The first translates the source point cloud so that the center of masses of both point clouds are aligned. Then the second step computes the rotation that aligns them using singular value decomposition (SVD) [22]. Fig. 3.1 shows this process in a 2D example.

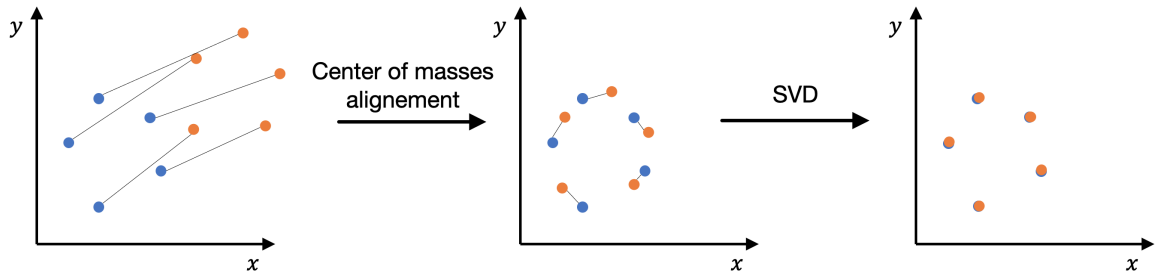


Figure 3.1: Basic form of point cloud registration

3.2 Iterative closest point

In most cases, the correspondences aren't known. This implies that the problem can't be solved with the previous method. The solution that is often taken is an iterative approach. The most widely used method is the iterative closest point (ICP) algorithm [22]. The idea of the algorithm is to estimate the correspondences during the optimization. It iterates between two steps. During the first, it estimates the correspondences between the point clouds. And the second finds the optimal transformation using the correspondences. The algorithm stops when its distance function drops below a threshold. As the name of the algorithm suggests, it estimates the correspondences using a closest point method. Algorithm 1 details the pseudocode of ICP. The problem can thus be formulated as:

$$\boldsymbol{R}^*, \boldsymbol{t}^* = \underset{\boldsymbol{R},\boldsymbol{t}}{\operatorname{argmin}} \sum_{\boldsymbol{x}_i \in X} \|\boldsymbol{p}_i - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2, \text{ with } \boldsymbol{p}_i = \underset{\boldsymbol{y} \in Y}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2 \tag{3.2}$$

During each iteration the problem returns to the problem with known correspondences. So, the same method can be used during each iteration to compute the transformation.

Algorithm 1 ICP algorithm

Input: Two point clouds $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ and $Y = \{\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_M\}$, and an initial transformation $\boldsymbol{R}_0$ and $\boldsymbol{t}_0$

Output: Optimal transformation $\boldsymbol{R}$ and $\boldsymbol{t}$ that aligns X to Y

$\boldsymbol{R}, \boldsymbol{t} \longleftarrow \boldsymbol{R}_0, \boldsymbol{t}_0$
$d \longleftarrow \infty$
while $d > threshold$ **do**
  for all $\boldsymbol{x}_i \in X$ **do**
    $\boldsymbol{p}_i \longleftarrow \underset{\boldsymbol{y} \in Y}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2$
  end for
  $\boldsymbol{R}, \boldsymbol{t} \longleftarrow \underset{\boldsymbol{R}, \boldsymbol{t}}{\operatorname{argmin}} \sum_{\boldsymbol{x}_i \in X} \|\boldsymbol{p}_i - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2$
  $d \longleftarrow \sum_{\boldsymbol{x}_i \in X} \|\boldsymbol{p}_i - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2$
end while
return $\boldsymbol{R}, \boldsymbol{t}$

As for many optimization problem, it needs an initial estimate. In our case, the odometry or the visual odometry of Spot can be used. On top of that, this algorithm can fall into local minima, and a better initialization can help avoid it.

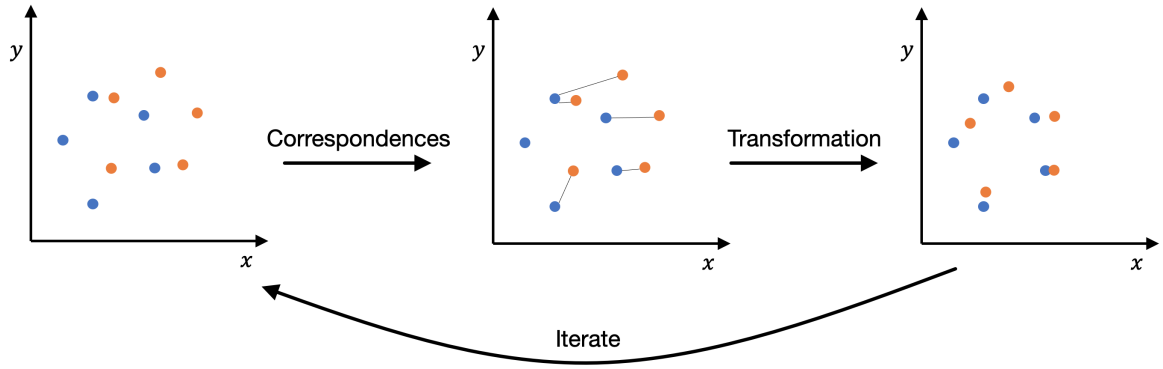


Figure 3.2: Iterative closest point process in a 2D example

3.3 Improvements of ICP

Several improvements can be made to ICP. Those can increase its precision and its convergence speed. The first improvement that can be made is to add weights to the correspondences. A LiDAR is not perfect and there is noise in its measurements. So, a bigger weight can be added to the more certain correspondences. Furthermore, the effect of outliers can be removed by assigning them a zero weight.

The problem can thus be formulated as:

$$\boldsymbol{R}^*, \boldsymbol{t}^* = \underset{\boldsymbol{R},\boldsymbol{t}}{\operatorname{argmin}} \sum_{\boldsymbol{x}_i \in X} w_i \|\boldsymbol{p}_i - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2 \,, \text{ with } \boldsymbol{p}_i = \underset{\boldsymbol{y} \in Y}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2, \tag{3.3}$$

where w_i is the weight associated to the point $\boldsymbol{x}_i$ of X.

There are many different ways to compute weights. Most of them use a function that outputs a number between 0 and 1 based on the correspondence's distance. Popular ones are the Huber loss or the Tukey loss [23].

Another improvement that has been widely used is the "point-to-plane" ICP [24]. The idea is to use the fact that point clouds often represent surfaces. So, instead of aligning the points between them, "point-to-plane" ICP tries to align the source point cloud to the underlying target surface. But the surface of the target point cloud is not accessible so it is approximated using estimated normals. To do that, the distances can be projected to the normals of the surface. The normals can be estimated using the neighborhood of the points [25]. Equation 3.4 shows the mathematical formulation of "point-to-plane" ICP. "Point-to-plane" ICP can be formulated as follow:

$$\boldsymbol{R}^*, \boldsymbol{t}^* = \underset{\boldsymbol{R},\boldsymbol{t}}{\operatorname{argmin}} \sum_{\boldsymbol{x}_i \in X} w_i (\boldsymbol{n}_i \cdot (\boldsymbol{p}_i - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t})^2) \,, \text{ with } \boldsymbol{p}_i = \underset{\boldsymbol{y} \in Y}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{R}\boldsymbol{x}_i - \boldsymbol{t}\|_2^2, \tag{3.4}$$

where n_i is the normal of the closest point $\boldsymbol{p}_i$ of $\boldsymbol{x}_i$.

It has been shown that this new distance metric outperforms the regular one in terms of convergence speed and precision of the final solution [26]. There is even a generalized version of ICP that take into account the surface structure from both point clouds instead of only the target point cloud [27].

With the added improvements, the same optimization method as in classical ICP can't be used. To solve the variants of ICP, a more general optimization approach must be used.

3.4 Gauss-Newton algorithm

As explained previously, The method used in ICP can't be used to minimize the modified distance functions. A more general optimization approach is needed. The solve this issue, the optimization problems can be formulated as non-linear least-square problems.

A non-linear least-square problem [28] can be defined as the problem of minimizing the sum of

non-linear squared functions of several variables

$$\min \sum_{i=1}^{m} r_i(\boldsymbol{\theta})^2 = \min \boldsymbol{r}(\boldsymbol{\theta})^T \boldsymbol{r}(\boldsymbol{\theta}), \tag{3.5}$$

where r_i is a non-linear functions, also called residual, $\boldsymbol{\theta} \in \mathbb{R}^n$ are the n variables of those functions, and $\boldsymbol{r}(\boldsymbol{\theta}) = [\, r_1(\boldsymbol{\theta}),\, r_2(\boldsymbol{\theta}),\, \ldots,\, r_m(\boldsymbol{\theta})\,]^T$.

One of the most used methods to solve this kind of problem is the Gauss-Newton algorithm [28]. The idea is to linearize the residuals using the first-order Taylor series expansion around the current guess of the solution $\tilde{\boldsymbol{\theta}}$ and then solve the problem as a linear least square problem.

The first order Taylor series expansion of the residual r_i around $\tilde{\boldsymbol{\theta}}$ is:

$$r_i(\boldsymbol{\theta}) \approx r_i(\tilde{\boldsymbol{\theta}}) + \sum_{j=1}^{n} \frac{\partial r_i(\tilde{\boldsymbol{\theta}})}{\partial \theta_j} (\boldsymbol{\theta} - \tilde{\boldsymbol{\theta}}). \tag{3.6}$$

Equation 3.6 can be generalized for $\boldsymbol{r}$ as:

$$\boldsymbol{r}(\boldsymbol{\theta}) \approx \boldsymbol{r}(\tilde{\boldsymbol{\theta}}) + \boldsymbol{J}(\boldsymbol{\theta} - \tilde{\boldsymbol{\theta}}), \tag{3.7}$$

where $\boldsymbol{J}$ is the jacobian matrix which is composed of $J_{ij} = \frac{\partial r_i(\tilde{\boldsymbol{\theta}})}{\partial \theta_j}$.

By replacing this approximation into the minimization problem, its first order derivative can be set to 0 to find the optimal value of $\boldsymbol{\theta}$:

$$\frac{\partial}{\partial \boldsymbol{\theta}} ((\boldsymbol{r}(\tilde{\boldsymbol{\theta}}) + \boldsymbol{J}(\boldsymbol{\theta} - \tilde{\boldsymbol{\theta}})) \cdot (\boldsymbol{r}(\tilde{\boldsymbol{\theta}}) + \boldsymbol{J}(\boldsymbol{\theta} - \tilde{\boldsymbol{\theta}})) = 2\boldsymbol{J}^T(\boldsymbol{r}(\tilde{\boldsymbol{\theta}}) + \boldsymbol{J}(\boldsymbol{\theta} - \tilde{\boldsymbol{\theta}})) = 0. \tag{3.8}$$

Equation 3.8 can be rearranged to find that:

$$\boldsymbol{\theta} = \tilde{\boldsymbol{\theta}} - (\boldsymbol{J}^T \boldsymbol{J})^{-1} \boldsymbol{J}^T \boldsymbol{r}(\tilde{\boldsymbol{\theta}}). \tag{3.9}$$

The process can be iterated using the value of $\boldsymbol{\theta}$ computed in equation 3.11 as the new guess of the variables until convergence.

3.4.1 Weights in Gauss-Newton algorithm

As explained is section 3.3, weights can be added to attenuate the impact of outliers on the optimization. Equation 3.10 becomes:

$$\min \sum_{i=1}^{m} w_i r_i(\boldsymbol{\theta})^2 = \min \boldsymbol{r}(\boldsymbol{\theta})^T \boldsymbol{W} \boldsymbol{r}(\boldsymbol{\theta}), \tag{3.10}$$

where w_i is the weight associated to the residual r_i and $\boldsymbol{W}$ is the diagonal matrix of weights. Following the same development as in the previous section the final iteration equation is expressed as:

$$\boldsymbol{x} = \tilde{\boldsymbol{\theta}} - (\boldsymbol{J}^T \boldsymbol{W} \boldsymbol{J})^{-1} \boldsymbol{J}^T \boldsymbol{W} \boldsymbol{r}(\tilde{\boldsymbol{\theta}}). \tag{3.11}$$

Chapter 4

Graph-SLAM implementation

This chapter will explore in detail our implementation of Graph-SLAM. As explained previously, the approach can be divided into two parts, the graph generation and the graph optimization. As the graph generation is more specific to the context than the graph optimization, most of this chapter is devoted to this part. Finally, the method used for the graph optimization will be discussed. The results of this implementation will be discussed in chapter 7.

4.1 Graph generation

The process starts with a first node initialized at the origin of the graph. It will serve as a reference for the construction of the graph. Each pose $\boldsymbol{X}_t$ associated to the node t can be represented by a transformation matrix $\boldsymbol{X}_t \in \mathbb{R}^{4\times4}$. Each pose thus represents the transformation between the robot's body and the reference of the graph $\boldsymbol{X}_0$. The first node being at the origin, its pose is the identity matrix $\boldsymbol{X}_0 = \boldsymbol{I} \in \mathbb{R}^{4\times4}$.

When the robot moves, a new node can be added to the graph. The pose of the node can be estimated with the robot's odometry. As the odometry and the graph are not expressed in the same reference frame, the odometry can't be used directly. Instead, the estimated displacement between the nodes can be computed and used to estimate the new node's pose. The new node pose can be estimated as

$$\boldsymbol{X}_t = \boldsymbol{X}_{t-1}((\boldsymbol{X}^R_{W,t-1})^{-1}\boldsymbol{X}^R_{W,t}), \tag{4.1}$$

where $\boldsymbol{X}^R_{W,t}$ is the robot's odometry taken at the same time as node t that expresses the pose of the robot with respect to the world origin frame W of the robot and $((\boldsymbol{X}^R_{W,t-1})^{-1}\boldsymbol{X}^R_{W,t})$ is the displacement between the two nodes.

But this new pose is only an estimation computed from the odometry. And as explained in the section 2.3.4, there are uncertainties in the odometry. Since the precision of the graph optimization depends on the uncertainties of the edges of the graph, the point clouds $\mathcal{P}_t = \{\boldsymbol{x} \in \mathbb{R}^3\}$ taken at the same times as node t can be used to improve this odometry. This is called LiDAR odometry. As a reminder, two point clouds taken successively will represent mostly the same geometric structure. Therefore, the two point cloud can be aligned to recover the transformation between the poses of both. This process can be visualized in Fig. 4.1.

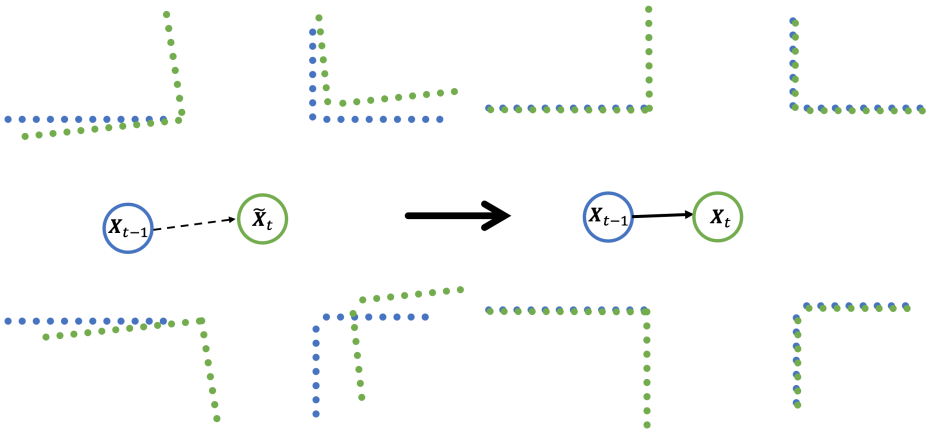


Figure 4.1: Visualization of the principle of LiDAR odometry. The LiDAR is displaced from a pose $\boldsymbol{X}_{t-1}$. The new pose of the LiDAR is first estimated with the robot odometry. But the estimation $\tilde{\boldsymbol{X}}_t$ is not precise enough. The estimation can be refined by aligning the point cloud of the new node (green) to the point cloud of the preceding node (blue) with a registration algorithm initialized with the estimation of the displacement.

But the point clouds are expressed in the frame of the LiDAR $\{L\}$. To avoid changing the frame of reference of the point clouds constantly, the graph's nodes will represent the LiDAR's poses instead of the robot's poses. The two are equivalent as the LiDAR is fixed to the robot. The transformation between the robot's pose to the LiDAR's pose can be expressed as the transformation matrix $\boldsymbol{T}_{R\to L} \in \mathbb{R}^{4\times4}$. So the odometry can be transformed to express the estimated pose of the LiDAR with respect to the world origin of the robot $\boldsymbol{X}^L_{W,t}$ with the expression:

$$\boldsymbol{X}^L_{W,t} = \boldsymbol{X}^R_{W,t}\boldsymbol{T}_{R\to L}. \tag{4.2}$$

And $\boldsymbol{X}^L_{W,t}$ can be used to replace $\boldsymbol{X}^R_{W,t}$ in equation 4.1. At any point in time, a node's pose can be transformed to represent the robot's pose using the inverse transformation: $\boldsymbol{X}_t\boldsymbol{T}^{-1}_{R\to L}$.

Before explaining the graph generation process in detail here is a quick overview of the process.

The process starts with a node associated with the pose $\boldsymbol{X}_0 = \boldsymbol{I} \in \mathbb{R}^{4\times4}$ that will be used as a reference for the graph generation. When the robot moves, the process uses the new point cloud $\mathcal{P}_t$ and new odometry $\boldsymbol{X}^L_{W,t}$ in the following processing steps:

1. The estimated displacement between node $t-1$ and node t is computed with the expression $((\boldsymbol{X}^L_{W,t-1})^{-1}\boldsymbol{X}^L_{W,t})$. It is then used as a first estimation of the next node's pose $\tilde{\boldsymbol{X}}_t = \boldsymbol{X}_{t-1}((\boldsymbol{X}^L_{W,t-1})^{-1}\boldsymbol{X}^L_{W,t})$
2. The point cloud $\mathcal{P}_t$ can then be used to update this pose using a point cloud registration algorithm to align it to the point clouds of the past nodes. This process is called LiDAR odometry. This new estimation can then be used as the pose of the new node. The point cloud and the odometry are saved for the later computations.
3. When a new node is added, the loop closure process searches through the nodes of the graph to find a loop closure candidate. If a loop can be closed then an edge is added between the two nodes and the graph is updated with the graph optimization process.

4.1.1 LiDAR odometry

For the LiDAR odometry process, we will assume that the process has already started and can be seen in Fig. 4.2. A new node t must be added to the graph. As explained previously, the first estimation of the node's pose $\tilde{\boldsymbol{X}}_t$ can be computed with the robot's odometry. Now the idea is to improve this estimation by using the new point cloud $\mathcal{P}_t$.

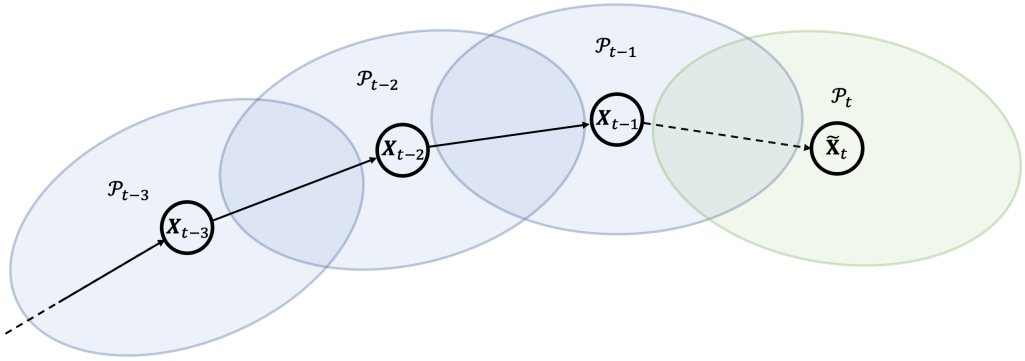


Figure 4.2: Example of the graph's state when a new node t is added but before the LiDAR odometry is executed. The point clouds associated to each node is represented as the ellipses.

The straightforward method is to align the point cloud $\mathcal{P}_t$ with the last point cloud $\mathcal{P}_{t-1}$ with a point cloud registration algorithm initialized with the displacement estimation. Any variant of ICP could be used in this process. The transformation that aligns the point cloud $\mathcal{P}_t$ to $\mathcal{P}_{t-1}$ can then be used to update the first estimation of the node's pose $\tilde{\boldsymbol{X}}_t$ into a better estimate $\boldsymbol{X}_t$.

But there are some issues with this technique. The number of points on the ground in both point clouds is usually small compared to the total number of points. Fig. 4.3 shows this problem in 2 different scenarios. This is a consequence of the positioning of the LiDAR on Spot, its characteristics, and the environment in which the robot navigates. Due to the small field of view of the LiDAR and its high position compared to the ground, the closest point to Spot on the ground is several meters away. Furthermore, its vertical angular resolution spaces the "lines" far away from each other. It creates an error in the normals estimation of the ground. Fig. 4.4 shows well that the normals estimations for the point on the ground are not perpendicular to the ground plane. Also, there are often tight spaces like small rooms or corridors in buildings. Hence, there are even fewer points on the ground. With the low number of ground points and the wrong estimation of the normals, it is hard for all variants of ICP to estimate the vertical positioning and orientation of the point clouds.

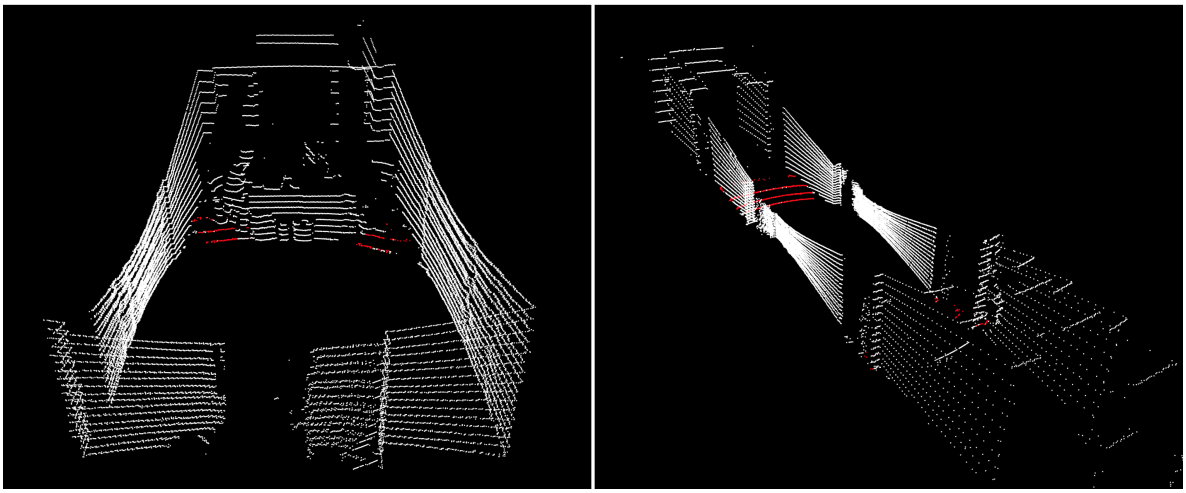


Figure 4.3: Ground points depicted in red in an office (left) and a corridor (right).

A solution is to concatenate multiple point clouds together to increase the number of points on the ground and improve the estimation of the normals. So instead of only using the last point clouds, the n last ones can be concatenated and used as the target point cloud for aligning $\mathcal{P}_t$. To create the concatenated point cloud, the point clouds can be transformed following the node's pose and fused to create a consistent point cloud. This method helps the estimation of the normals and improves the overall accuracy of the displacement estimation.

But this solution has still some issues. The first one is that ICP uses all points for the optimization.

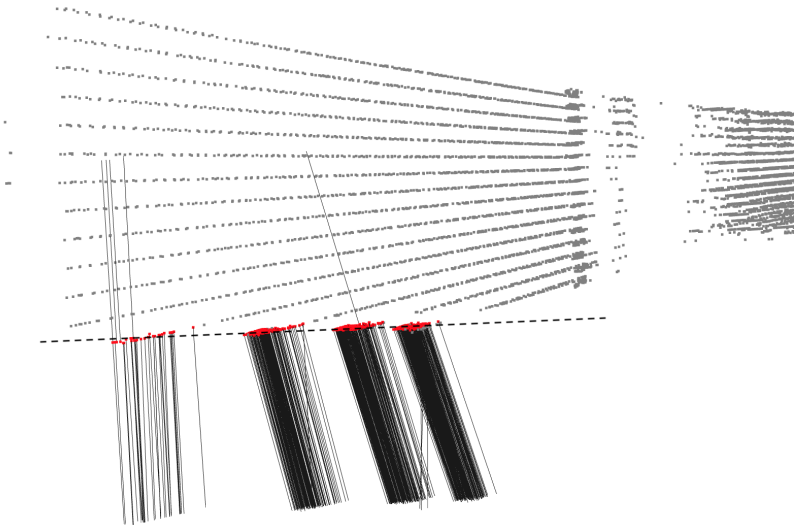


Figure 4.4: Estimated normals of the ground point in red. Due to the lack of points, they are not perpendicular to the ground plane.

While it uses all the information of the point cloud, it impacts the computation time. As explained earlier, ICP finds correspondences for all the points of the point cloud. This process is expensive. But in our context, we must map the environment in real-time, so the LiDAR odometry must be executed several times per second. It is even a problem for our LiDAR, which has a low resolution compared to other popular LiDARs. Furthermore, for the "point-to-plane" ICP and Generalized ICP, the normal estimation must be computed for every point of the target point clouds, which can be computation expensive. The second problem is that even if the variants of ICP use the entire point cloud's information, it doesn't use it efficiently. The variants of ICP use the normals to improve the convergence speed, but even with more points, the normal estimation can be wrong. As an example, the normal of the points that fall on the edge of two planes will be wrongfully estimated and we can't assume that these points represent flat surfaces. But all exposed variants of ICP assume it which leads to errors in the alignment.

An idea to solve all these problems is to filter the points. By keeping only the important ones, the convergence speed can be improved and even the result of the optimization [26]. With the selected features, the computation of the correspondences can be changed to better use the information of the point clouds.

4.1.1.1 Feature selection

In our context, the environment in which the robot moves is a construction site. As in most buildings, we can assume that it is composed of planar surfaces. We can take advantage of this by using the method developed in the LOAM algorithm [29]. The idea is to select points that come from the underlying planar surfaces and edges of the point cloud. LOAM stands for Lidar Odometry and Mapping and is one of the best odometry methods on the KITTI dataset [30]. This dataset is one of the most popular benchmarks for odometry estimation.

The feature points can be extracted by computing a roughness value for each point. A small roughness means the point is on a plane while a large one means that it lies on an edge. The roughness can be estimated using the neighborhood of a point. But this process can take time. Especially as the roughness needs to be computed for every point of the point cloud.

The LiDAR properties can be used to approximate the neighborhood. The LiDAR has a higher horizontal angular resolution than its vertical. The points that are next to the point in the same channel can be used to approximate the neighborhood efficiently. Unfortunately, the point clouds received from Spot are not separated in channels. But they can be separated by using the angles of the points compared to the LiDAR orientation. This follows the same idea as LeGO-LOAM [31], where it transforms the point clouds into images based on the angles and the ranges. Computing the neighborhood of a point means searching for the point's neighboring indices in the same channel list, which is quite efficient.

The roughness of $\boldsymbol{x}_{k,i}$, the i-th point of channel k of a point cloud $\mathcal{P}$, with a neighborhood of size $2n$ can thus be computed using the formula:

$$r_{k,i} = \| \sum_{j=i-n,\ i \neq j}^{i+n} (\boldsymbol{x}_{k,i} - \boldsymbol{x}_{k,j}) \|_2^2. \tag{4.3}$$

Like in the paper, a neighborhood of size 10 is chosen.

This equation computes the squared norm of sums of the vectors that go from the neighboring points to the i-th point. When the points are mostly aligned, vectors will tend to cancel each other as opposed to when the points are not aligned. To give a more visual sense to this equation, Fig. 4.5 shows the sum of vectors for points that fall on a surface and an edge.

The points in each channel are sorted according to their roughness and the ones with the highest roughness values are taken as edges points and the ones with the smallest for the planar points. To select points evenly around the robot, each channel is divided into 6 equal parts [31]. From each part, a maximum of 20 edge points and a maximum of 40 planar points are selected per channel [31].

A point near an edge point will also have a high roughness value. And to avoid wrongfully picking

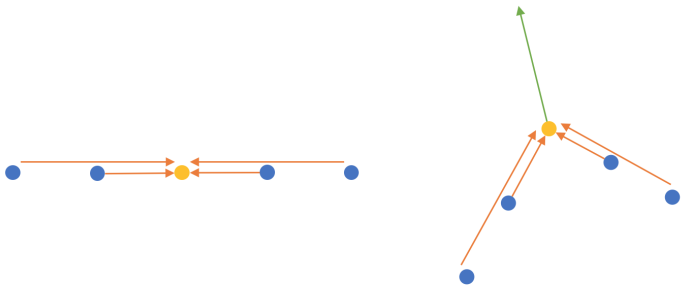


Figure 4.5: Roughness vector (in green) for a point (in orange) that falls on a planar surface (left) and for a point that falls on an edge (right). On the left, the vectors cancel each other perfectly.

it, the points that are close to already selected points are never selected. This also helps to distribute the points across the point clouds.

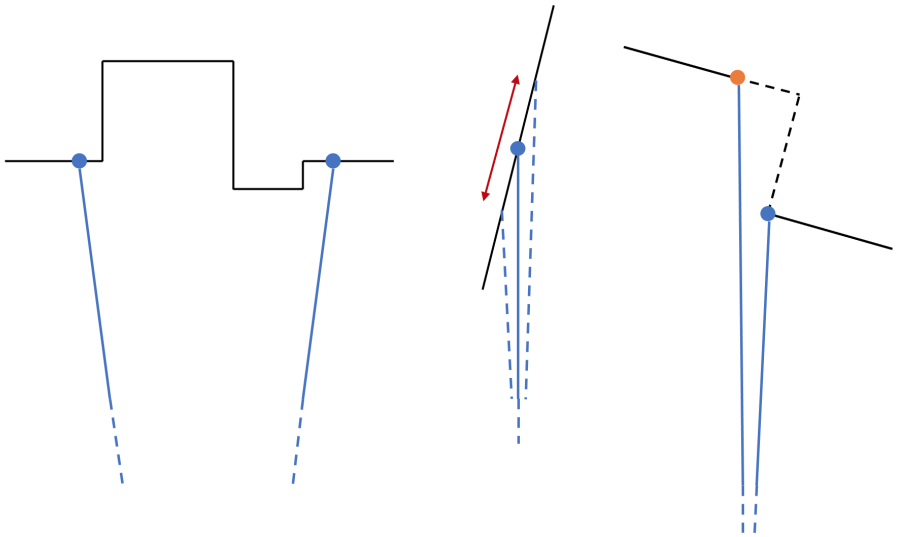


Figure 4.6: Unreliable points. On the left, the points are far from the LiDAR and don't hold enough information about the surface. In the center, The surface is nearly parallel to the laser and a small deviation in the direction creates a large error in the estimation of the range. On the right of the image, the orange point must not be taken as an edge because the LiDAR might move and discover unseen space.

Some points can be considered as unreliable and can be removed from the selection [29]:

- Points that are too far or too close to the LiDAR. The ones too close might be linked to the LiDAR "seeing" the robot. And the neighborhood of the points that are too far doesn't hold information about the underlying point's surface due to the LiDAR's resolution (shown on the left of Fig. 4.6).
- Points on a surface that is nearly parallel to the LiDAR's laser. This is because a slight error in the direction of the laser leads to a large variation in the estimated range (shown in the center of Fig. 4.6).
- Points on the frontier of an occluded area can be wrongfully picked as edge points (shown on the right of Fig. 4.6).

Fig. 4.7 shows an example of features on the point cloud of a room. The edges are shown in red and the planar point in blue. This process is not perfect, especially for edge points. The first cause is because of the resolution of the LiDAR. Some edge points might not fall exactly on the edge of surfaces. And the second is because of the way the neighborhood is estimated. Some points that have a high roughness might not be edge points at all. Outlier rejection and specific feature matching techniques can help to mitigate those problems and reduce the weight of wrong edge points during optimization.

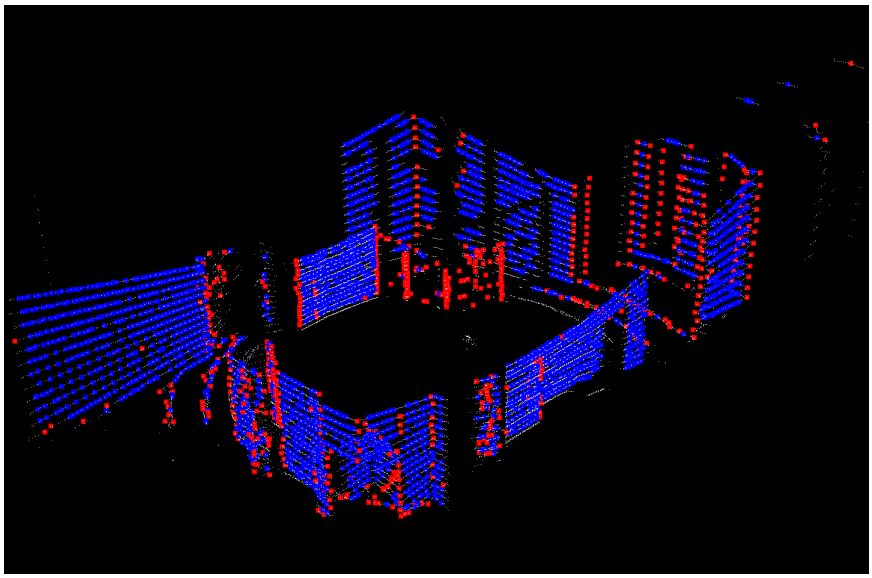


Figure 4.7: Selected feature points in a point cloud. In red the edges and in blue the points on planar surfaces

4.1.1.2 Feature matching

The goal of estimating the correspondences in ICP is to calculate the distance between the two point clouds. LOAM leverages the features to build a distance function that takes more into account the surfaces of the point clouds. The idea is to compute the distance between the feature points of the source point cloud and the closest surface feature of the target point cloud. For the edge points in the source point cloud, the distance is computed using the closest edge in the target point cloud. And the same principle is used for the planar points but with the nearest plane.

Let $\mathcal{F}_e^s$ and $\mathcal{F}_e^t$ be the set of edge points from the source and the target point clouds. Let $\boldsymbol{x}_e \in \mathbb{R}^3$ be the points of $\mathcal{F}_e^s$ for which the distance to the target point cloud should be computed. As explained earlier, the distance function is estimated at each iteration of the process. So, the actual point used is the transformed point $\tilde{\boldsymbol{x}}_e = \boldsymbol{R}\boldsymbol{x}_e + \boldsymbol{t}$. Where $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ and $\boldsymbol{t} \in \mathbb{R}^3$ are the estimated rotation matrix and the translation vector during the iteration. As this is an edge point, its distance to the point cloud is the distance to the closest edge in the point cloud. The edge can be estimated by extracting the closest points in $\mathcal{F}_e^t$ from $\tilde{\boldsymbol{x}}_e$. As in the LeGoLOAM [31] implementation, five points are selected. The extracted patch of points is an edge if one of the eigenvalues of its covariance matrix is significantly larger than the two other [29]. If the patch satisfies this criterion, the direction of the line is the eigenvector of the largest eigenvalue and its position is the mean position of the patch of points.

Both information can be used directly to compute the distance but to facilitate the computation of the optimization it is computed differently. From the direction and the position of the line, two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ can be generated. The distance to the line can finally be expressed as [29]:

$$d_e(\tilde{\boldsymbol{x}}_e) = \frac{\|(\tilde{\boldsymbol{x}}_e - \boldsymbol{x}_1) \times (\tilde{\boldsymbol{x}}_e - \boldsymbol{x}_2)\|}{\|(\boldsymbol{x}_1 - \boldsymbol{x}_2)\|} \tag{4.4}$$

The technique is similar for the planar points. Let $\mathcal{F}_p^s$ and $\mathcal{F}_p^t$ be the set of planar points from the source and the target point clouds. As for the edge points, we have to use the transformed version $\tilde{\boldsymbol{x}}_p$ of the point $\boldsymbol{x}_p$. For planar points, the distance is the distance to the closest plane in the target point cloud. To estimate the closest plane, the closest points of $\tilde{\boldsymbol{x}}_p$ in $\mathcal{F}_e^t$ are extracted. If a plane can be fitted with a low error on this patch of points, then the patch of points lies on a flat surface. A plane is fitted on the points by solving a linear least square problem. For each point used to compute the plane, its distance to the plane must be lower than a threshold. This ensures that only the patches of points that represent real planes are used. Using the same idea as for the edge points, the distance to the surface can be computed by extracting 3 points $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$ from the plane and using the formula [29]:

$$d_p(\tilde{\boldsymbol{x}}_p) = \frac{\|(\tilde{\boldsymbol{x}}_p - \boldsymbol{x}_i) \cdot ((\tilde{\boldsymbol{x}}_1 - \boldsymbol{x}_2) \times (\tilde{\boldsymbol{x}}_1 - \boldsymbol{x}_3))\|}{\|(\tilde{\boldsymbol{x}}_1 - \boldsymbol{x}_2) \times (\tilde{\boldsymbol{x}}_1 - \boldsymbol{x}_3)\|} \tag{4.5}$$

Using those two distances, the distance function can be formulated using the distance for all edge points and planar points of the source point cloud [29]. The problem can then be formulated as:

$$\boldsymbol{R}^*, \boldsymbol{t}^* = \underset{\boldsymbol{R},\boldsymbol{t}}{\operatorname{argmin}} \sum_{\boldsymbol{x}_e \in E_s} d_e(\boldsymbol{R}\boldsymbol{x}_e + \boldsymbol{t})^2 + \sum_{\boldsymbol{x}_p \in P_s} d_p(\boldsymbol{R}\boldsymbol{x}_p + \boldsymbol{t})^2 \tag{4.6}$$

As for the variants of ICP, this minimization problem can be formulated as a non-linear least square problem and the Gauss-Newton algorithm can be used to optimize it.

There could be some outliers in the distances' estimations. As explained in section 4.1.1.1, the process of selecting edge points is not perfect and some wrong edge distances could be estimated. Also, some regions that weren't visible from the old LiDAR's point of view could be revealed when the robot moves. This means that points don't necessarily have a correspondence in the old point cloud. The outliers can have a huge effect on the optimization and to reduce it we can weigh the distances and use the optimization method developed in section 3.4.1. The idea would be to associate a lower weight to points that have larger distances and to assign a zero weight to the points that are farther than a threshold. For that purpose, we chose the Tuckey loss function [23] which combines these two properties. The weights can be computed with the formula:

$$w(d) = \begin{cases} (1 - (\frac{d}{k})^2)^2 & \text{if } |d| \leq k \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

where k is the threshold distance. The value of k will be set depending of the context of use of this algorithm.

4.1.1.3 Overall process of the sequential graph generation

Using the point cloud registration algorithm developed, the process of creating the sequential nodes and edges of the graph can be summarized as follow.

When a new point cloud $\mathcal{P}_t$ is received it is first split into two feature set $\mathcal{F}_e$ and $\mathcal{F}_p$ representing edge points and planar points. The target edge feature set $\mathcal{F}_e^t$ can be created by fusing the last n saved edge feature sets. For the implementation, n was set to 40. The same can be done for the target planar feature set $\mathcal{F}_p^t$. To distribute evenly the points in the target sets, only one point is kept for each 5cm $\times$ 5cm $\times$ 5cm region of the target edge set and for each 10cm $\times$ 10cm $\times$ 10cm region of the target planar set. The point cloud registration algorithm detailed above can then be used to better estimate the displacement of the robot by aligning the two feature set of $\mathcal{P}_t$ to the target features sets. The initialization of the algorithm is done using the odometry displacement. The optimized displacement is then used to estimate the new pose of the node and the feature sets $\mathcal{F}_e$ and $\mathcal{F}_p$ are saved for future

computations.

For the parameters of this process, we chose to set the distance threshold k to 5 cm. The distance threshold can be set so low because the robot's odometry is already a good estimate of the displacement and the registration algorithm therefore only makes some adjustments. The low distance threshold also helps to reduce the impact of the dynamic objects in the environment. As dynamic objects move around the robot, the points related to them in successive point clouds will not be in the same positions. Hence a low distance threshold helps remove the effects of these points.

The total number of nodes in the graph can increase very quickly because the robot can send many point clouds per second. Therefore, instead of creating a new node every time the robot moves, we can add a new node when the robot has moved a certain distance from the last saved node. In our implementation, we have chosen a distance of 20 cm. Furthermore, it is not useful to save every point cloud because point clouds taken with a small delay between them will overlap by a lot.

One question one might ask is why take only the last n point clouds as targets instead of all the point clouds that are close to the new pose in the graph? This is due to the drift caused by the displacement estimates. Fig. 4.8 illustrates this problem. In a situation where the robot has traveled a long way without returning to a previously seen location, there will be drift in its pose estimate. If the robot returns to a previously seen area and the target point cloud is constructed using the point clouds around the robot's current pose, there will be ambiguities in the point cloud. As shown in Fig. 4.8, the walls can be doubled, which makes it difficult for the registration algorithm to find the optimal transformation. This problem can be partially solved by using only recent point clouds. It removes the ambiguities but the drift in the pose estimation is not corrected.

4.1.2 Loop closure

As a reminder, the loop closure process tries to link together nodes that are not sequential and that are close to each other in the real environment. This process closes loops inside the graph. It helps to reduce the overall error of the graph. In our context, the robot is moving through construction sites. In buildings, the large majority of loop closure will be small as the likelihood of the robot taking a long path without ever going back to an already seen place is low. As it can be assumed that the drift is small for a small path, we can check for loop closure directly in the surroundings of the current pose.

When a new node is created, all the nodes in a small sphere centered at the new node are extracted as pictured in Fig. 4.9. In our implementation, the sphere has a radius of 0.5 m. Inside the sphere, we will only consider the nodes that are old enough compared to the new ones to avoid closing loops with nodes that were just added to the graph. A node can be considered old if it is no longer part of the recent set used for the LiDAR odometry. Selecting the oldest one ensures that the current path is corrected with the path that has the lowest overall drift.

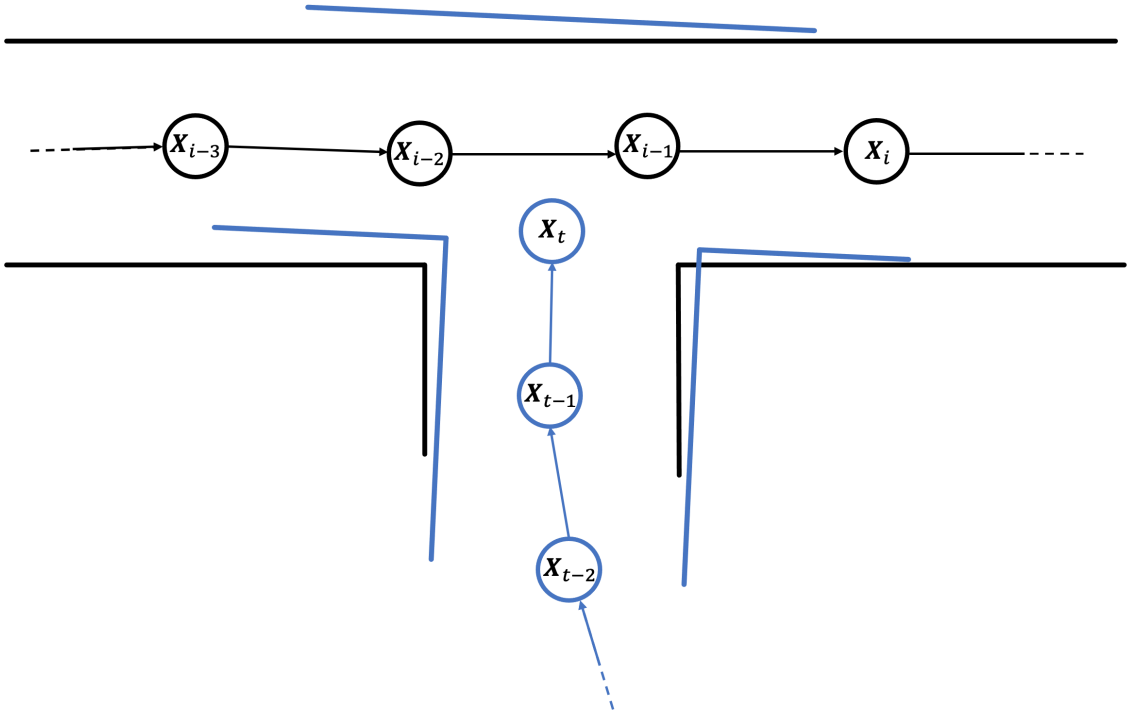


Figure 4.8: Visualization of the ambiguities caused by the drift in the pose estimation along a path.

If a candidate is found around the new node, then we can correct the graph by putting an edge between the two that represent the real transformation between them. To compute it, the same principle as the LiDAR odometry can be used. The real transformation can be estimated using a point cloud registration algorithm. In the same way as in the LiDAR odometry, the point cloud of the new node can be aligned to the point cloud of the candidate node. For the same reason as in section 4.1.1, the target point cloud can be extracted by selecting point clouds that are on the same path as the candidate node. In our implementation, we select the 20 before and after the candidate.

The error in the graph prevents the use of the registration algorithm developed in section 4.1.1. Instead, ICP can be used to find a first estimate of the real transformation between the nodes. The convergence of ICP can even be used as a criterion to know if a loop can be closed between the two nodes. If the mean of the squared euclidean distances between the two point clouds is lower than a threshold then the loop closure is accepted. To refine the estimated real transformation, the algorithm developed in 4.1.1 can be used after ICP. The distance threshold of the algorithm must be put at a higher value than for the LiDAR odometry as the initialization of the registration algorithm is not as good. We set this threshold to $0.1m$.

When a loop closure has been found and the displacement between the node is correctly estimated,

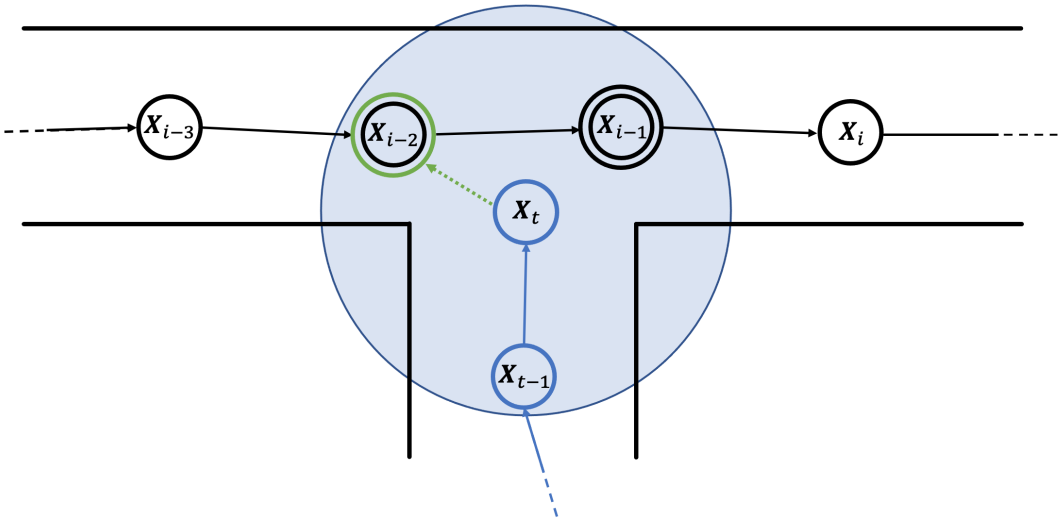


Figure 4.9: Visualization of the candidate selection process for the loop-closure. A sphere is created around the robot to take only the surrounding nodes that are close enough to be considered as loop-closure. The nodes circled are considered as candidate. The oldest one is taken as the loop-closure candidate.

an edge can be added to the graph. The graph optimization process can then optimize the graph and update the poses of the nodes. As the process uses ICP and our algorithm in sequence, it is done in another thread to avoid impacting the LiDAR odometry computation time.

This process works well with the assumption that only small loops will be closed. If the robot performs a large path without going back to already seen places, then another technique must be used to find loop closure in the graph. For example, AprilTags can be used to detect loop closure. The same process using ICP and the registration algorithm developed in 4.1.1 can be used with the initialization of the process set as the detected transformation to the AprilTag.

It is important to note that the more loop-closure performed on the robot path, the more consistent the map will be with the reality. Especially in maps made on several floors. Moreover, for the floors to be correctly oriented and for the planes they form to be parallel, there must be at least three different access points for each floor. Fig. 4.10 shows that 2 floors may not be parallel if there are not enough different access points. This problem comes from the fact that the minimal velocity of the robot is quite high on stairs which includes mistakes in the distortion estimation done by Spot. These errors in the undistorted point clouds impact negatively the LiDAR odometry.

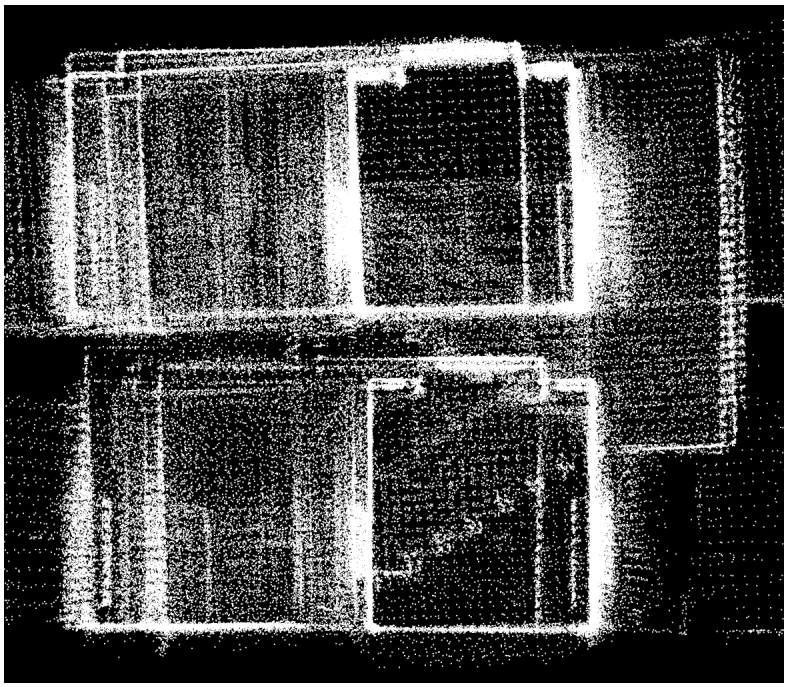
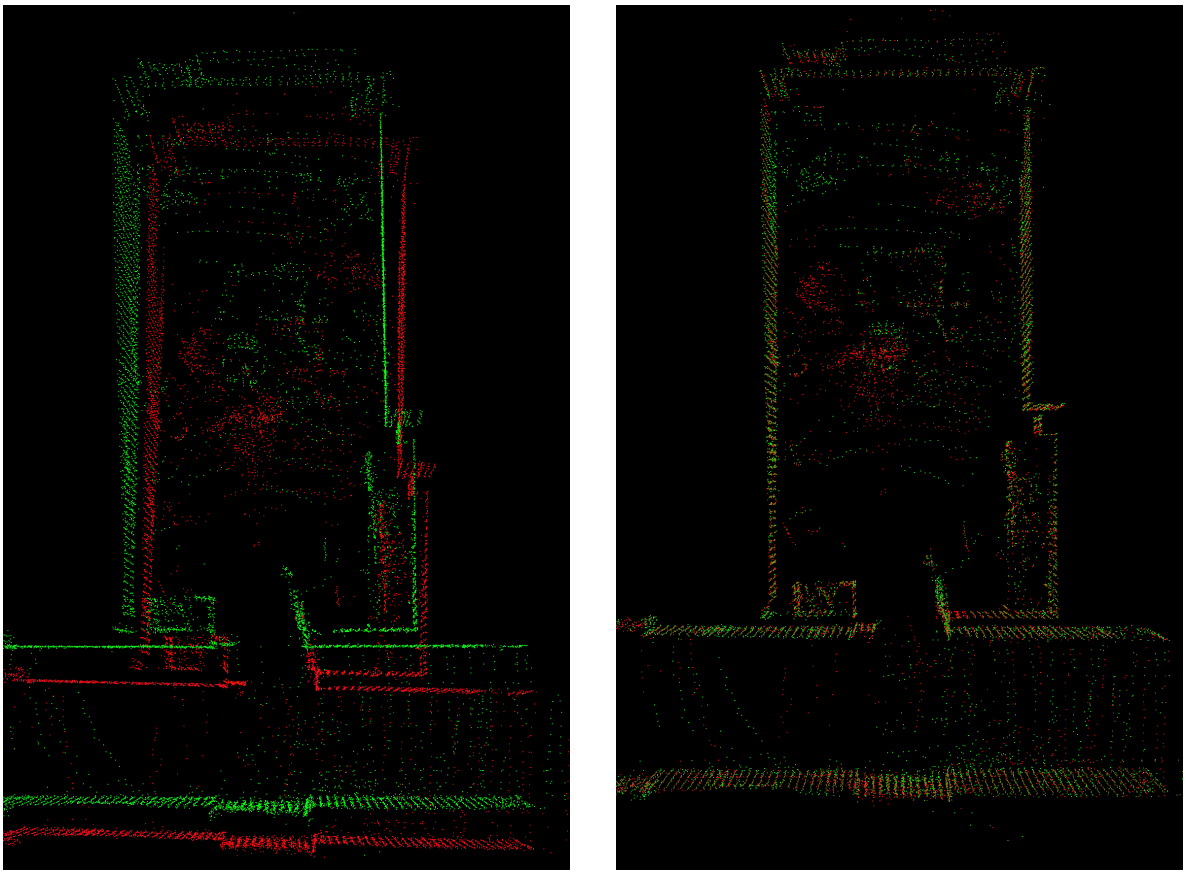


Figure 4.10: Side view of a the map of two floors. As there are not enough different access points between the two, the error due to the high speed of the robot on stairs could not be corrected by the loop-closure process and the 2 floors are not exactly parallel.

4.2 Graph optimization

To represent the graph and perform the graph optimization, we use the GTSAM [32] library. GTSAM is a specialized C++ library that implements efficient algorithms for graph optimization. The actual algorithm we used for the graph optimization is the ISAM2 [16] algorithm. The idea of ISAM2 is to use efficient data structures developed specifically for graph optimization to optimize efficiently the problem stated in section 2.3.3. This algorithm is also incremental. It means that it uses variable reordering to optimize the graph based on the past optimization. Its efficiency and its incremental property allow us to execute the graph optimization every time a loop is closed in the graph.

The final map of the graph can be extracted from the node of the graph. All point clouds saved can be concatenated inside a single point cloud by transforming each one to its associated node pose. Examples of maps extracted with this algorithm are shown in Fig. 4.12 and 4.13.



(a) Before loop closure

(b) After loop closure

Figure 4.11: Comparison between the Graph-SLAM process before and after loop closure. It can be seen that the loop closure process correctly closes the loop of the path. The start of the path is represented with the red point cloud and the end with the red one.

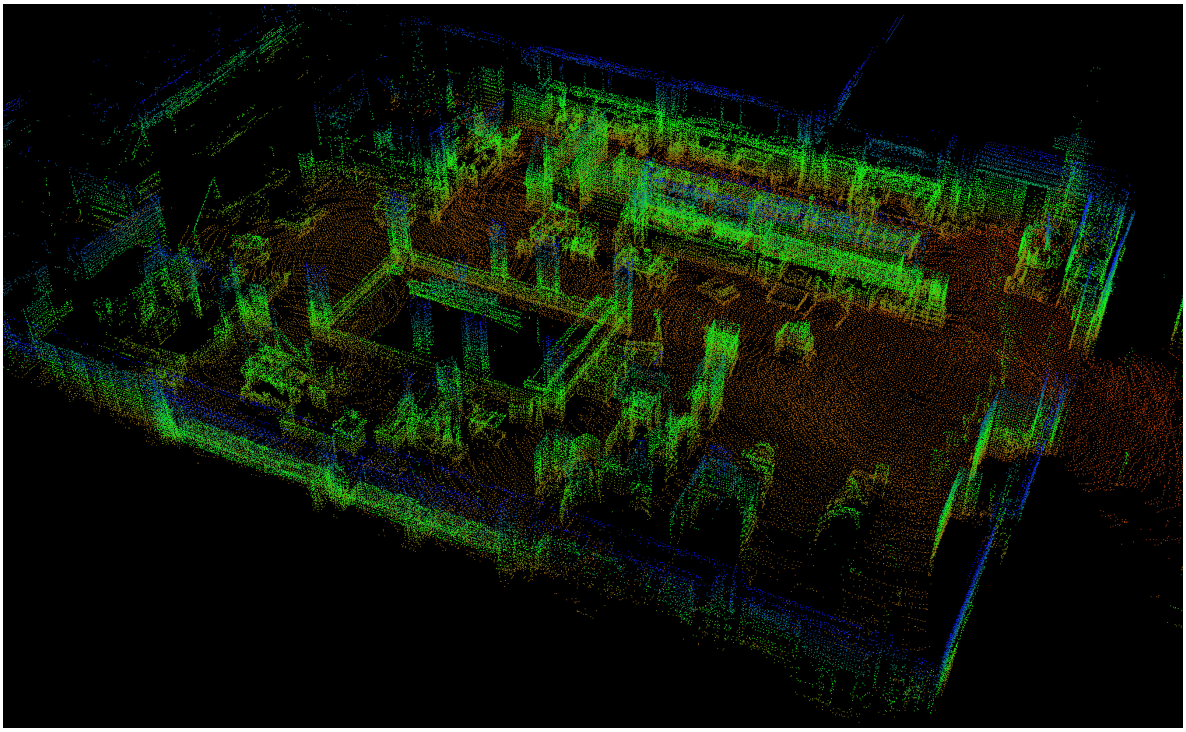


Figure 4.12: Map of a testing hall computed with the Graph-SLAM algorithm. The details of the environment are well preserved and nothing is duplicated which could happen when using only the odometries as a mapping solution.

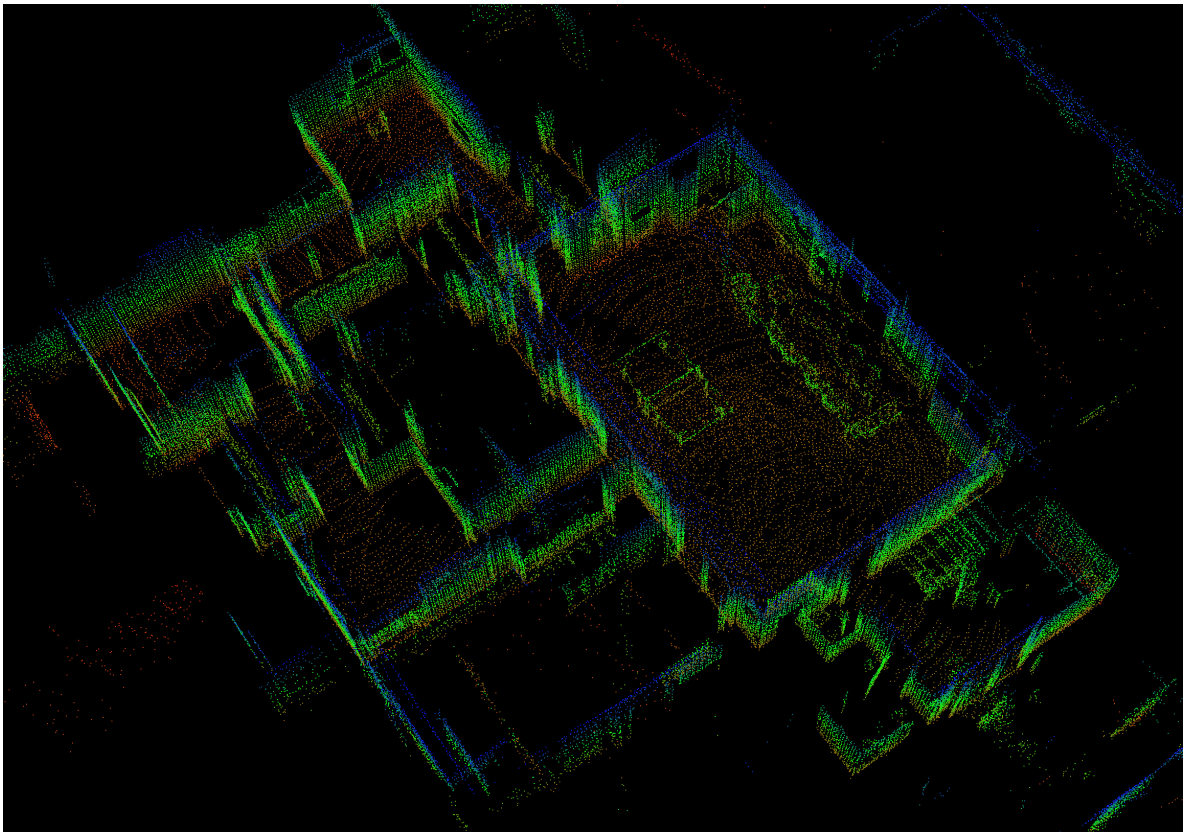


Figure 4.13: Map of the ground floor of a small house computed with the Graph-SLAM algorithm. The walls are consistent with to each other.

Chapter 5

Navigation and path planning

5.1 Context and motivations

Navigation is a very broad topic in the world of robotics and is still widely developed today. Simply put, navigation is the grouping of 3 essential aspects in the movement of robots in their environment: The ability to map its environment, the ability to localize itself in it and finally the ability to program paths between points in this environment. Localization in the mapped environment and path planning are therefore the two cornerstones of robot navigation. These two capabilities can themselves take different forms depending on the application and the environment in which the robot works. However, the goal stays the same: to be able to move safely in an environment and to plan paths connecting several targets.

In this part, we will focus on the planning part of the navigation, i.e. how to reach a point B from a point A.

5.2 Objectives

For the moment, in the simple version of Spot, it can carry out missions only if they have been programmed in advance. Indeed, a technician must first make the desired path with the robot, giving it intermediate points, and then the robot will be able to recreate this path by adapting to minor disturbances, like small obstacles. However, this approach has the disadvantage of needing a technician each time the path needs to be modified, and of not being flexible when larger, unexpected disturbances occur. It would then be interesting to develop new methods to make the Spot navigation system more autonomous and robust. It is in this idea that we have developed on the one hand a SLAM part to

map a new environment and on the other hand a path-planning part. Thus path-planning part will have for objective to allow the creation of a new path in a completely autonomous and decentralized way. In more concrete terms, the goal is to allow a client to give Spot an objective on a map of the building, and that Spot can move from its starting point to this objective without any prior technical manipulation. The time savings would be significant, as well as the possibilities opened by this kind of more autonomous navigation such as constant and precise monitoring of activities in a building, movement of objects without human assistance, characterisation and mapping of construction sites, etc. This method also has the advantage of not requiring a technician every time the robot has to go a different way. The only human intervention would be when we have to explore the environment to create a global map.

Within the framework of this project, the exact goal is to allow Spot to move autonomously in a building in construction and thus in a dynamic environment, by avoiding the obstacles. The objective of a whole building being composed of several sub-objectives, we chose to try to solve only a part of these sub-objectives, the others being left in future development. These objectives are as follows:

- Autonomous travel.
- Moving only on one floor, without stairs.

5.3 Path-planning's inputs

To carry out the navigation, we need several inputs. There are several pieces of information available:

- **Output of the SLAM :** The SLAM developed in the previous chapter allows the reconstruction of a point cloud of the building or environment in which the robot will have to move. Thus, the navigation knows the environment in which it evolves, at least in a static way.
- **Data given by Spot :** Spot has a certain amount of information that can be retrieved for use in the navigation part of the system: the robot's internal IMUs allow the robot's orientation to be known. It is also possible to retrieve information relating to the odometry of the robot as well as the information given by the different cameras of the robot.
- **The localization :** The SLAM also allows us to locate ourselves on the global map. The robot and the path planning know where they are in the environment.
- **Target point :** Finally, the customer or user gives an objective point in the environment.

5.4 Path planning definition

A path planning problem consists in finding in a particular environment a sequence of configurations from a starting state to a final state while avoiding any collision. The path planning inputs can be various, depending on the amount of information available. Typically, path planning works with an online or offline description of the environment such as areas to be avoided, free areas, a starting point and an ending point. The environment can be described in 2D or 3D, with the algorithms having to be adapted in both cases. Although the applications of the path planning and navigation problem are very diverse, let us focus particularly on the case of a mobile robot-like Spot in a 3D environment like a building under construction. Thus, the free zones of the environment will be the zones where Spot will be able to move, while the zones to avoid will be the obstacles. More formally, we describe the search space of a path planning algorithm by 4 subspaces:

- **Configuration Space** C **:** describes the set of possible configurations in the environment. Depending on the problem, these configurations can be described in different ways. For example, for a path planning problem where the robot is considered as a point, the set of configurations can be described by a set of coordinates (x,y,z). If the robot is considered as an object with a certain dimension and certain orientations, then the set of possible configurations will be described by $(x\,, y\,, z\,, \alpha,\, \gamma,\, \phi)$. α,γ and ϕ being the roll, pitch and yaw angles.
- **Free Space** C_{free} **:** describes the set of configurations where the robot does not collide with an obstacle.
- **Target Space** C_{target} **:** describes the configurations that represent the destination of the path planning.
- **Obstacle Space** C_{obst} **:** represents the set of configurations where the robot comes into contact with an obstacle or an area where it cannot access

An important property of path-planning algorithms is the completeness property. In a map, if there is a path between A and B, a complete algorithm will find this path. If no path exists between A and B, a complete algorithm will say that no path exists. On the contrary, a non-complete algorithm will be blocked in the case where no path exists, and will never return anything. It is therefore important in our application that the navigation algorithm chosen by Spot is complete, at the risk of finding ourselves in situations that are solvable but impossible to treat.

5.4.1 Representation of the space

A point cloud is obtained from the output of the SLAM which represents a reconstruction of the environment in which the robot will navigate. However, the point cloud format is not relevant for the path-planning part of the navigation. Indeed, our objective is to be able to define the free space and the obstacle space, and this objective is difficult to achieve with a format including a discrete point cloud. It is then interesting to look at how most path planners represent their environment. This one can be considered as continuous as in the case of the potential field path-planning, or discrete as in the case of Dijkstra or A*. This last format will be more interesting in our practical case because of its simplicity and the completeness property of the algorithms linked to this format. The goal is to transform a 3D point cloud into a 2D matrix where each cell corresponds to a position in the environment, as shown in Fig. 5.1. Each cell can then contain different information about the environment, such as the presence of obstacles, and free space or highlight the objective and the current position of the robot.

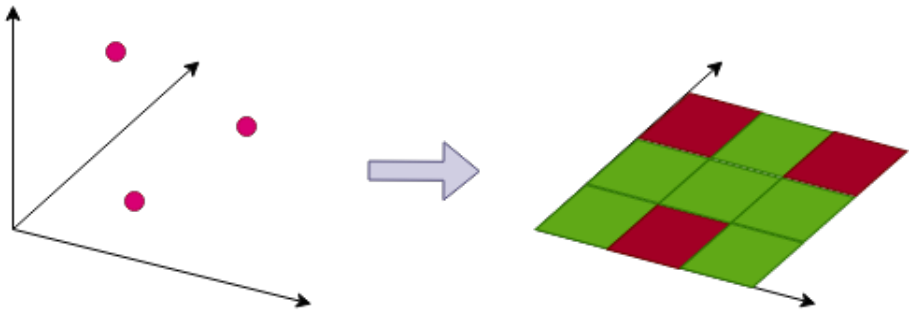


Figure 5.1: Transformation from 3D point cloud to 2D occupancy grid

5.4.1.1 Transformation from point cloud to 2D grid

The data structure used to represent point clouds is the one given by the open-source library PCL (Point Cloud Library) [33]. This structure contains a number n of points composing the point cloud. Each point is described by its 3 coordinates (x, y, z). An example of a point cloud of a corridor, which will be our reference during this section is shown in Fig. 5.2.

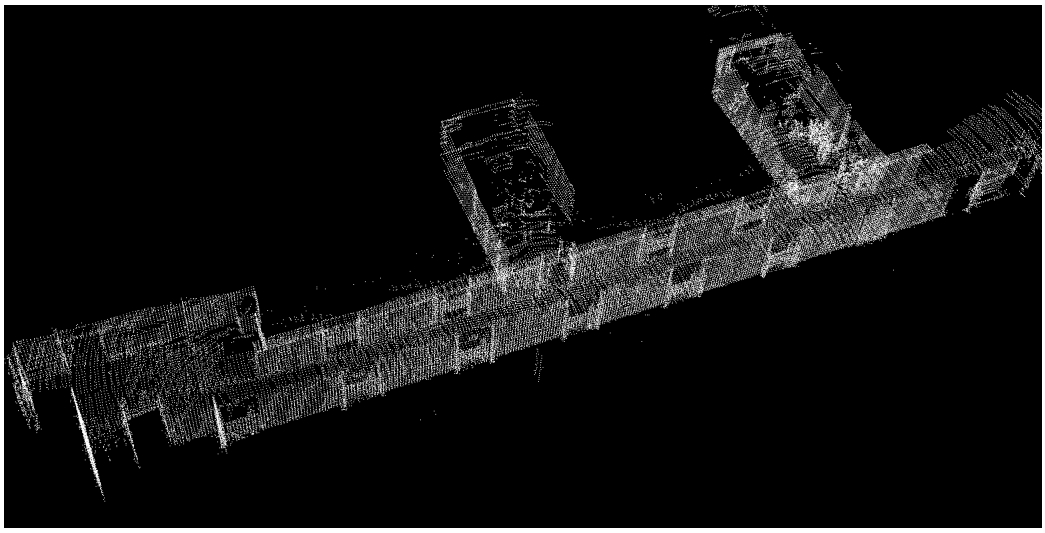


Figure 5.2: 3D point cloud of a corridor constructed thanks to the SLAM algorithm

The first step of our algorithm consists in transforming this 3D point cloud into a 2D point cloud. This step greatly facilitates the conversion from the point cloud to the grid. This is done by taking a slice of the point cloud in the (x, y) plane and projecting each point of this slice into a 2D plane. We then introduce the first two parameters on which we will be able to play: the thickness and the vertical position of the chosen slice. In the implementation we choose a 40cm thick slice, going from 20cm below the LiDAR to 20cm above the LiDAR. To visualize the operation, the choice of the slice is shown in Fig. 5.3. The choice of the slice has the advantage of keeping only the interesting parts for the path-planning part of the navigation, i.e. the walls and the obstacles at the height of the robot. Indeed, the information of the point cloud such as the points composing the floor or the ceiling is not interesting to find a path from a point A to a point B.

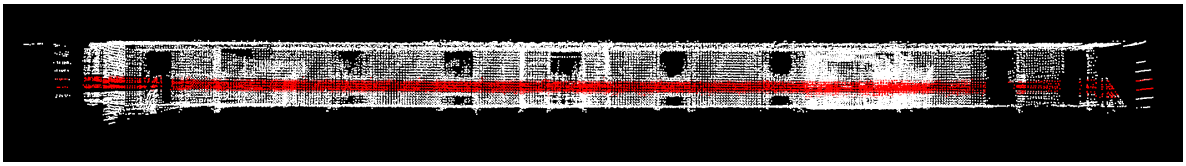
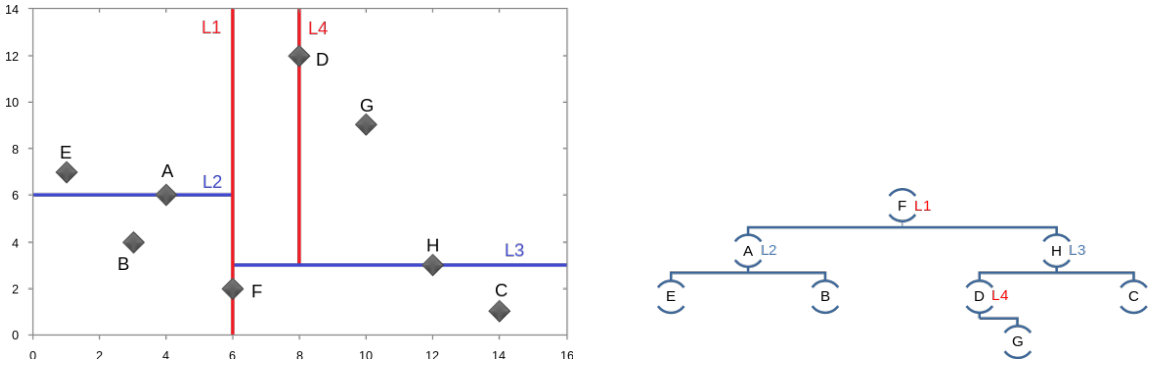


Figure 5.3: In red the chosen points selected by the slice operation on the 3D point cloud of the corridor seen in Fig. 5.2

Now that we have a 2D point cloud, the goal is to transform this point cloud into an occupancy grid. A solution is to create a map with an arbitrary resolution and then check the whole point cloud to see if there is a point contained in the range of a cell. For the sake of efficiency, we will use a KD Tree data structure to perform this operation. Indeed, placing all these points in a KD tree has the

advantage of being efficient in the case where we perform searches in this tree. As a reminder, a KD tree is a data structure that partitions the space and places the data in a tree, which simplifies and accelerates searches. A Fig. of a 2D example KD tree can be found in Fig. 5.4.



(a) 2D KD tree partitioning of points

(b) Tree corresponding to the applied partitioning

Figure 5.4: A KD tree divides the space and places points in a tree to improve searching efficiency [34].

To create our occupancy grid, we will then start by creating an artificial reference point cloud X, each point of which corresponds to the center of each cell of the occupancy grid. We then have 2 point clouds: the one corresponding to the environment Y, and another artificial one X. We place Y in a KD tree and we will then take each point of X and we will look for the point of Y closest to the point of X. If this point is at a distance less than half the resolution, then the corresponding occupancy grid cell will be considered an obstacle. Algorithmically, this gives:

Algorithm 2 *Occupancy_Grid_Filling*()

Input : A filtered 2D point cloud $PCL = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$
Output : Occupancy grid as a matrix $M^{n \times n}$

$Map \leftarrow \emptyset$
for $i = 0\,;\, i < Width\,;\, i++$ **do**
  for $j = 0\,;\, j < Height\,;\, j++$ **do**
    $Artifical\_Point.x \leftarrow Min\_X + i * Resolution$
    $Artifical\_Point.y \leftarrow Min\_Y + j * Resolution$
    $Distance \leftarrow KD\_Tree\_Search\_Closest(Artificial\_Point, PCL)$
    if $Distance < Resolution/2$ **then**
      $Map[i][j] \leftarrow 1$
    else
      $Map[i][j] \leftarrow 0$
    end if
  end for
end for

An example of transformation from the point cloud at Fig. 5.2 to the corresponding occupancy grid is shown in Fig. 5.5.

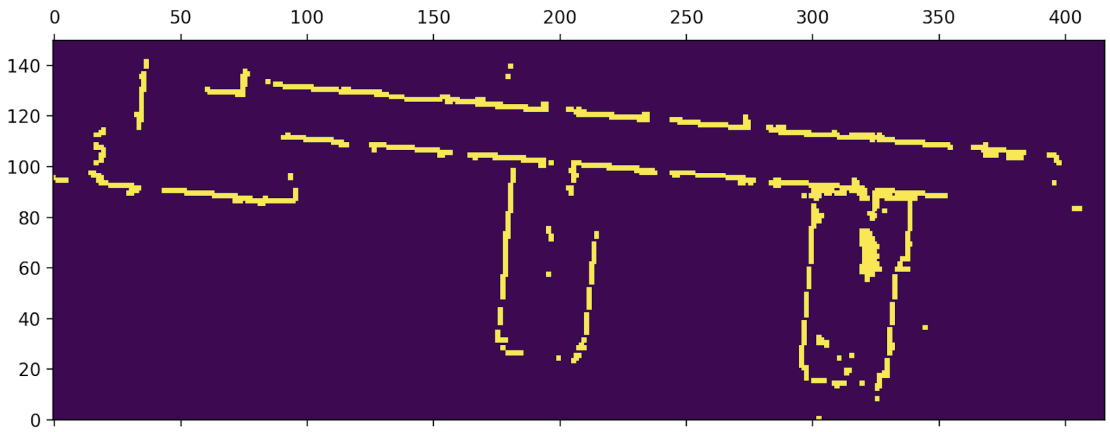


Figure 5.5: Occupancy grid corresponding to the point cloud of the corridor shown in Fig. 5.2

In addition, it is important to consider the treatment of obstacles. Indeed, we must not forget that the robot is a 3-dimensional object and therefore has a certain width and length. In the current situation, if we give the occupancy grid as it is to the path planning, it will consider the robot as a point, and not as an object. The problem is that by not considering the dimensions of the robot, one runs the risk of colliding with obstacles. An easy solution to avoid this situation is to continue to consider the robot as a point (thanks to the ease that this represents in terms of calculation) but to artificially increase the size of the obstacles. Another parameter of the occupancy grid is therefore the augmented size of the obstacles. Fig. 5.6 shows the difference between a map without increased obstacles and a map with increased obstacle size.

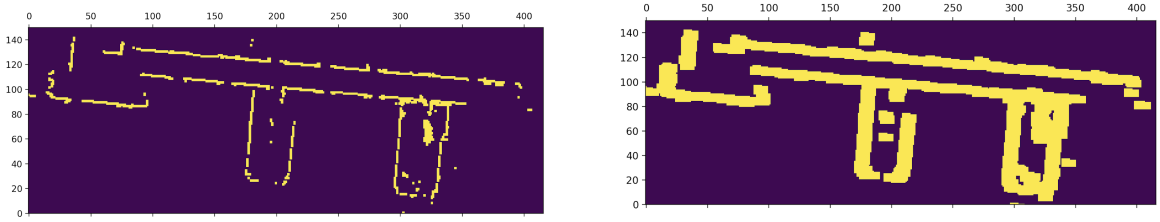


Figure 5.6: Transformation from not increased obstacle map to increased obstacle occupancy grid.

To summarise, the different parameters that can be used to create the occupancy grid are :

- R the resolution of the occupancy grid, i.e. the actual distance that a grid cell represents in reality.
- W the width of the slice to be taken from the 3D point cloud.
- I the increase in size that will be applied to the obstacles.

The parameters W and I will in general be fixed and do not significantly influence the calculation and processing time from the 3D point cloud to the occupancy grid.

However, the resolution parameter R will have a significant impact on the calculation time. It is thus important to make a trade-off between the precision that we want to have and the calculation time taken by the processing of the occupancy grid. The graph in Fig. 5.7 shows the influence of the resolution, and by extension the size of the occupancy grid, on the calculation time. The tests were performed on a Debian virtual machine running on 2 cores of an Apple Macbook Pro 2017 computer, with a 2.3 GHz processor and 8GB of RAM. The processing time must remain acceptable to be able to carry out calculations in real-time. A reasonable value with a trade-off between accuracy and computing speed in the case of this corridor is 15 cm. The resolution to choose depends on the size of the cloud point. If it is larger than the previous example, the resolution will have to be chosen as less precise to keep the trade-off between precision and computation time.

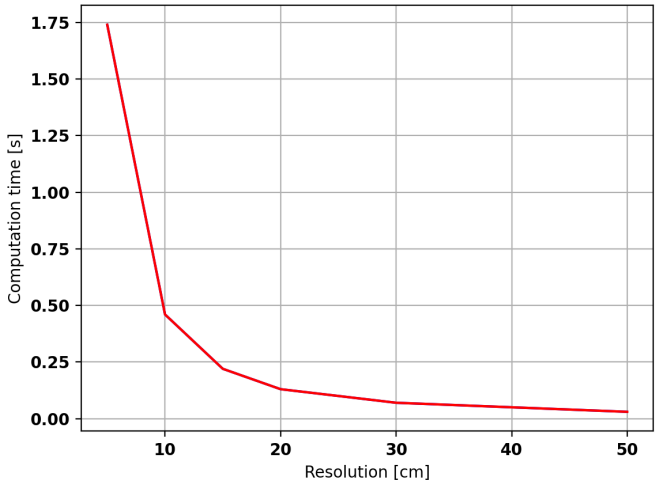


Figure 5.7: Computation speed of the process of the occupancy grid as a function of the resolution for the occupancy grid shown in Fig. 5.5

.

5.4.2 The different types of path planning

Now that we have a usable discrete map, the goal is to develop an efficient path-planning algorithm to connect point A to point B. The efficiency of this algorithm is defined by :

- its speed of execution.
- its completeness property.
- the accuracy and practical efficiency of the calculated path.
- its use of the memory.

There are many different path planning algorithms, each with its advantages and disadvantages. These can be either continuous like the Potential Field Algorithm[35] or discrete. In our application, we have chosen to work in the discrete case. We will start by describing the main algorithms for discrete map search, and then other algorithms with more interesting features will be explained.

Before explaining the basics of path planning, it is important to clarify some information about the occupancy grid. The grid is defined by cells that all have the same dimension in our case. We consider each cell centre as a node, and each node is connected to adjacent nodes by an edge with a cost equal to the distance between them. For example, 2 nodes corresponding to 2 cells sharing a common side will be connected by an edge of cost 1. 2 nodes corresponding to 2 cells sharing a common vertex (i.e. diagonally) will be connected by an edge of cost $\sqrt{(2)}$. This concept is explained visually in Fig. 5.8.

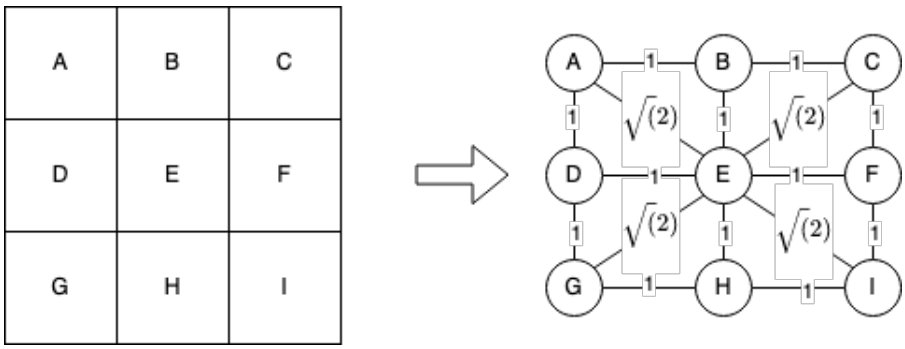


Figure 5.8: Occupancy grid to weighted graph used for path planning

5.4.2.1 Dijkstra

The basic algorithm for pathfinding in a discrete map is the Dijkstra algorithm. This algorithm finds the shortest path between two points. It was invented by Edsger Dijkstra in 1959 [36]. His algorithm

is the following:

Algorithm 3 *Dijkstra*()

Input : Connected graph with n nodes and e edges. $S = (\boldsymbol{x}_s, \boldsymbol{x}_s)$ the starting node. $G = (\boldsymbol{x}_g, \boldsymbol{x}_g)$ the goal.

Output : Path from starting node to goal node as a list $L = \{\boldsymbol{n}_1, \boldsymbol{n}_2, \ldots, \boldsymbol{n}_N\}$

```
1: List ← Graph(n, e)
2: for all Node in List do
3:    Node.distance = ∞
4: end for
5: while List != ∅ do
6:    Node Current ← List.min_Distance()
7:    List.delete(Current)
8:    for all S_1 of Current.successors() do
9:       if S_1.distance > Current.distance + Distance(Current, S_1) then
10:         S_1.distance = Current.distance + Distance(Current, S_1)
11:         S_1.pid = Current.id
12:      end if
13:   end for
14: end while
```

This algorithm starts by placing all the nodes of the graph (or occupancy grid in our case) in a list and initializing all the distances of the nodes from the goal as infinite. Then, as long as this list is not empty, we will choose the node that is currently considered to be closest to the goal (on the first iteration this will be the starting node). Then, each neighbour of the current node will be analyzed. If the previous distance of the neighbour is greater than the sum of the distance of the current node plus the distance between the current and the neighbour, then it means that the path from the current to the neighbour is shorter. We then define the parent of the neighbour as current. We perform this operation for all the neighbours of the current node and repeat the operation until we find the goal. We then have to look at the parent of each node to find the shortest path to the goal.

The complexity of the Dijkstra algorithm is : $O((e+n) \cdot \log n)$, with n the number of nodes and e the number of edges. So although Dijkstra's algorithm gives the shortest path, it is very slow when analyzing large maps. It is therefore important to find a way to speed up the search so that it can be performed in real-time.

5.4.2.2 A*

A* is based on the same concept as Dijkstra's algorithm, except that the cost of each node is no longer simply defined as the so-far cost from the starting point, but an estimate of the distance remaining to reach the goal is added. This concept is called a heuristic. This is possible because we are working in

a discrete world where it is possible to evaluate the distance between 2 coordinates. For example, if we are at the node corresponding to the coordinate (0,5) and the goal is at (10,10), the value of the heuristic will be $(10-0)+(10-5)$. This heuristic is called the Manhattan Distance [37]. There are many different heuristics, but this one has the advantage of not altering the optimality of the result returned by the path-planning algorithm if the graph is cost-uniform [37]. The cost $f(x)$ of a node x with a parent y is, therefore :

$$f(x) = g(y) + distance(x, y) + Manhattan\_distance(y, goal) \tag{5.1}$$

with $g(y)$ the so-far cost of the parent node y from the starting node and $distance(x, y)$ the cost between the node x and y. The implementation of A* is as follows:

Algorithm 4 $A*()$

Input : Connected graph with n nodes and e edges. $S = (\boldsymbol{x}_s, \boldsymbol{x}_s)$ the starting node. $G = (\boldsymbol{x}_g, \boldsymbol{x}_g)$ the goal.
Output : Path from starting node to goal node as a list $L = \{\boldsymbol{n}_1, \boldsymbol{n}_2, \ldots, \boldsymbol{n}_N\}$

```
1: open_list ← ∅
2: closed_list ← ∅
3: open_list.push(start)
4: while open_list is not empty do
5:    current ← open_list.pop()
6:    if current is GOAL then
7:       return closed_list
8:    end if
9:    for all s in current.successors() do
10:      if s not in closed_list then
11:         if s in open_list and s.cost < current.cost_so_far + distance(current, s) + s.heuristic()
            then
12:            continue
13:         else
14:            s.cost = current.cost_so_far + distance(current, s) + s.heuristic()
15:            open_list.push(s)
16:         end if
17:      end if
18:   end for
19:   closed_list.push(current)
20: end while
21: return closed_list
```

Chapter 6

Jump Point Search

In the context of an autonomous mobile robot like Spot, we need the most efficient, fast and lightweight path planning algorithm possible so that the possible applications can be realized in real-time, which means in our application, less than 0.2 seconds. As the acquisition time of a point cloud varies, 0.2 seconds is a robust margin needed to satisfy the real-time constrains. From this perspective, it is interesting to analyze the weaknesses of the A* algorithm. To understand its main weakness, it is necessary to introduce the concept of symmetrical paths. Two paths are symmetrical if they have the same starting point and the same ending point and if the distance made by the two paths between these two points is the same. In other words, it is possible to obtain path 1 by exchanging the vectors constituting path 2. An example of several symmetrical paths is given in Fig. 6.1. When A* expands its nodes, it explores all symmetrical paths, which is a waste of time and memory, resulting in a decrease in performance. Many algorithms are trying to solve this weakness by exploiting the concept of pruning to increase performance such as Rectangular Symmetry Reduction[38]. The concept of pruning is the fact of ignoring certain parts of the search space because these parts will mathematically not give the right solution. In this work, we will focus on a recent and widely used algorithm: Jump Point Search (JPS). This algorithm has been developed by Daniel Harabor and Alban Grastien in 2011[39].

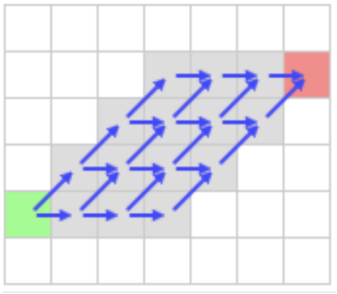


Figure 6.1: Example of symmetrical paths [40].

JPS uses the notion of pruning, i.e. the disregard of some nodes to reduce the computation time and the memory used. The algorithm follows certain rules to achieve this selection of nodes. JPS is a derivative of A*, making it faster, using less memory, but above all keeping an interesting property of A*, namely its optimality[39]. The optimality of JPS means that the path given by the algorithm is the shortest and that the algorithm is complete.

It is also interesting to note that JPS is particularly useful in the context of an indoor mobile robot. Indeed, the grid derived from the SLAM result has the characteristic that a lot of open and free spaces are present, for example, a long empty corridor. This situation gives rise to a large number of symmetrical paths which are therefore detrimental to A* but which JPS can avoid.

6.1 Implementation

Let's now look at how JPS is implemented in practice, i.e. what are the constitutive rules of the algorithm. We, therefore, start again from the occupancy grid that we calculated in the previous chapter. We assume that each node has 8 neighbours, including diagonal movements. The specificity of JPS, as explained above, is to ignore some nodes that we are sure are not interesting for the path calculation. There are therefore pruning rules, i.e. under which conditions we can ignore certain nodes and jumping rules. The latter comes from the fact that, unlike A* which explores the nodes adjacent to the parent node, JPS will continue its expansion and will only place a node in the open list when certain conditions are met.

Fig. 6.2 represents this concept of jumping points. Instead of exploring each adjacent node, we will continue in the previous direction until we find or do not a jumping point. A cost from the node x to the jumping node y is given by

$$g(y) = g(x) + dist(x, y)$$

with g() the cost so-far from the starting point. The exploration thus continues from this jump point.

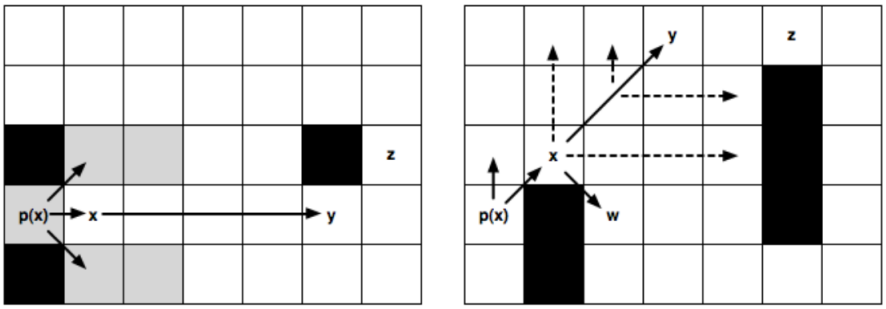


Figure 6.2: Visual example of the straight and diagonal jump points concept [39]. The black cells represents obstacles, the white cells the free spaces and the grey ones the cells that are pruned

The algorithm is therefore based on neighbour pruning rules which define which nodes can be ignored in the path calculation. The goal here is to start from a parent node p(x) and arrive at an adjacent node x along a certain direction and look at all its neighbours. Among these neighbours, we will estimate which ones are not worth visiting to reach optimality. The rules concerning this selection depending on whether the direction is straight or diagonal.

For a straight motion, we will consider each neighbor n of node x reached by its parent $p(x)$ and prune each neighbor satisfying the following equation [39]:

$$len(< p(x), \ldots, n > \setminus x) \leq len(< p(x), x, n >) \tag{6.1}$$

That is, we ignore all neighbouring nodes whose path from the parent to these nodes without passing through x is shorter than or equal to the path between these same nodes but passing through x. In the example shown in Fig. 6.3, we can see that all the shaded nodes respect the equation and are therefore eliminated from the path calculation. The remaining nodes, considering an environment without obstacles, are called the natural neighbours of x.

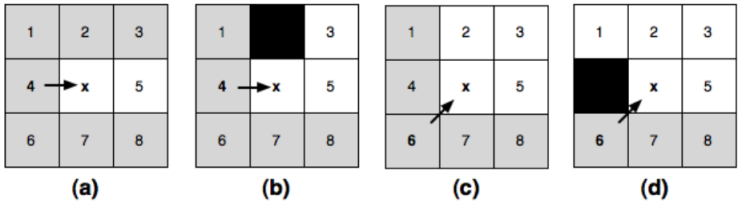


Figure 6.3: Pruning example in the case of different direction moves [39]. The black cells represents obstacles, the white cells the free spaces and the grey ones the cells that are pruned.

The diagonal movements have to answer another equation, almost similar, with the difference that the paths passing and not passing through x cannot be equal.

$$len(< p(x), ..., n > \setminus x) < len(< p(x), x, n >) \tag{6.2}$$

In practice, we will see that it is possible to determine the neighbours to be pruned in diagonal movements by a combination of vertical and horizontal movement analysis. An example of diagonal motion pruning is given in Fig. 6.3.

We have previously introduced the concept of natural neighbours. As a reminder, natural neighbours are those nodes that are not eliminated by the pruning process, in an unobstructed environment. It is now important to introduce the concept of forced neighbours. When passing through a non-ideal environment with an obstacle, it may happen that some nodes that would have been pruned in an unobstructed environment are not removable. They are not ignorable because they do not meet either the 6.1 or the 6.2 equations. These nodes are then called forced nodes. An example demonstrating this concept is shown in Fig. 6.3. the formal definition of jump points introduces the concept of forced neighbours and is defined by [39]:

Definition 1. *A neighbor n belonging to the direct neighbors of x is said to be* ***forced*** *if :*

1. *n is not a natural neighbour of x*
2. $len(< p(x), x, n >) < len(< p(x), ..., n > \setminus x)$

Finally, it is necessary to define the decision criteria for jump points. During the search, we will start from a parent node p(x) and go in one direction. If the current node is not a jump point, we continue in the same direction until we find a jump point to place in the *"to be explored"* list. If there is no jump point in that direction, we place nothing more in the *"to be explored"* list and the algorithm continues. The different rules to define if a node x is a jump point are the following:

Definition 2. *The node y is the* ***jump point of x****, if considering the direction d such that y = x + k*d, and one of the following conditions is met [39]:*

1. *Node y is the goal node.*
2. *The node y has at least one of its neighbors that is forced.*
3. *If d is a diagonal direction, then there exists a node z such that* $z = y + k_i * d_i$ *with* $d_i = d_1, d_2$ *the 2 horizontal and vertical vectors constituting the direction d and z is a node satisfying at least condition 1 or 2.*

Fig. 6.2 illustrates these jump point rules in a straight and diagonal manner.

To clarify the functioning of this path-planning, let's look more precisely at the practical implementation of the algorithm.

Algorithm 5 $JPS()$

Input : Connected graph with n nodes and e edges
Output : Path from starting node to goal node as a list $L = \{\boldsymbol{n}_1, \boldsymbol{n}_2, \ldots, \boldsymbol{n}_N\}$

```
1:  open_list ← Ø
2:  closed_list ← Ø
3:  open_list.push(start)
4:  while open_list is not empty do
5:    current ← open_list.pop()
6:    if current is GOAL then
7:      return closed_list
8:    end if
9:    moves ← actions(current)
10:   for i in moves do
11:     if i not in the map or obstacle then
12:       continue
13:     end if
14:     jump_point ← jump(current, direction(current, i))
15:     if jump_point is NULL then
16:       continue
17:     end if
18:     if jump_point is GOAL then
19:       open_list.push(jump_point)
20:       break
21:     end if
22:     if jump_point not in closed_list then
23:       open_list.push(jump_point)
24:     end if
25:   end for
26:   closed_list.push(current)
27: end while
28: return closed_list
```

The main loop of the algorithm starts by initializing the open list and the closed list. The open

list corresponds to the nodes to be explored and is implemented as a priority queue with as sorting criterion

$$f(x) = g(x) + h(x)$$

with $f(x)$ the so far distance and $h(x)$ the chosen heuristic. Here, the euclidean distance has been chosen to be the heuristic. This heuristic guarantees the optimality of the result since it is admissible and consistent[37]. The closed list represents the list of nodes already visited.

As long as the open list still contains nodes to visit, it means that there are still possibilities to explore, otherwise, the algorithm stops and notifies the user that there is no possible path.

In the main loop starting at line 3 of algorithm 5, we start by taking the first node of the open list. Then we check if this node is the goal. In this case, we return the result of the traversed path, i.e. the closed list. In practice, we do not return the closed list. We have to go back up the path travelled using the id of the parents of each node until the id of the first node.

If the current node is not the goal, we take all the possible actions of this node, i.e. 8 possible moves. We then go through each move and check that it does not lead off the map or into an obstacle. If not, we run algorithm 6, which returns the potential jump point related to the movement between the current node and the studied move. If this jump point exists, we place it in the open list to be explored later.

After having studied each possible action linked to the current node, we place this node in the closed list.

Algorithm 6 $jump(current, direction)$

Input : Connected graph with n nodes and e edges. *current* the current observed node. *direction* the direction applied during the jump

Output : First jump point found or $NULL$ if no jump point

```
1: new_node = step(current, direction)
2: if new_node not in map then
3:    return NULL
4: end if
5: if new_node is obstacle then
6:    return NULL
7: end if
8: if new_node is GOAL then
9:    return new_node
10: end if
11: if new_node has forced neighbours then
12:    return new_node
13: end if
14: if direction is diagonal then
15:    if jump(new_node, d1) is not NULL then
16:       return new_node
17:    end if
18:    if jump(new_node, d2) is not NULL then
19:       return new_node
20:    end if
21: end if
22: return jump(new_node, direction)
```

Algorithm 6 is a recursive function. We start by exploring the next node in the studied direction. After checking the natural conditions, such as the presence of an obstacle or if the node is outside the map, the studied node is checked and returned if it has forced neighbours as defined in 1.

In the particular case of exploration in a diagonal direction, we recall the same function but extend the search to the two horizontal and vertical directions corresponding to the constituent vectors of this diagonal direction :

$$\boldsymbol{d} = \{d_1, d_2\}$$

In the case where no jump point is found, we continue the exploration by recursively calling the function but extending the search to the next node.

An example of a JPS result is shown in Fig. 6.4 with a comparison with the A* algorithm result. The next section is dedicated to a more detailed analysis.

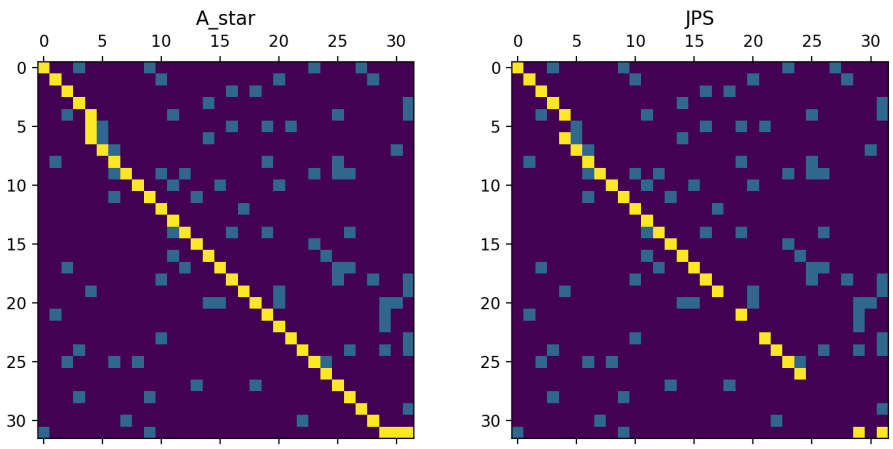


Figure 6.4: Visual representation of JPS result compared to A* result. The ability of JPS to make jumps is then highlighted.

6.2 Improvements

Our JPS implementation works well and is faster than A*, especially when the map gets large, i.e. more than a 600x600 grid. Nevertheless, some optimizations can be made. In the next sections, we will describe the improvements we have made to JPS, which are on the one hand inspired by the JPS+ paper written by Daniel Harabor [41], and on the other hand completely original.

6.2.1 JPS+

At the moment, each cell on the card is implemented as an int (or a boolean), which is not memory efficient at all. An idea inspired by JPS+ is to implement the map data as 32-bit words, i.e. int, but where each bit would correspond to a value of 1 if there is an obstacle and 0 if the space is free [41]. It is then necessary, for simplicity, that the dimensions of the map be a multiple of 32, to facilitate the bitwise operations that we are going to perform. We, therefore, divide the memory used by 32 (or 8 in the case when booleans are used). For a map which at the beginning is of dimension NxN, we obtain a map of N/32xN, since we put the values on words of 32 bits only on the lines, and not on the columns. We can then perform bitwise operations during the analysis of the map which also improves the speed of the algorithm.

In the paper on JPS+, the authors use bitwise operations to check the presence of obstacles or forced neighbours. However, in the current implementation, we do not use this technique which will

be left for future improvement.

6.2.2 Filtering

In Fig. 6.4, we can observe that despite some jumps for JPS performed between the points, it often happens that several points could still be eliminated. In the final result, a large number of jump points could be reduced. When looking at Fig. 6.4, we can easily notice that all the jump points from the cell with coordinates (7,5) to the cell (31,29) could be summarized in only 2 jump points. Unfortunately, due to the intrinsic implementation of the JPS algorithm, it is not possible to solve this problem in real-time. However, it is possible to apply a filter after the result.

This filter concept can be more easily understood from the point of view of a mobile robot like Spot. Indeed, the purpose of navigation is to give the robot orders (direction, distance) that it will have to follow. It is then greatly appreciated if the number of turning points, i.e. the change of direction, is as low as possible to facilitate the movements. The comparison between JPS without a filter and JPS with a filter applied is shown in Fig. 6.5, which shows the practical usefulness of this filter.

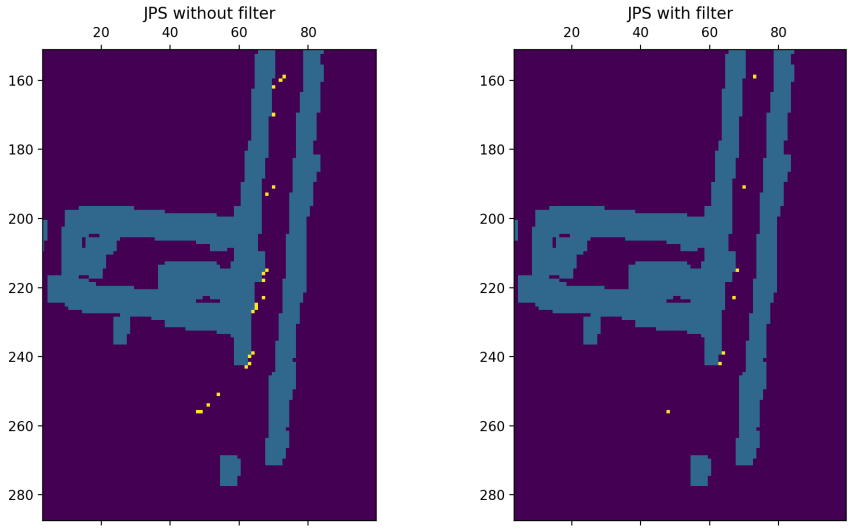


Figure 6.5: Effectiveness of the filter on JPS

A more quantitative analysis of the number of turning points shown in Table 6.1 reveals a decrease of on average 360 % thanks to the filter. This analysis is performed on randomly generated maps with a density of 30% of obstacles.

Size of the map	JPS	JPS filtred
100x100	102 ± 1.9 turning points	31 ± 3.6 turning points
200x200	237 ± 5.2 turning points	70 ± 4.9 turning points
500x500	512 ± 5.2 turning points	145 ± 8.2 turning points
1000x1000	1055 ± 7.7 turning points	294 ± 11.8 turning points

Table 6.1: Analysis of the number of turning points as a function of map size with and without the filter applied. Each test was performed 10 times on each card.

The algorithm used to perform this filtering is the Bresenham line algorithm [42]. This one was invented in 1962 by Jack E. Bresenham and its concept is to draw in a discrete plane, i.e. a grid, an estimation of a line connecting 2 points. An example of the result that we can hope to obtain is shown in Fig. 6.6.

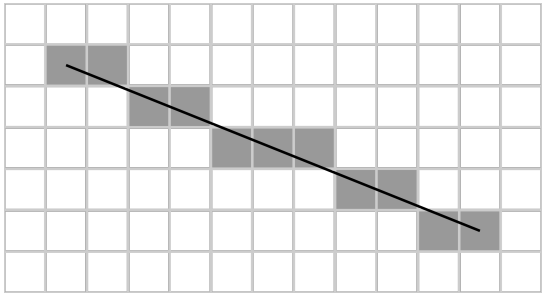


Figure 6.6: Bresenham line algorithm result [43].

In the case of the filtering that we carry out on the results given by JPS, its use is different, but we will develop this later. Let's start by explaining briefly how the Bresenham algorithm works.

Let's consider a line equation with a slope between 0 and 1, namely :

$$y = m * x \quad with\ 0 \leq m \leq 1$$

The objective that we will have is to determine in the grid if we choose to take (x+1,y) or (x+1,y+1). This choice will be determined by a simple minimization of the error committed, as can be seen in Fig. 6.7.

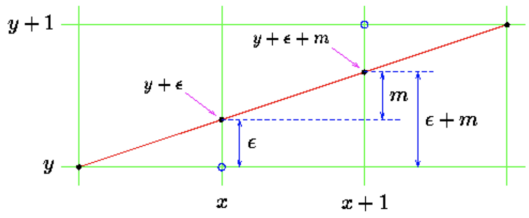


Figure 6.7: Bresenham line algorithm notations [44].

In x we define an error between the continuous value and the discrete value of ϵ. In x+1, it is obvious that this error will have increased by the value of the slope m. The decision will then be to choose (x+1,y) if this error is smaller than 0.5. Mathematically, we will write :

$$y + m + \epsilon \leq y + 0.5 \tag{6.3}$$

Then, we repeat the process by taking an initial error ϵ' equal to the total error at the previous step:

if (x+1,y) has been chosen :

$$\epsilon' = m + \epsilon$$

if (x+1,y+1) has been chosen :

$$\epsilon' = m + \epsilon - 1$$

The problem so far is that the error calculation employs floating point values. The goal is, for efficiency and speed, to use only integers. A technique that can be used is to start from the equation 6.3, and multiply each side of the equation by $2\Delta x$. We then obtain :

$$2\epsilon\Delta x + 2\Delta y \leq \Delta x \tag{6.4}$$

Here, all terms are integers. ϵ is demonstrated as an integer by the following proof where the case (x+1,y) is chosen. The process is the same if we would have picked (x+1,y+1), only the equations are slightly different :

$$\epsilon \rightarrow \epsilon + m$$

By multiplying by Δx :

$$\Delta x\epsilon \rightarrow \epsilon\Delta x + m\Delta x$$

We can notice that the update will always be an integer and we can then use this result in equation 6.4. It is obvious that this approach only works for slopes between 0 and 1. To extend the algorithm to other possible slopes, it is necessary to slightly modify the equations, but the principle remains the same. The final algorithm is as follows:

Algorithm 7 *Call_to_Bresenham*(x_1, x_2, y_1, y_2)

Input : Starting point coordinates $\boldsymbol{S} = (\boldsymbol{x}_1, \boldsymbol{y}_1)$. Final point coordinates $\boldsymbol{F} = (\boldsymbol{x}_2, \boldsymbol{y}_2)$
Output : Discrete line joining starting and final points

$d_x = x_2 - x_1$
$d_y = y_2 - y_1$
if $d_x > d_y$ **then**
  $Bresenham(x_1, y_1, x_2, y_2, d_x, d_y, 0)$
else
  $Bresenham(y_1, x_1, y_2, x_2, d_y, d_x, 1)$
end if

Algorithm 8 *Bresenham*$(x_1, y_1, x_2, y_2, d_x, d_y, decide)$

Input : Starting point coordinates $\boldsymbol{S} = (\boldsymbol{x}_1, \boldsymbol{y}_1)$. X resolution $\boldsymbol{d}_x$. Y resolution $\boldsymbol{d}_y$. Boolean variable *decide* for the sign of the slope Final point coordinates $\boldsymbol{F} = (\boldsymbol{x}_2, \boldsymbol{y}_2)$
Output : Discrete line joining starting and final points

$p_k = 2 * d_y + d_x$
$i = 0$
for $i < d_x$ **do**
  if $x_1 < x_2$ **then**
    $x_1 + +$
  else
    $x_1 - -$
  end if
  if $p_k < 0$ **then**
    if $decide == 0$ **then**
      $plot(x_1, y_1)$
    else
      $plot(y_1, x_1)$
    end if
    $p_k = p_k + 2 * d_y$
  else
    if $y_1 < y_2$ **then**
      $y_1 + +$
    else
      $y_1 - -$
    end if
    if $decide == 0$ **then**
      $plot(x_1, y_1)$
    else
      $plot(y_1, x_1)$
    end if
    $p_k = p_k + 2 * d_y - 2 * d_x$
  end if
end for

The differences between the presented algorithm and the one described theoretically come simply from the fact that the latter must be adapted to be applicable for all possible slope values, as explained above.

It is now interesting to look at how this algorithm is used to filter the results given by the JPS algorithm. The goal is to reduce the turning points as much as possible. Once the result of JPS is obtained, we will look at the first point and draw a fictitious line between this point and the next one thanks to the Bresenham algorithm. If no cell in this dummy line is an obstacle, then we again draw a line between the first point and the third intermediate point. If again, no obstacle is detected between these two points, then we simply eliminate the second intermediate point, as it has become unnecessary in the final path we are trying to obtain. If there are no obstacles between the first and the second or between the first and the third, then the second is useless in the final path. We repeat this procedure until we find an obstacle between point 1 and point n. In this case, we keep only point 1 and point n-1 and start the process again at point n-1. A visual description is provided in Fig. 6.8.

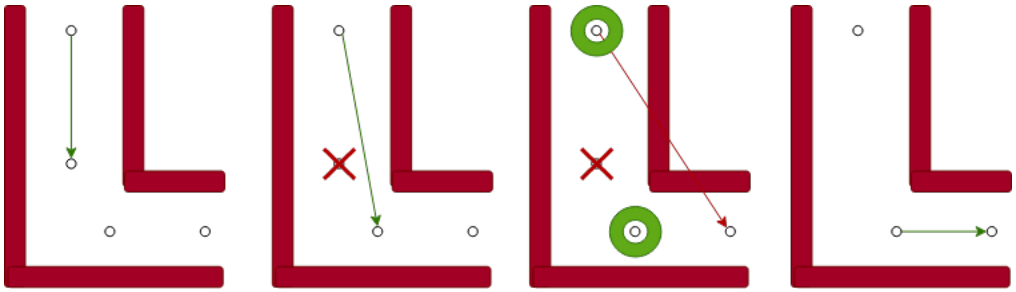


Figure 6.8: Filtering of jump points using the Bresenham algorithm. The white circles represent jump points. The green line represents a successful check between two points. A red line is displayed when 2 points cannot be connected by a straight line.

The results obtained in practice are variable, depending on the architecture of the building analyzed by Spot, but we can expect an average division by 3 of the number of turning points in the final result, for a very negligible computational cost, as we will analyze in the following sections.

6.3 Results

Let's now focus on the performance and the analysis of the influence of parameters such as the resolution, the size of the map to be analyzed, and the density of obstacles or their layout. The various tests were performed on a Debian virtual machine running on 2 cores of an Apple Macbook Pro 2017 computer, with a 2.3 GHz processor and 8GB of RAM and all the algorithms are coded in C++. We will compare the performances of A*, which is the reference path-planning algorithm [45], with the

performances of JPS coupled with the different improvements we have made to it.

We will start by generating a random map of size 600x600 corresponding to 360 000 cells, and let's randomly generate obstacles in this map. The density of the obstacles will then vary from 10 % to 70 %. A* and JPS will be compared in terms of computing time.

The results of the analysis are shown in Fig. 6.9. We notice that for randomly generated obstacles, the performances of A* and JPS are sensibly identical at low intensity but differ greatly when this density increases. However, this analysis alone does not reveal the advantages of JPS, since it eliminates symmetrical paths, which does not necessarily happen in the case of randomly generated obstacles on a map. The conclusion that can be drawn from this first analysis is that, even in a situation where JPS is not supposed to obtain excellent performances, it is still more efficient than A*.

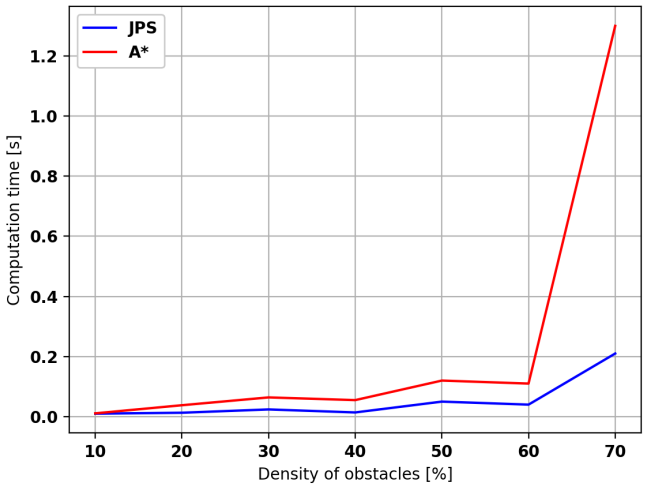


Figure 6.9: Influence of the density of obstacle on a 600x600 random map

For the second analysis, let's take a randomly generated map but the variable is now the size of the map from 32x32 to a relatively large map of 1600x1600 with a fixed obstacle density of 30% randomly generated. Let's compare the performance of A* and JPS in this type of environment. The results of this analysis are shown in Fig. 6.10.

When the size of the random map is relatively small (up to about 500x500), the performances of A* and JPS are more or less equal. But we notice that when the size of the map increases, the computation time of A* increases drastically (300% increase between 200x200 and 1600x1600), while JPS remains relatively stable (40% increase between 200x200 and 1600x1600). This shows the ability of JPS to remain real-time efficient, even in the case of a large map and in an environment that is not

optimal for JPS.

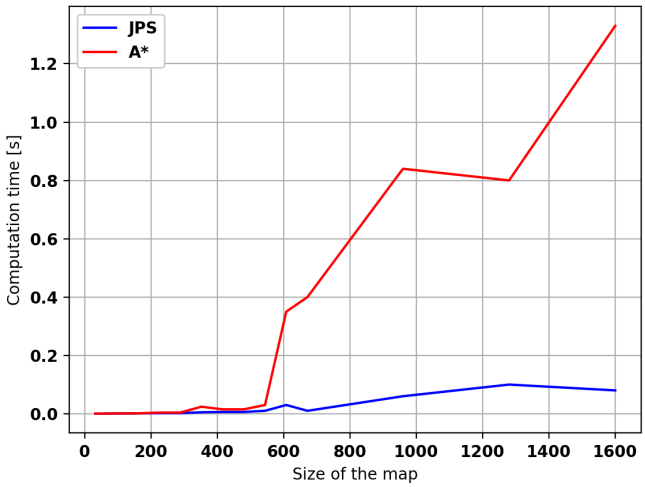


Figure 6.10: Influence of the size of the random map with fixed 30% obstacle density

For the third analysis, we will take an environment that corresponds to what could be found on a construction site: we take a cloud point reconstructed by SLAM, and we transform it into an occupancy grid. The goal is to go from one end of a corridor to the other. The parameter we will play with is the resolution chosen to fill the occupancy grid. This resolution will go from 5 cm to 50 cm, a smaller resolution means a higher accuracy but also a larger map. It is also important to note that in this type of environment there is a lot of free space, which means that JPS is in a more optimal environment than a randomly generated map. Moreover, during the simulations, we notice that A* is not optimal and is not adapted to a real-time application. Indeed, even with a resolution of 50 cm, the computation time is several tens of seconds, depending on the distance between the starting point and the arrival point. A* will therefore not be displayed in the graph illustrating the analysis and which can be found in Fig. 6.11.

We notice that if the resolution is too precise, JPS starts to show signs of weakness. Fortunately, this weakness quickly diminishes and the algorithm becomes usable from a resolution of 10 cm. If we keep the same resolution and increase the size of the converted cloud point, we increase the size of the occupancy grid and consequently the computation time. For a corridor of about 30m and a resolution of 10cm, the size of the map is about 400x400 and JPS performs the path-planning in 0.15 seconds, which makes it usable in a real-time application.

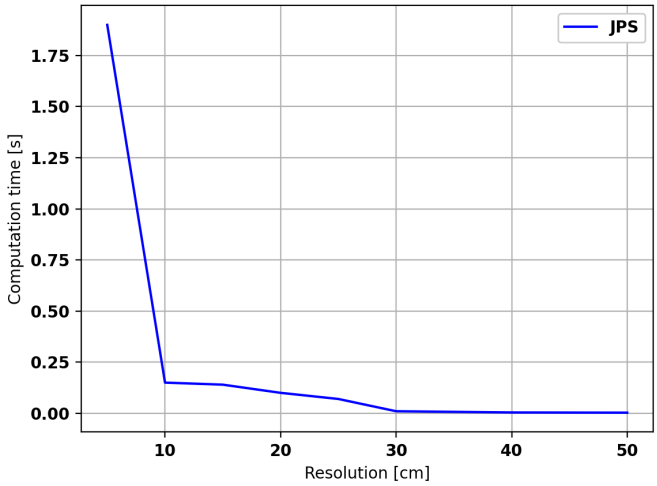


Figure 6.11: Influence of the resolution on the reconstructed corridor map

Chapter 7

Results

7.1 SLAM results

To evaluate our SLAM approach, we have to evaluate our LiDAR odometry and the overall SLAM system. This section will evaluate first our LiDAR odometry in multiple indoor environments. All the tests were realized with a sequence of point clouds and the robot's odometry was taken at a frequency of 5 Hz during the robot's path. This frequency was chosen because a point cloud can be reliably recovered from Spot within 0.2s. The mean speed of the robot along each path was set to 0.6 m/s. All tests done below were executed on a Ubuntu 20.04 virtual machine with 2 cores of an Intel i7-4870HQ @ 2.50GHz CPU and 8GB of RAM. For reasons of computation speed, the SLAM algorithm is implemented in C++.

7.1.1 LiDAR odometry evaluation

The LiDAR odometry can be evaluated by computing the drift of the poses estimation along the path of the robot. In an indoor environment, it can be computed by making the robot do a loop inside the environment. The drift in the odometry can then be computed as the difference between the starting pose and the ending pose. But it is hard to make the robot start and end in the same place for every test. Instead, AprilTags can be used to compute the difference in position. The robot can start its path near an AprilTag and detect its relative position to it. When the robot ends its loop, the AprilTag can also be detected and the relative position of the robot with respect to it can then be used to estimate the drift of the overall path. The overall drift can then be divided by the length of the path to having to estimate the amount of drift in the odometry per meter.

As a first test, the LiDAR odometry is tested in two different environments against the robot's odometry and the robot's visual odometry. The first environment is shown in Fig. 7.1 and is a small empty house. One particularity of this environment is that there are a lot of windows that under a certain angle reflect the lasers of the LiDAR. It can be seen on the map that there are a lot of ghost walls due to this problem. The second environment is a hall used as a test environment at the BBRI (Belgian Building Research Institute). As it can be seen in Fig. 7.2, it is far less empty and there are also far fewer surfaces that could cause reflection of the LiDAR's lasers. The path taken by the robot in both environments follows the big squares in the Fig. 7.2 and 7.1. For each environment, 3 tests were realised. Table 7.1 lists the drift estimations of this set of tests.

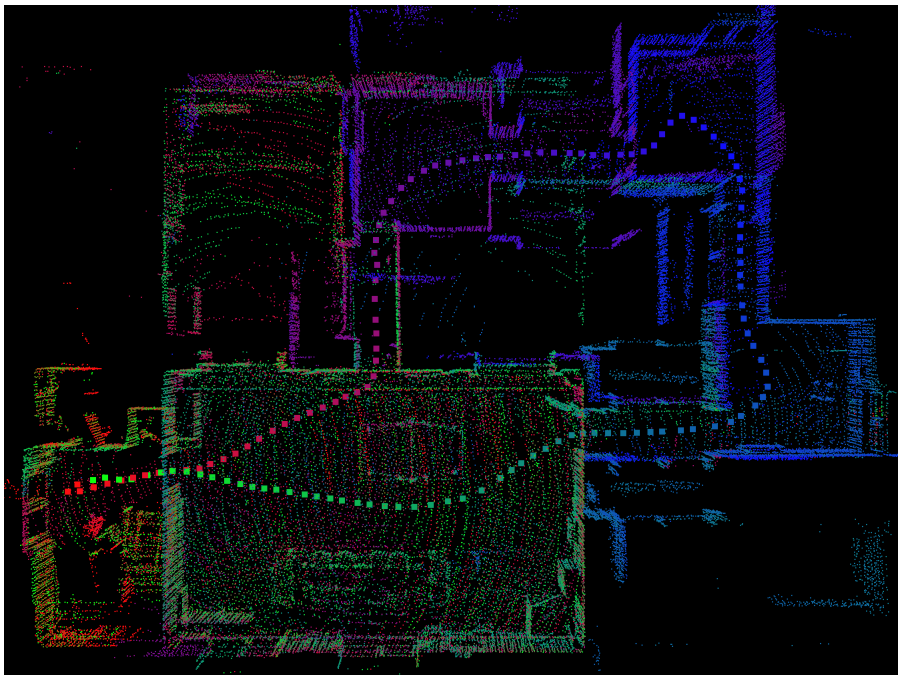


Figure 7.1: Top down view of the map of the house computed with the Graph-SLAM algorithm. The path of the robot follows the larger square. The robot's path starts at the red and ends at the green.

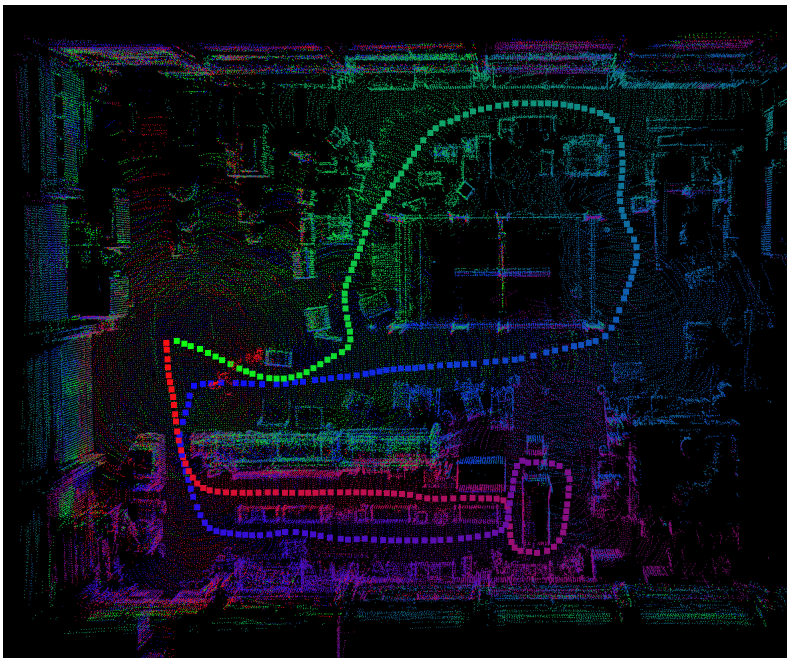


Figure 7.2: Top down view of the map of the hall computed with the Graph-SLAM algorithm. The path of the robot follows the larger squares. The robot's path starts at the red and ends at the green.

	Test	Path length	Odometry relative drift	Visual odometry relative drift	LiDAR odometry relative drift
Hall	1	∼110.6 m	1.910 %	0.354 %	0.083 %
	2	∼113.4 m	1.875 %	0.183 %	0.238 %
	3	∼112.6 m	1.889 %	0.19 %	0.048 %
House	1	∼49.9 m	1.996 %	0.856 %	0.215 %
	2	∼48.6 m	1.867 %	1.005 %	0.398 %
	3	∼48.1 m	1.969 %	0.237 %	0.472 %

Table 7.1: Results of the tests on the relative drift error of the LiDAR odometry in the house and hall environments.

The results of Table 7.1 show that our LiDAR odometry performs quite well in both environments. But as we rely on the measurements of the AprilTag, which are not accurate, we must look at the scale of the relative drifts and not the exact values.

The results clearly show that our method outperforms the basic odometry of the robot in both environments. The error of the classic odometry is more or less the same throughout the tests. This can be expected since no external measurements are taken into account. So the relative error stays the same whatever the path the robot takes. One major issue our technique solves is the major drift in the z direction of the classic odometries as seen in Fig. 7.3.

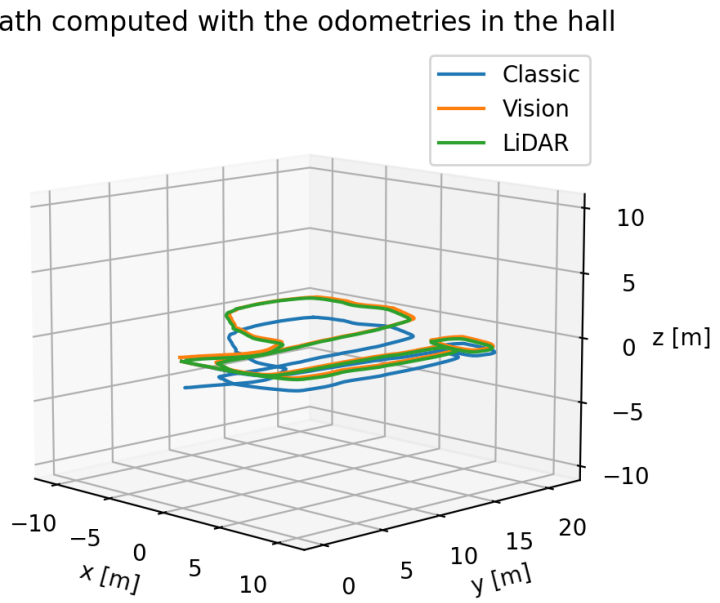


Figure 7.3: Paths of the first test inside the hall reconstructed with the different odometries.

Overall on these tests, our method performed better than the vision odometry implemented on Spot. Especially in the house environment even if the LiDAR is negatively impacted by the reflections on the windows.

While our navigation solution doesn't allow the robot to cross stairs, the LiDAR odometry and the Graph-SLAM can be used when the robot goes through multiple floors. To test this ability, the same evaluation method was used on another type of environment. This test was performed on two floors, both contain long corridors with few features. The environment is represented in Fig. 7.4. The results of this experiment can be seen in Table 7.2.

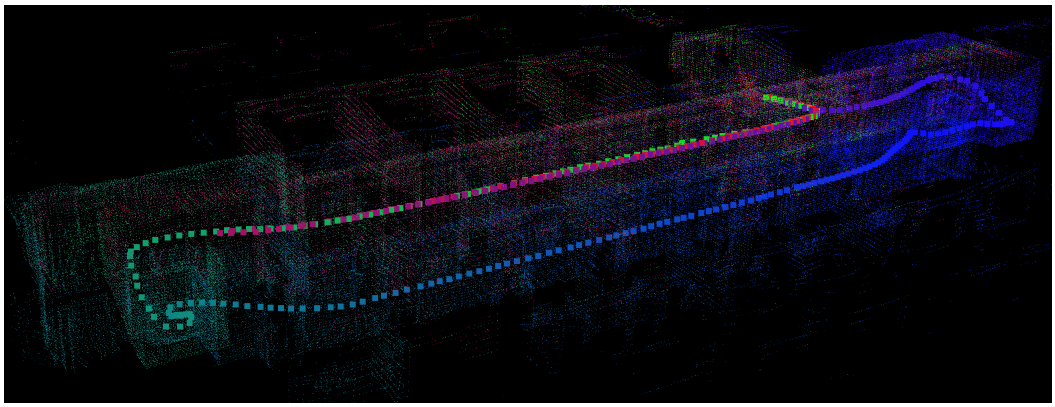


Figure 7.4: Side view of the map of the corridors computed with the Graph-SLAM algorithm. The path of the robot follows the larger squares. The robot's path starts at the red and ends at the green.

	Test	Path length	Odometry relative drift	Vision odometry relative drift	LiDAR odometry relative drift
Corridors	1	∼148.9 m	1.695 %	0.522 %	0.629 %
	2	∼149.2 m	1.654 %	0.157 %	0.440 %

Table 7.2: Results of the tests on the relative drift error of the LiDAR odometry in the environment with two floors.

The result of Table 7.2 shows that our LiDAR odometry performs less well when the robot explores multiple floors. This can be explained by the fact that the minimum speed the robot can go up and down the stairs is higher than its minimal speed on flat ground. This induces errors in the correction of the distortion in the point clouds done by the robot when it is located on stairs. If the distortion of the point cloud is not corrected enough, it induces an error in the LiDAR odometry as it will be based on the wrong measurements.

7.1.2 Graph-SLAM evaluation

The first test that can be made for our Graph-SLAM algorithm is to evaluate its ability to close loops in the robot path. As all the tests done previously are simple loops in the environment, we can test the ability to close loops by verifying the position with the AprilTags. This verification can be done in the same way but here the drifts will not be divided by the length of the paths. Table 7.3 reports these results on all the previously tested environments.

	Test	LiDAR odometry position error	Graph-SLAM position error
Hall	1	0.092 m	0.067 m
	2	0.270 m	0.126 m
	3	0.054 m	0.086 m
House	1	0.107 m	0.050 m
	2	0.193 m	0.107 m
	3	0.227 m	0.094 m
Corridors	1	0.936 m	0.068 m
	2	0.657 m	0.271 m

Table 7.3: Position errors in the house, hall, and corridors environments between the LiDAR odometry and the positions computed with Graph-SLAM.

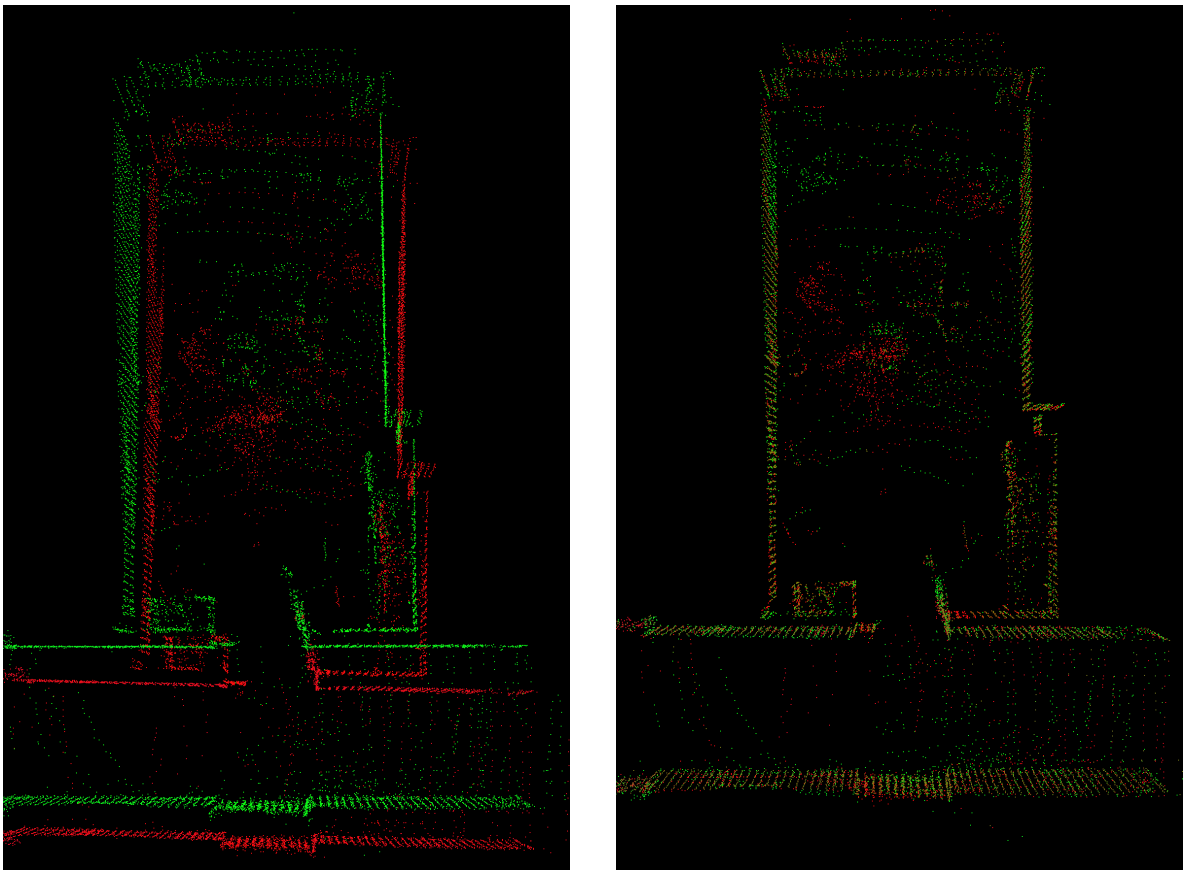
As we can see from Table 7.3, the loop closure process helps to reduce even more the error in the computation of the position of Spot. The effect is even more present in the corridor environment. But, we must not forget that these values are computed through AprilTag detection which is not a perfect process. As it can be seen in Fig. 7.5, the point clouds in the second test on the corridors are visually perfectly aligned after the loop closure. This gives reason to believe that the error in the positions computed with Graph-SLAM in Table 7.3 is mainly due to the AprilTag detection inaccuracies.

Table 7.4 shows the mean execution time for each point cloud processed by the Graph-SLAM algorithm. The results show that the average execution time is lower than the 0.2 s of delay needed between each data acquisition on Spot. As the maximum execution time is not too far from the target time of 0.2 s, it validates that the algorithm can be executed in real-time.

	Test	Mean execution time	Max execution time
Hall	1	104.5 ms	255.1 ms
	2	99.9 ms	299.7 ms
	3	102.1 ms	249.5 ms
House	1	72.8 ms	187.8 ms
	2	72.6 ms	160.3 ms
	3	72.1 ms	198.7 ms
Corridors	1	69.7 ms	165.3 ms
	2	70.2 ms	241.4 ms

Table 7.4: Mean and max execution time per point cloud processed by our Graph-SLAM algorithm on every environment.

Maps of the environments are showed in Fig. 7.6 and 7.7. They show that the environments are precisely reconstructed.



(a) Without loop closure

(b) With loop closure

Figure 7.5: Comparison between the Graph-SLAM process with and without loop closure. It can be seen that the loop closure process correctly closes the loop of the path. The start of the path is represented with the red point cloud and the end with the red one.

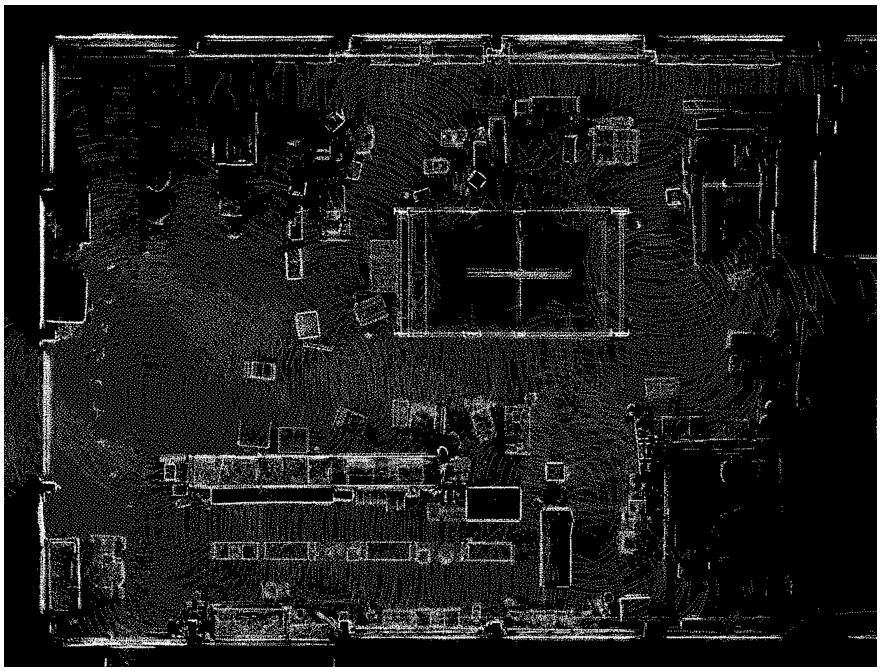


Figure 7.6: Top view of the map of the hall computed with the Graph-SLAM algorithm. A lot of details are preserved in the map

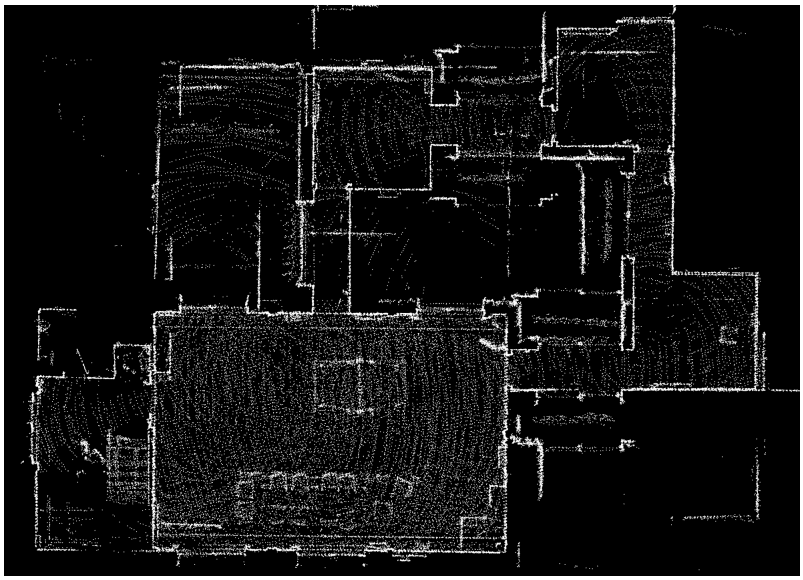


Figure 7.7: Top view of the map of the house computed with the Graph-SLAM algorithm. All the are consistent with each others.

7.2 Navigation results

To evaluate navigation performance, it is necessary to develop an architecture to transmit information from Spot to the path-planning algorithm in real-time. The solution is implemented in three blocks, as shown in Fig. 7.8. Firstly, the block representing Spot captures information from its location internally. The second block is a python client, which facilitates communication and data exchange to and from Spot. The communication is done through the API developed by Boston Dynamics and is mainly based on gRPC. Finally, the third block is a C++ server which is in charge of localisation, mapping and path-planning. The python client and the C++ server communicate via the gRPC protocol.

In this program structure, the python client will first make a data request to Spot. These data are point clouds, fiducial positions or robot odometry. Then, the python client will ask the C++ server to use this data to define the next action that Spot will have to perform according to the defined objective and the position of the robot. This order will be returned by the C++ server to the python client, which will transmit it to Spot. The choice of this structure is motivated by the fact that communication with Spot is much simpler in Python. It is also motivated by real-time constraints. It is therefore necessary to perform the costly processes in C++, and the communication with Spot in python, leading to this 3-block structure.

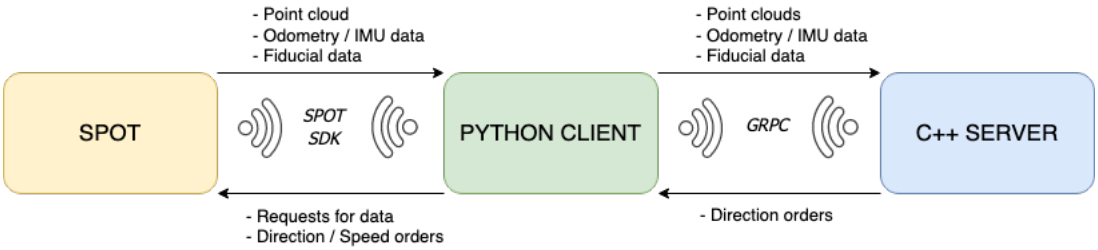


Figure 7.8: Global architecture of the program. Spot communicates with the python client thanks to the Spot SDK. The Python client sends and receives data from the C++ server thanks to GRPC protocol.

To test the autonomous navigation part, we give the path-planning a global map pre-computed by SLAM. During the computation of the global map, the robot positions its inertial marker with respect to an AprilTag present at initialization. Spot can then localize itself in this map with respect to a fixed inertial reference frame. We then give an objective to the navigation in terms of coordinates in the occupancy grid. The localization algorithm then receives the point clouds of the environment perceived by the robot. It uses the point cloud registration algorithm developed for the LiDAR odometry to align the point cloud to the map. The alignment is then used as the localization of the robot inside the map. The path-planner uses this localization information to recalculate the path in real-time at each call and gives orders to the robots according to the calculated result. The format of the given order is a

directional order towards the first jump point, at a constant speed. Thus, the orders given to the robot are adapted at each iteration. If the angle between the direction of the target point and the orientation of Spot differ by more than a threshold, the robot is first asked to turn on itself in the correct rotation. An example of the path followed by Spot is shown in Fig. 7.9. This visualization demonstrates that the achieved behaviour of the robot corresponds to its real behaviour. The contributions made by our improvements are therefore the following:

- Environment Mapping.
- Precise localization without diverging errors.
- Autonomous and untrained movement of Spot towards a defined goal.

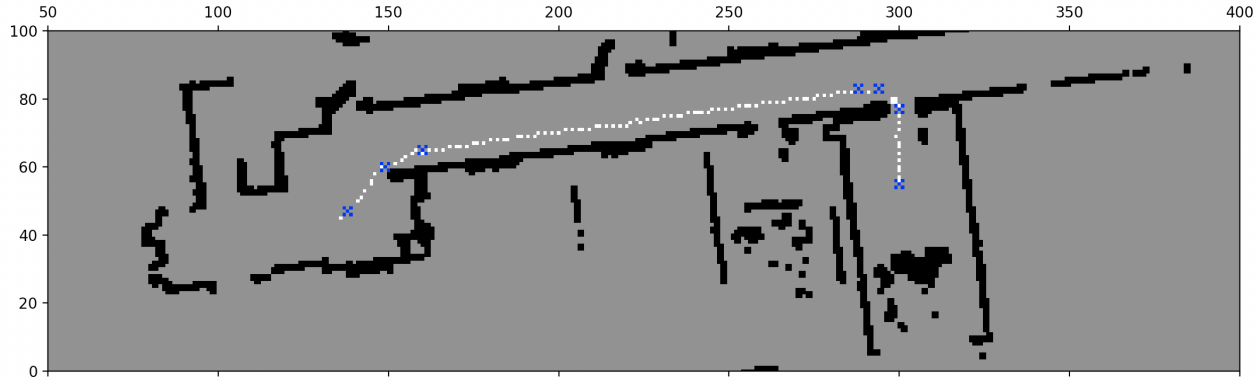


Figure 7.9: Visual representation of the test performed on Spot. In black the obstacles in the occupancy grid. The blue crosses represent the jump points calculated during the initialization. In white, the real path taken by Spot to reach its objective.

Chapter 8

Future improvements

This thesis develops effective solutions that form a basis on which many applications can be developed. There are therefore several possible improvements to the current implementation. On the one hand, these improvements concern the SLAM part of the navigation, and on the other hand the path-planning part. This chapter, therefore, focuses on a non-exhaustive list of the various upgrades we have considered.

- One of the big challenges in the SLAM problem is to be able to deal with dynamic environments. This ability to remove dynamic objects from the measurements could improve the robustness of the method and the quality of both the localization and the mapping. Our method reduces the impact of the dynamic objects by assigning a strict threshold in the outlier rejection function of the LiDAR odometry but it doesn't remove it. Solutions to the dynamic objects removal problem are often based on point cloud segmentation which can allow the tracking of objects between the point clouds to remove them.

- For the moment our method for detecting loop-closures is rather limited in its capabilities. It assumes that the chances that the robot will travel a long way without returning to a previously visited location are low. Using this assumption, loop closures can be detected in the direct neighborhood of a new node. However, this assumption cannot be maintained on large-scale construction sites. To be able to reposition correctly without having to put AprilTags everywhere, another method of loop-closure detection must be used. One idea is to detect features in the images acquired by the five stereo-cameras of Spot and compare them to detect potential loop-closure in the path of the robot.

- With the proposed implementation of Graph-SLAM, a node is added to the graph every 20 cm of the robot path. As the complexity of the graph optimization depends on the number of nodes,

this can become a problem for very long paths. One solution is to detect when we revisit a part of the path and stop adding new nodes in the graph. Also to allow an update of the map when revisiting, the point clouds of the nodes could be updated.

- For now our navigation method is limited to a 2D plane which restricts the capabilities to navigate through multiple floors of an environment. To lift this constraint another navigation algorithm that allows 3D path planning could be developed and used specifically on stairs.

- An important part of autonomous navigation is the ability to avoid dynamic obstacles. One possible solution with Spot is to use information from the local grid. This is represented by a fixed-size grid with areas where the robot can walk around it. This grid is updated in real-time. It is then possible to integrate this local grid into the grid representing the global map. As the path-planning algorithm is performed regularly, it would take into account the dynamic obstacles to update the movements to be made by Spot to avoid them. However, due to some major inconsistencies in the documentation in the Spot SDK provided by Boston Dynamics, it was not possible to implement this solution.

- In the paper on JPS+[41], the authors develop a solution to improve the speed of Jump Point Search. This solution consists in using a 32-bit word representation of the grid, where each bit represents the presence or absence of an obstacle. They then use bitwise operations to speed up the algorithm. We already implement the 32-bit word map representation, but not the bitwise operations. An acceleration of the path-planning algorithm is therefore possible.

- For a mobile robot operating in an environment such as a building under construction, it is important to have the safest possible movements. Being as far away from obstacles as possible is therefore desirable. In the current implementation, the path-planning calculates the optimal path, which then often passes very close to the obstacles. One idea would be to develop a different heuristic that would sacrifice optimality for safety. One solution is to calculate for each position in the map the distance to the nearest obstacle. This distance would then be added to the cost calculation:

$$f(n) = g(n) + h(n) + o(n)$$

where $g(n)$ is the so-far cost from the start, $h(n)$ is the current heuristic, i.e. the Euclidean distance between the node and the goal, and $o(n)$ is the distance to the nearest obstacle. A well-chosen design of the weights to be assigned to each part of the cost would allow Spot to move more safely.

Conclusion

Autonomous navigation of mobile robots in dynamic environments is challenging in many ways. Our objective was to make Boston Dynamics' Spot more autonomous in a construction environment where the topology is regularly changing. In this context, a solution to map an unknown environment and to locate itself accurately was developed through the SLAM algorithm. More precisely, the problem was tackled with a graph-based approach with the help of state-of-the-art optimization algorithms. This, coupled with the implemented improvements such as the loop-closure detection process and the feature-based point cloud registration algorithm, gives significantly better results for the localization than the original Spot odometry. It also allows to map accurately in real-time a construction site.

Furthermore, to use the mapped environment, an efficient path-planning solution has been developed through the Jump Point Search algorithm. The algorithm improves A^* by taking into account the topological specifies of grids used for robot navigation. We have also enhanced Jump Point Search by applying a filtering on its output. The filtering allows to reduce the number of turning points the robot should make on its path which improves the overall quality of the navigation. With our solution, it is now possible for an operator to direct Spot to move from point A to point B in an environment, without having to walk the path beforehand with the robot.

The output of our work is a framework and basis for many applications. It represents an efficient way to develop new solutions for more specific needs. The autonomous exploration of an unknown environment is an example of a goal that can be achieved using our solution. The monitoring of installations or parts of buildings in a decentralised way is another example.

We have also identified a number of improvements that can be made to the current solution, such as the management and avoidance of dynamic objects in the mapping and path-planning part, and multi-floors navigation.

Bibliography

[1] Boston Dynamics, "Spot SDK." https://dev.bostondynamics.com/, 2022. [Online; accessed 30-May-2022].

[2] Boston Dynamics, "About Spot — Spot SDK." https://dev.bostondynamics.com/docs/concepts/about_spot, 2022. [Online; accessed 30-May-2022].

[3] gRPC, "What is gRPC ? — gRPC." https://grpc.io/docs/what-is-grpc/introduction, 2022. [Online; accessed 1-June-2022].

[4] Boston Dynamics, "Networking — Spot SDK." https://dev.bostondynamics.com/docs/concepts/networking, 2022. [Online; accessed 30-May-2022].

[5] Boston Dynamics, "Geometry and Frames — Spot SDK." https://dev.bostondynamics.com/docs/concepts/geometry_and_frames, 2022. [Online; accessed 19-May-2022].

[6] C. Stachniss, "Graph-based SLAM using Pose Graphs (Cyrill Stachniss)." https://www.youtube.com/watch?v=uHbRKvD8TWgt=764s.

[7] V. Lidar, "Vlp-16 user manual." https://velodynelidar.com/wp-content/uploads/2019/12/63-9243-Rev-E-VLP-16-User-Manual.pdf.

[8] E. Olson, "Apriltag: A robust and flexible visual fiducial system," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3400 – 3407, 06 2011.

[9] N. Fiedler, *Distributed Multi Object Tracking with Direct FCNN Inclusion in RoboCup Humanoid Soccer*. PhD thesis, 06 2019.

[10] Boston Dynamics, "Autonomy services — Spot SDK." https://dev.bostondynamics.com/docs/concepts/autonomy, 2022. [Online; accessed 30-May-2022].

[11] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Intelligent Robotics and Autonomous Agents series, MIT Press, 2005.

[12] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*, ch. 46. Springer Handbook of Robotics, Springer Berlin Heidelberg, 2008.

[13] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[14] Wikipedia contributors, "Kalman filter — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Kalman_filter&oldid=1082910871, 2022. [Online; accessed 5-May-2022].

[15] Wikipedia contributors, "Particle filter — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Particle_filter&oldid=1075444767, 2022. [Online; accessed 6-May-2022].

[16] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "isam2: Incremental smoothing and mapping using the bayes tree," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 216–235, 2012.

[17] D. Nistér, O. Naroditsky, and J. Bergen, "Visual odometry," in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 1, pp. I–I, Ieee, 2004.

[18] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*, pp. 2564–2571, Ieee, 2011.

[19] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the seventh IEEE international conference on computer vision*, vol. 2, pp. 1150–1157, Ieee, 1999.

[20] D. Scaramuzza and F. Fraundorfer, "Visual odometry [tutorial]," *IEEE robotics & automation magazine*, vol. 18, no. 4, pp. 80–92, 2011.

[21] H. Zhu, B. Guo, K. Zou, Y. Li, K.-V. Yuen, L. Mihaylova, and H. Leung, "A review of point set registration: From pairwise registration to groupwise registration," *Sensors*, vol. 19, no. 5, p. 1191, 2019.

[22] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Sensor fusion IV: control paradigms and data structures*, vol. 1611, pp. 586–606, Spie, 1992.

[23] P. Babin, P. Giguere, and F. Pomerleau, "Analysis of robust functions for registration algorithms," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 1451–1457, IEEE, 2019.

[24] Y. Chen and G. Medioni, "Object modelling by registration of multiple range images," *Image and vision computing*, vol. 10, no. 3, pp. 145–155, 1992.

[25] R. B. Rusu, "Semantic 3d object maps for everyday manipulation in human living environments," *KI-Künstliche Intelligenz*, vol. 24, no. 4, pp. 345–348, 2010.

[26] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *Proceedings third international conference on 3-D digital imaging and modeling*, pp. 145–152, IEEE, 2001.

[27] A. Segal, D. Haehnel, and S. Thrun, "Generalized-icp.," in *Robotics: science and systems*, vol. 2, p. 435, Seattle, WA, 2009.

[28] C. A. Floudas and P. M. Pardalos, *Encyclopedia of optimization*. Springer Science & Business Media, 2008.

[29] J. Zhang and S. Singh, "Loam: Lidar odometry and mapping in real-time.," in *Robotics: Science and Systems*, vol. 2, pp. 1–9, Berkeley, CA, 2014.

[30] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[31] T. Shan and B. Englot, "Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4758–4765, IEEE, 2018.

[32] GTSAM, "GTSAM C++ Library." https://gtsam.org/, 2022. [Online; accessed 4-June-2022].

[33] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), IEEE, May 9-13 2011.

[34] Wikipédia, "Arbre kd — wikipédia, l'encyclopédie libre." https://fr.wikipedia.org/w/index.php?title=Arbre_kd&oldid=160130088, 2019. [En ligne; Page disponible le 14-juin-2019].

[35] Y. Hwang and N. Ahuja, "A potential field approach to path planning," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 1, pp. 23–32, 1992.

[36] M. M. Edsger Dijkstra, Laurent Beauguitte, "A note on two prob- lems in connexion with graphs," in *Numerische Mathematik*, vol. 1, 1959.

[37] P. N. Stuart Russel, "Heuristic functions," in *Artificial Intelligence, A Modern Approach*, vol. 4, p. 116, Pearson Education, 2021.

[38] D. Harabor, A. Botea, and P. Kilby, "Path symmetries in undirected uniform-cost grids," *SARA 2011 - Proceedings of the 9th Symposium on Abstraction, Reformulation, and Approximation*, 01 2011.

[39] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 25, pp. 1114–1119, 2011.

[40] N. Witmer, "A visual explanation of jump point search." https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html, 2013.

[41] D. Harabor and A. Grastien, "Improving jump point search," *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, vol. 2014, pp. 128–135, 01 2014.

[42] C. Yao and J. Rokne, "An integral linear interpolation approach to the design of incremental line algorithms," *Journal of Computational and Applied Mathematics*, vol. 102, pp. 3–19, 02 1999.

[43] Wikipedia contributors, "Bresenham's line algorithm — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Bresenham's_line_algorithm, 2022.

[44] C. Flanagan, "The bresenham line-drawing algorithm." https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html.

[45] P. Teleweck and B. Chandrasekaran, "Path planning algorithms and their use in robotic navigation systems," *Journal of Physics: Conference Series*, vol. 1207, p. 012018, 04 2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**