Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Drone Localization in Ad-hoc Indoor Environment

Master's Thesis

Author
Marcell Rausch

Advisor
dr.Gábor Fehér

May 10, 2020

Contents

Chapter 1

Introduction

A prerequisite for future expansion of mobile robots is the ability to navigate autonomously and safely in their ever-changing environment without collision. The positioning of drones in ad hoc indoor environment where no infrastructure is available is needed to fulfill this prerequisite. To know the position of a robot in an unknown environment, the robot needs to build a map if its surroundings so its position can be referenced. Mapping and positioning were considered as two separate problems at first and research focused on either of them, but the solution to these problems separately proved to be divergent. Only by considering these two problems together and finding solutions simultaneously, it became convergent [7]. This process is referred to as Simultaneous Localization And Mapping (SLAM).

The proposed issue by SLAM has many solutions depending on the use-case. SLAM technique can be used with a focus on improving localization accuracy in GPS degraded environments, by fusing the outputs of a SLAM algorithm with GPS and IMU data [10]. Or as presented in [5], the solution focuses on building highly accurate maps online of building interiors or even outdoors, but as a byproduct position becomes more accurate as well. The above mentioned papers present a mapping approach that uses industrial-grade hexacopter and octocopter and the same precise LIDAR scanner that can measure up to 100m.

Localization and mapping of the environment for a small quadcopters can be challenging, because adding any extra equipment means extra weight that shortens flight time and changes flight behavior and the onboard computation capacity is limited. Rotating planar scanners, that are also used in the above mentioned papers, undoubtedly provide highly accurate scans and therefore highly detailed maps. These LIDAR scanners are perfect for professional use and require reliably octo- or hexacopters to carry. The popular Velodyne VLP-16 for example weighs 830g. Adding it to a quadcopter that is under 1kg means significant extra weight and the drone might turn out to be uncapable of flying.

By using stationary LIDAR sensors placed around the quadcopter, these unwanted effects can be eliminated and the weight of extra equipment can be kept significantly lower, than using a 3D LIDAR scanner discussed before. The sensors need to be placed in a layout that offers an optimal scan of the environment, therefore the movements of quadcopter during flight needs to be taken into account to be able to scan the environment around the vehicle.

In this thesis the design of a 3D positioning and mapping system is being discussed, that uses stationary 1D LIDAR sensors to acquire distance measurements from an Unmanned Aerial Vehicle's (UAV) surrounding and uses SLAM technique to map its surrounding, es-

timate its position and velocity. The goal is to provide a lightweight hardware and software alternative to accurate planar LIDAR scanners, that are too heavy both computationally and physically for a UAV under 1kg. The focus during the design is on the quality of the map and accuracy of the position estimates.

In chapter 2 basic aviation terminologies are introduced that will be used in the thesis, followed by introduction of SLAM techniques and similar products on the market. Visual SLAM techniques describe the use of monocular cameras, stereo cameras or RGB-D cameras to be used for data acquisition, while ORB-SLAM and LSD-SLAM are two popular SLAM techniques for processing data coming from these sensors. Ranging sensors are non-visual sensors that can use sound or light to measure distance, like ultrasonic sensors or LIDAR sensors. Two interesting researches are discussed, that use LIDAR scanners to improve position or map quality online. Cartographer SLAM is a is a software package that will be used for the first prototypes and to evaluate layout design of the stationary sensors, because it has compatibility with ROS system.

In chapter 3 software components and their interaction is discussed, that will be used for future simulations. Simulations help to evaluate sensors, layout designs and SLAM algorithms without the need to build the system physically.

The last chapter 4 gives an overview of the chosen LIDAR sensor for the first prototype, first designs of the layout of stationary sensors and a unit for collection of measurements from the sensors.

Chapter 2

Literature review

2.1 Basic aviation terminologies

2.1.1 UAV or Drone

UAV is short for Unmanned Aerial Vehicle, in other words an airborne vehicle that is capable of movement without having a pilot on board. Drone is a subset of UAV, a vehicle that is capable of autonomous flight. Usually UAV and the word Drone are used interchangeably in the literature, and it is used so in this thesis.

2.1.2 Body axes used in aviation

In aviation Euler angles are used to describe the orientation of an aircraft. Euler angles are three angles that describe the orientation of a rigid body with respect to a fixed coordinate system. These axes are referred to as Yaw, Pitch and Roll. Rotation around these axes are called Yaw, Pitch and Roll angles respectively. All three angles follow the right-hand rule.

Magnetic north is used as reference for Yaw, so an angle of 0° or 360° means that the vehicle is heading North, 90° means it's heading East. Yaw is sometimes called Heading.

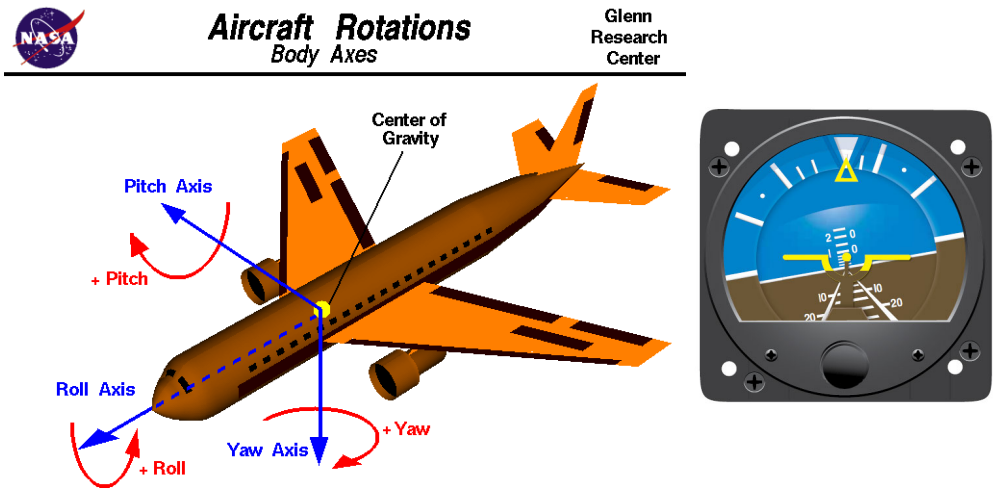


Figure 2.1: Aircraft body axes[13] and attitude instrument

Pitch and Roll angles are calculated in reference to the horizontal plane. The normal vector of the horizontal plane is used for the calculations, and it is the norm of the gravity vector. It is favorable to use the gravity vector, because its direction can be measured using an accelerometer. Both Roll and Pitch angles are measured from -90° to 90°. A Pitch of 90° is straight up and 0° is Horizon. If an aircraft flies with 0° Roll angle, the vehicle is horizontal, on the other hand a 90° Roll angle means the vehicle is turning right and is perpendicular to the horizon. Pitch is sometimes referred to as Tilt.

The limitation of using Euler angles is reached when Pitch or Roll angles approach 90°, because in this case one of these angles become parallel with the gravity vector and the other angle cannot be determined, due to lack of reference. Euler angles and singularities are well described in [3]. To avoid singularities of Euler angles, Unit quaternions can be used. Quaternions are mainly used for calculations, while Euler angles are used to provide humanly readable values.

2.1.3 Ground Control Station

A software running on a ground computer, used for receiving in-flight information via telemetry from a UAV. It displays status and progress of mission, that often includes sensor or video data. It can also be used for sending commands up to the UAV during flight.

Ground Control Station is often referred to as GCS.

2.1.4 Multicopters

Multicopter or Multirotor is a generic term to describe a UAV with more than two rotors. The term covers quadcopters, octocopters, hexacopters etc.

Quadcopters are quickly gaining popularity thanks to their easy usability, and because of commercially available drones that can be used for filming. Octocopters have eight rotors in total and mostly used by professionals. It provides a more stable flight, can lift more weight and it is safer, because it can fly with any of the rotors damaged.

2.2 Simultaneous Localization And Mapping - SLAM

Simultaneous Localization And Mapping is the computational problem that asks if it is possible to place a robot in an unknown environment and for the robot to incrementally build a consistent map of this environment while simultaneously determining its position. A good overview of SLAM is presented in[7] and [3] focusing on finding a solution to the above proposed problem in general, with no dependency on the type of sensor to be used.

During SLAM a robot is placed in an unknown location and it is capable of estimating its trajectory and location of all landmarks online, without the need of priori knowledge of the location. The robot makes relative observations of its surroundings, a number of unknown landmarks using sensors as seen on figure 2.2. The following quantities are defined at a time instant k:

- $\mathbf{x}_k$: State vector describing the location and orientation of the robot
- $\mathbf{u}_k$: Control vector, applied at *k-1* and drove the robot to state $\mathbf{x}_k$ at time k

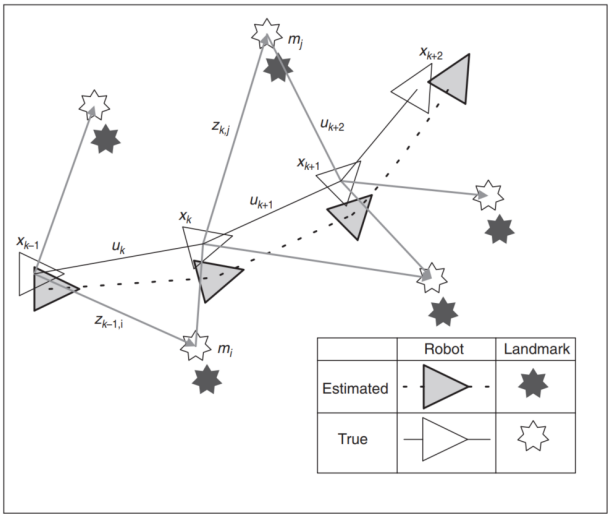


Figure 2.2: The essential SLAM problem presented in [7]

- $\mathbf{m}_i$: Landmark location vector. A time independent vector that describes the location of a static landmark
- $\mathbf{z}_{ik}$: Observation taken from the robot to the ith landmark at time k
- $\mathbf{X}_{0:k} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k\}$ A set of all robot location vectors until time k
- $\mathbf{U}_{0:k} = \{\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_k\}$ A set of history of all control vectors until time k
- $\mathbf{m} = \{\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_n\}$ A set of all landmarks
- $\mathbf{Z}_{0:k} = \{\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_k\}$: A set of all landmark observations until time k

In probabilistic form of SLAM, the probability distribution function 2.1 needs to be computed for all times k. In other words the current state vector and the landmark location probabilities need to be calculated based on all k observations, control vectors and the initial state of the robot.

$$P\left(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0\right) \tag{2.1}$$

A recursive solution is desirable for the SLAM problem. Starting with an estimate for the posteriori distribution (equation 2.2) at time k-1, then a control vector $\mathbf{u}_k$ is used to estimate the next state and observations using Bayes theorem. This calculation requires an observation model and a state transition model.

$$P\left(\mathbf{x}_{k-1}, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k-1}, \mathbf{x}_0\right) \tag{2.2}$$

The observation model 2.3 describes the probability of making an observation $\mathbf{z}_k$ given the current robot and landmark locations.

$$P\left(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{m}\right) \tag{2.3}$$

The state transition model 2.4 describes the probability of the next position of the robot based on the previous state and the control vector at time k. The state transition is assumed to be a Markov process, so the next state $\mathbf{x}_k$ depends only on the current state

$\mathbf{x}_{k-1}$ applied control $\mathbf{u}_k$. $\mathbf{x}_k$ is also independent from both landmark locations $\mathbf{m}$ and observations $\mathbf{z}_k$.

$$P\left(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k\right) \tag{2.4}$$

Using equations 2.2, 2.3 and 2.4, the SLAM algorithm can now be described using a two-step recursive form: prediction and correction. The prediction step or time-update is used for estimating the next state and map, based on all posteriori observations, control vectors and initial state.

$$P\left(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, \mathbf{x}_0\right) = \int P\left(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{u}_k\right) \times P\left(\mathbf{x}_{k-1}, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k-1}, \mathbf{x}_0\right) d\mathbf{x}_{k-1} \tag{2.5}$$

The measurement update provides a correction to the prediction state, using observations made at time k.

$$P\left(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0\right) = \frac{P\left(\mathbf{z}_k \mid \mathbf{x}_k, \mathbf{m}\right) P\left(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, \mathbf{x}_0\right)}{P\left(\mathbf{z}_k \mid \mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}\right)} \tag{2.6}$$

The localization problem can be solved with the assumption the map is known:

$$P\left(\mathbf{x}_k \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{m}\right) \tag{2.7}$$

And the mapping problem can be solved with the assumption the location is known:

$$P\left(\mathbf{m} \mid \mathbf{X}_{0:k}, \mathbf{Z}_{0:k}, \mathbf{U}_{0:k},\right) \tag{2.8}$$

These problems introduced in 2.7 and 2.8 are dependent on each other and need to be solves simultaneously. In the early stages of SLAM mapping and localization were tried to be solved independently and work often focused on either of them. Once the realization came that combined mapping and localization can be formulated as a single estimation problem, the problem became convergent.

Many solutions can be found for probabilistic SLAM problem, the most common is the use of the extended Kalman filter (EKF) that provides an optimal estimation to non-linear models with additive Gaussian noise. The motion model can also be represented with a non-Gaussian probability distribution, that leads to the use of the Rao-Blackwellized particle filter, or FastSLAM algorithms.

The above introduced SLAM problem was a general introduction, it is not dependent on the sensor used for observations. Many kind of observers can be used for this purpose, but the two main categories are camera based and ranging sensor based. Visual observers mostly use simple or special camera configurations for depth sensing, while ranging sensor based observers use a physical phenomena to measure distances like sonars or LIDARS.

2.2.1 Visual SLAM

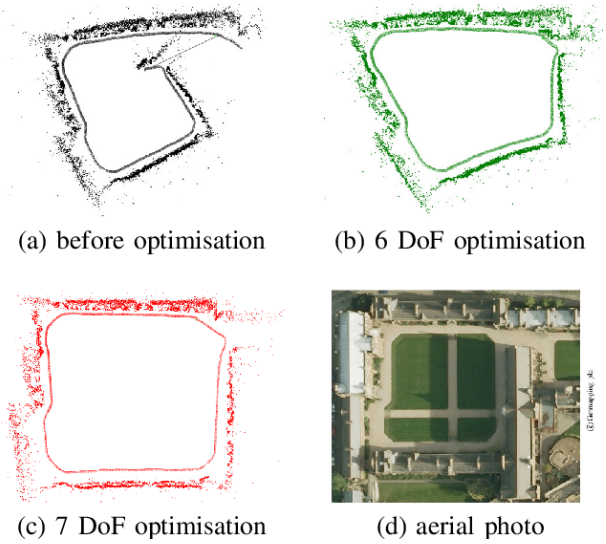
Visual SLAM is a subset of SLAM algorithms that processes camera images to determine the camera's position and orientation relative to its surroundings, while building a map of its environment. In the simplest case a monocular camera is used to collect images,

but more advanced active or passive stereo cameras and RGB-D cameras are also used. A pro of using images for SLAM is that images contain high amounts of data, therefore very detailed maps can be made. On the other hand, processing big amounts of data means higher complexity and demands higher processing capabilities.

Two main categories of visual techniques are feature-based and direct SLAM algorithms. Feature-based techniques detect certain key-points called features in images, like corners and edges, and only use these features to extract depth information. Direct SLAM algorithms however don't search for features, but use the regions of high intensities on the image to estimate the location and surroundings. Two popular visual SLAM methods are feature-based ORB-SLAM and direct LSD-SLAM.

2.2.1.1 Monocular camera

The pro of using monocular camera for localization and mapping is that each image contains high density of information and cameras will always be cheaper easier to use than other depth sensing camera rigs. A big pro, but in the same time a big con of using a monocular camera for SLAM is that a single camera cannot detect the scale. It is an advantage, because the same camera can be used indoor on a small scale even down to centimeters, and used outdoors on a scale of kilometers. Nonetheless during transmission between these scales, the scale can drift and therefore size of landmarks will not be presented correctly. This is called scale-drift, that algorithms try to compensate.



(a) before optimisation (b) 6 DoF optimisation

(c) 7 DoF optimisation (d) aerial photo

Figure 2.3: Scale drift and compensation illustration from [17]

2.2.1.2 Stereo camera

A stereo camera is a type of camera with typically two image sensors. Two image sensors mimic the human binocular vision and makes depth perception possible. The two cameras are placed relatively close together so there is a high correlation between the images taken by these cameras. To find correlations between the two images sufficient details and structure needs to be present on the images, therefore light conditions affect the performance of depth-sensing. This technique is also called passive stereo vision.

Active stereo cameras solve this problem employing a projector onboard, that projects a structured light to simplify the stereo matching problem. Using this technique the depth sensing performance is less affected by lighting conditions and reoccurring patterns will not confuse the matching of the images.

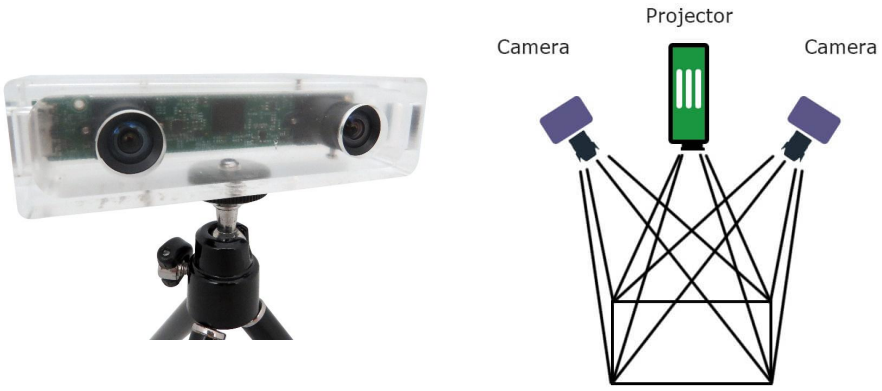


Figure 2.4: Passive stereo camera on the left, concept of active stereo camera on the right

2.2.1.3 LSD-SLAM

LSD-SLAM for monocular cameras were proposed in [8] and for stereo or RGB-D cameras in [9]. LSD stands for Large-Scale Direct SLAM and it can be seen by the name that it is a direct SLAM algorithm. The map of the surroundings is created based on keyframes that consists of three fields, a camera image, an inverse depth map and the variance of the inverse depth map. The solution of the problem proposed by SLAM is divided into three parts: tracking, depth map estimation and map optimization as seen on figure 2.5.

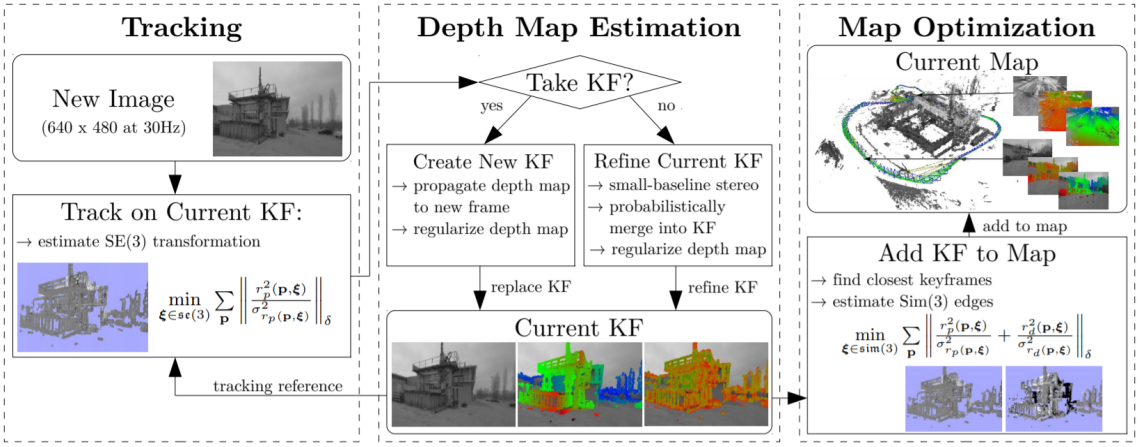


Figure 2.5: LSD-SLAM overview of the complete algorithm [8]

New images are input during the tracking phase and the position is estimated with respect to the current keyframe from previous iteration. The depth map estimation component uses tracked frames to either refine or replace the current keyframe. If the camera has moved more than a set threshold, a new keyframe is initialized with the combination of current and new keyframe. If the tracked keyframe is not taken, then it is used to refine the

current keyframe by performing small-baseline stereo comparisons and regularization of the depth map. Once the current keyframe is updated, it is built into the global map, by the map optimization. Keyframes that are close to each other can be detected and closed, this process is called loop-closure. For loop closures and scale-drift compensation, similarity transform is used. The map is further optimized using using pose graph optimization from g2o package.

2.2.1.4 ORB-SLAM

ORB-SLAM is a feature-based SLAM algorithm introduced in [12]. The features extracted are represented by ORBs, therefore the name of the algorithm. ORB-SLAM has three threads that run in parallel: tracking, local mapping and loop closing.

The tracking phase is responsible for localizing the camera and to decide when to insert a new keyframe. The features extracted from the new image are FAST corners, where FAST is a computationally efficient process to find corners on images. These corners are described using ORB, that is a method to describe FAST corners by a center of mass and an orientation parameter. This way the place and orientation is known of each feature. The initial pose is estimated using a constant velocity model and if the tracking is lost, relocation is done instead of pose estimation. Track local map is a map of keyframes that share a common map point. Finally it is decided if new keyframe is need to be created.

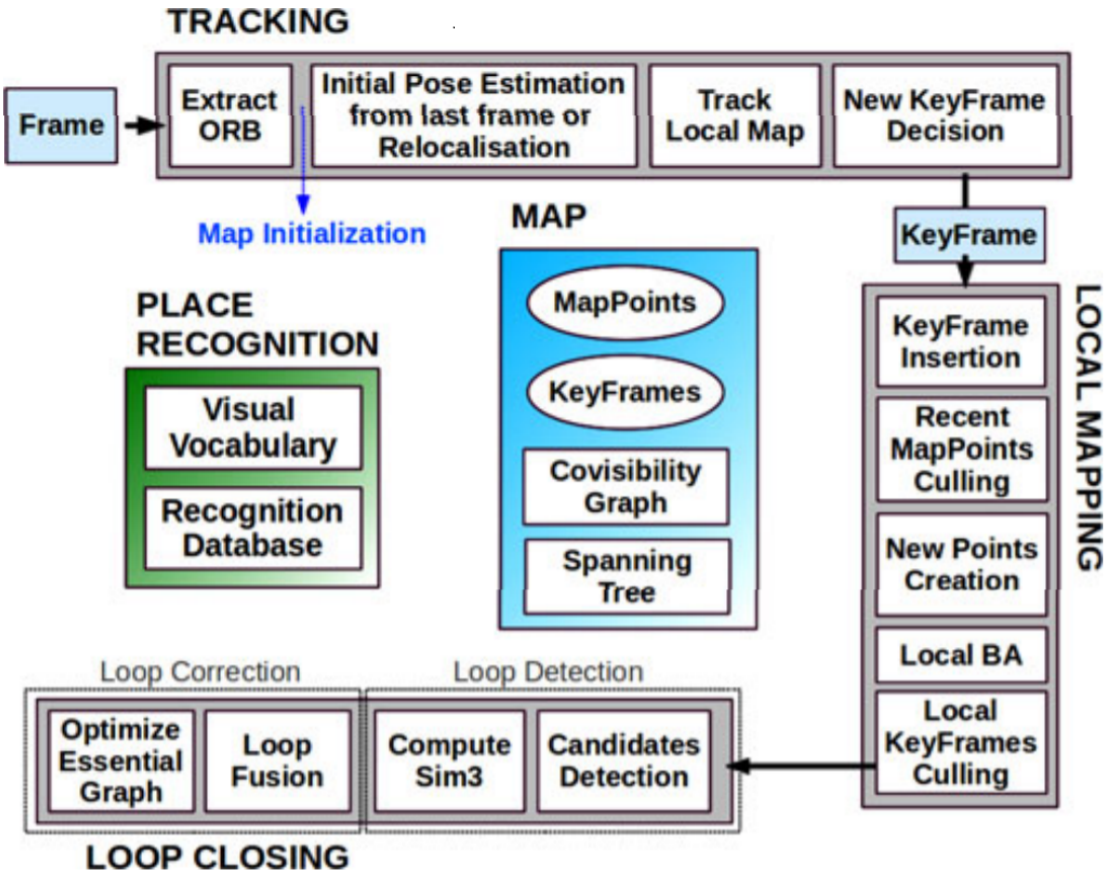


Figure 2.6: ORB-SLAM overview [12]

In the first step of local mapping the new keyframe is inserted into the co-visibility graph and it is linked keyframe to keyframe with the most points in common. ORB features from

the connected keyframes are used to calculate new map points using triangulation. Unmatched ORBs are compared with other unmatched ORBs of other keyframes. Projection error and consistency is checked, then the map is projected to the matched keyframes. The new map points need to be validated, by going through a test to increase the likelihood that these map points are valid part of the map. In local bundle adjustment step, the current keyframe and connected keyframes through the co-visibility graph are optimized. Finally abundant keyframes get discarded to keep simplicity.

Loop closing is quite effective in ORB-SLAM, because matching ORBs can be done with relatively high confidence. Similarity transformation is calculated.

2.2.2 Ranging sensor based SLAM

Cameras are cheap, popular and can be used to create highly detailed maps, but they produce high amounts of data that demands high processing capabilities. Ranging sensor based approaches produce lower amount of data that is dependent on the scale, these sensors have a minimum and maximum range they can measure in between.

Ultrasonic sensors have been used for distance measurements for many years, but they suffer from significant measurement noise, that comes from background noise and the variability of speed of sound. Almost no development has been seen on ultrasonic sensors in the previous years LIDAR sensors on the other hand are actively developed, because this technique uses light to measure distance, offer more accurate measurements, typically lower form factor, higher resolution and update rates. LIDAR sensors are also significantly more expensive than most of ultrasonic sensors and camera based SLAM systems.

A planar laser scanner is a LIDAR sensor that is extended to make distance measurements in 2D. Typically it is used for scanning the horizontal plane. Usually the scanning is solved by rotating a LIDAR sensor in 360° while it is making measurements evenly distributed on the plane. Planar scanners are good for building floor maps and navigating in indoor environments. Some scanners are further extended for 3D scanning.

2.2.2.1 3D LIDAR SLAM to improve localization accuracy

The study of University of California[10] presents an algorithm that fuses LIDAR, IMU and GPS signals to have a more accurate estimation of the absolute position and velocity of a UAV. The LIDAR scanner provides local position updates using SLAM technique,

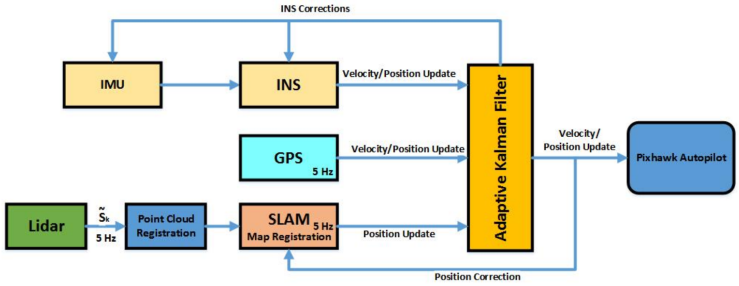


Figure 2.7: 3D LiDAR SLAM Integration with GPS/INS for UAVs in Urban GPS-Degraded Environments [10] - algorithm overview

GPS provides corrections when available and an Internal Navigation System is used as an

additional input to the Adaptive Extended Kalman filter. The outline of the filter can be seen on figure 2.7.

The 3D LIDAR scanner used is Velodyne VLP-16, that has a vertical field of view of 30° with a vertical resolution of 2°. On the horizontal axis the resolution is 0.1-0.4° and rotation rate can be adjusted between 5-20Hz. The sensor weighs 830g according to the company website.

During the experiment an octocopter is used to carry the sensor. Octocopters are stable and can carry high amounts of weight. The proposed filter is capable of reducing the GPS drift of 24.3m and LIDAR drift of 7.5m to 3.42m. This shows that fusing GPS, LIDAR and IMU measurements result in a more accurate position estimate in GPS-degraded environments. The capabilities of the LIDAR was also tested on an even moon-like surface, with very few landmarks that can be used by the SLAM algorithm. In this case GPS data is much more accurate than LIDAR SLAM, therefore LIDAR data is not used for calculation of position updates in these cases.

	Drift	Error %
GPS	24.3m	6%
LiDAR	7.5m	1.8%
AEKF	3.42m	0.84%

Figure 2.8: GPS, LIDAR and EKF position estimates and position drift after 405 meters[10]

2.2.2.2 3D LIDAR SLAM for mapping

The solution seen in 2.2.2.1 uses a LIDAR scanner to improve position and velocity data, by fusing the output of SLAM algorithm with sensors. The map created simultaneously is not used in that solution. 3D LIDAR data and SLAM can be also used to create highly detailed maps and this paper presented by David Droeschel and Sven Behnke [5], discusses an efficient implementation of SLAM with focus on improving online mapping quality.

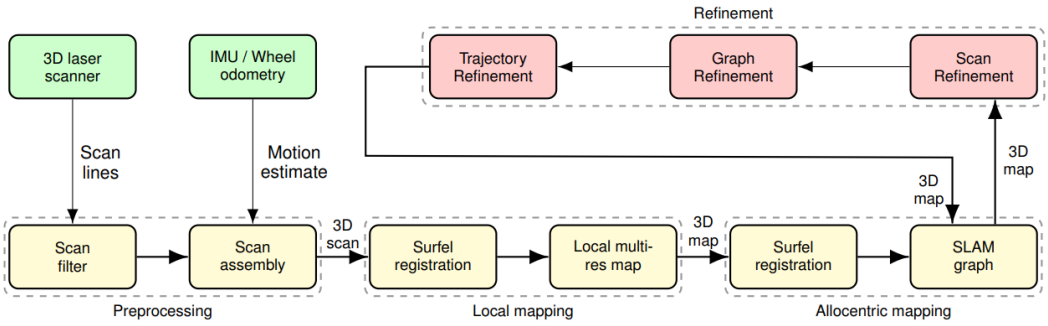


Figure 2.9: Efficient Continuous-time SLAM for 3D LIDAR-based online mapping[5] - algorithm overview

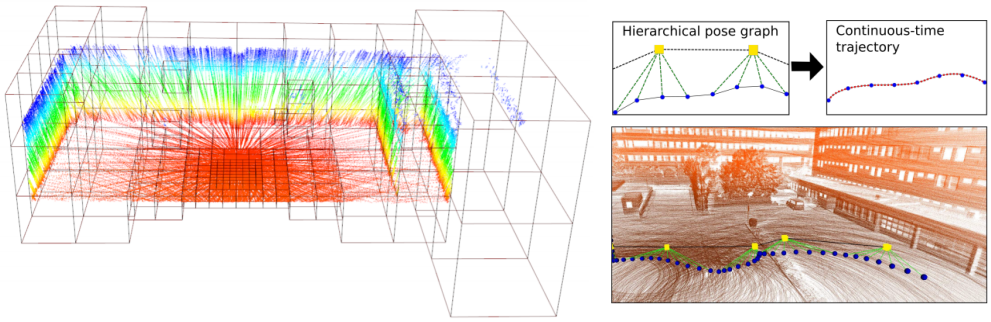


Figure 2.10: Grid-map presented in [6] on the left, estimated trajectory and built map on the right

The system first aggregates range measurements into a robot-centric grid-map [6], that has high resolution in the close proximity and lower resolution with increasing distance. A grid cell stores the cell's occupancy probability and a surface element(surfel), that describes the mean of samples and covariance. IMU or odometry measurements are used to account for motion of the sensor during acquisition, to compensate rolling shutter artifacts. After the preprocessing step, surfels become registered and the local map is updated. The older local map measurements are overwritten with the current 3D scan. During allocentric mapping phase, the 3D local multiresolution maps from different view poses are aligned with each other using surfel based registration. Local maps that are registered to the SLAM graph are subject to refinement. During these steps, the scans are refined based on available local maps, then graph and sensor trajectories are also refined. Local mapping, allocentric mapping and refinement phases are independent from each other, so these can run parallel.

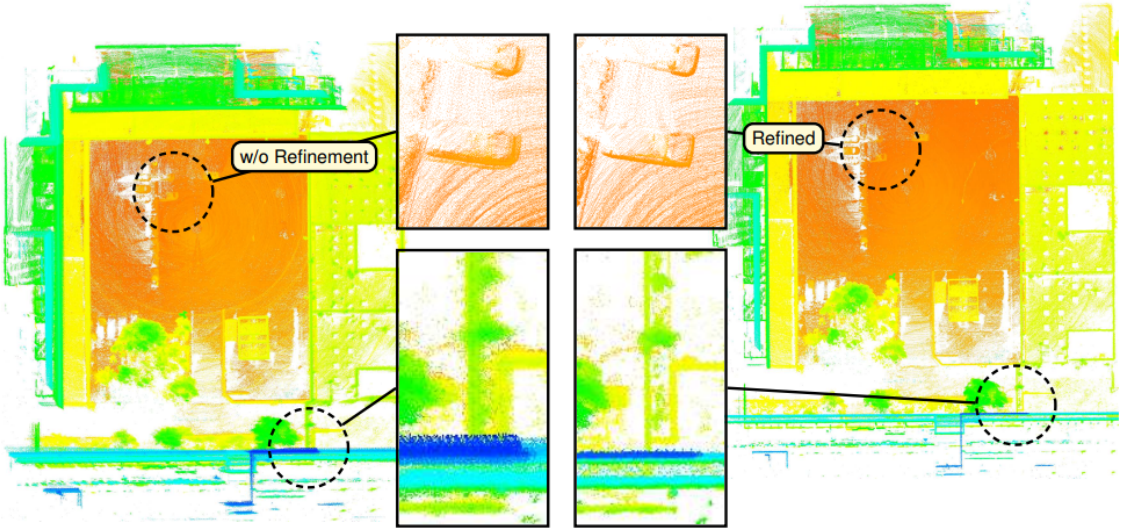


Figure 2.11: Comparison of maps with and without refinement[5]

2.2.3 Cartographer SLAM

Cartographer is a system developed by Google that provides real-time simultaneous localization and mapping in 2D and 3D across multiple platforms and sensor configurations.

Cartographer is used by Google street view for mapping of building interiors, using a backpack mounted LIDAR scanner.

There are officially available SLAM packages in ROS, the most popular packages are Gmapping, Cartographer and Hector SLAM. For the scope of this thesis, only ROS compatible packages are being concerned. Gmapping and Hector SLAM are popular choices for SLAM among developers and even unofficial 3D extensions are available, but officially they only support 2D mapping. Cartographer supports 3D SLAM without any modifications and for this reason Cartographer has been selected. For some yet unforeseen if Cartographer is proved to be incapable of handling the selected sensors, other SLAM algorithms can be considered.

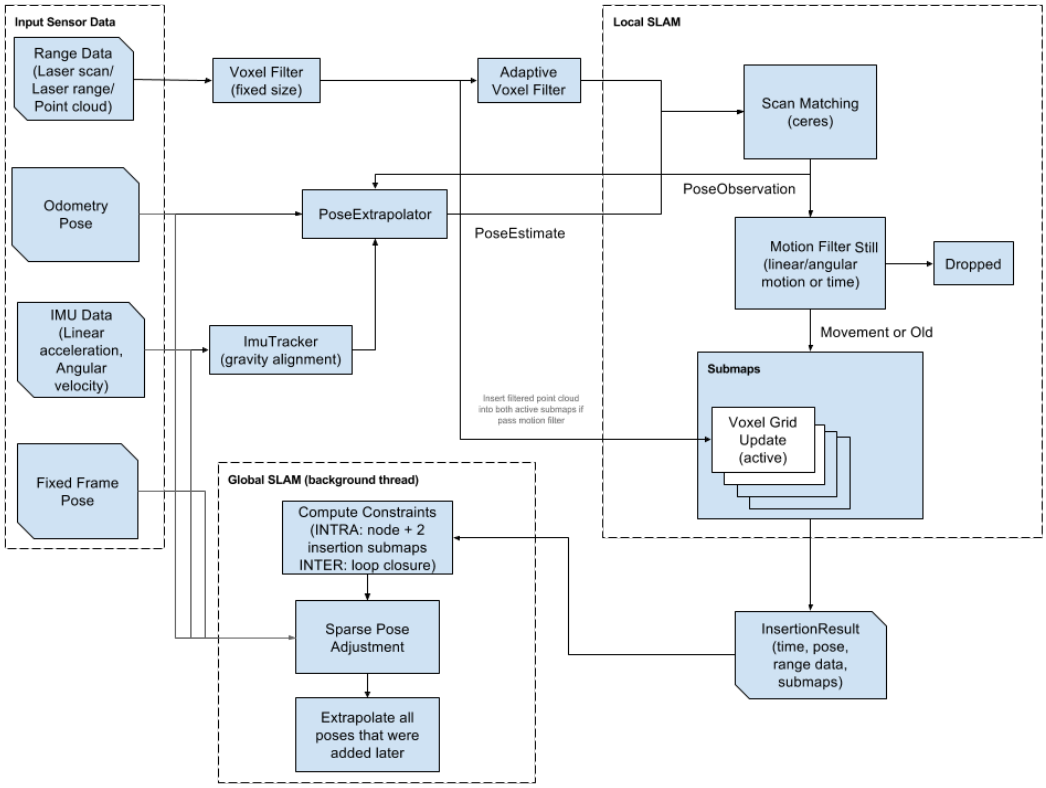


Figure 2.12: Cartographer SLAM overview[16]

On the other hand Cartographer ROS package supports both 2D and 3D SLAM and has a great and active community. The overview of Cartographer is seen on figure 2.12. The core components are sensor inputs, Local SLAM and Global SLAM. On a higher abstraction the job of local SLAM is to build submaps and the job of global SLAM is to tie submaps the most consistently together. It is common to build submaps or local maps in SLAM implementations, because by updating the whole map increases complexity quadratically, but by using local maps or submaps the complexity can be limited to the contents of the submap that is preferable during online processing. In this case alignment of submaps needs to be solved, but it requires less processing and it can be run in parallel.

The most important input of Cartographer is the Range Data, range measurements coming from LIDAR sensors. The data is first pre-filtered, because some measurements are irrelevant, the sensor might be directed to a part of the robot or it can be covered by dust, and some sensors set unsuccessful range measurements to a value that is significantly higher than the maximum range of the sensor. These outliers need to be filtered

and Cartographer uses a bandpass filter for pre-filtering, that keeps the values in a predefined range. The minimum and maximum values need to be set according to the sensor specifications.

Close objects are very often hit by the LIDAR measurements and offer more points, however distant objects offer much less points. The Voxel filter down-samples raw points where density is higher, but leaves scarce measurements to reduce computation needed, while keeping data integrity.

Inertial Measurement Units (IMU) provide a direction of gravity vector and a noisy estimation of the robot's movement. IMU data has to be provided for 3D SLAM because it greatly reduces complexity of scan matching, while IMU data is optional for 2D SLAM. Odometry Pose and Fixed Frame Pose are optional inputs of Cartographer, that can further reduce complexity.

In Local SLAM phase, first scans are inserted into a submap at the best estimated position. Scan matching is done only against current submap and not against the global map. During this process the error of pose estimates accumulate in the world frame. To reduce the accumulated error over time, pose optimization is run regularly. All finished submaps are considered for loop closure, that happens on a background thread Global SLAM.

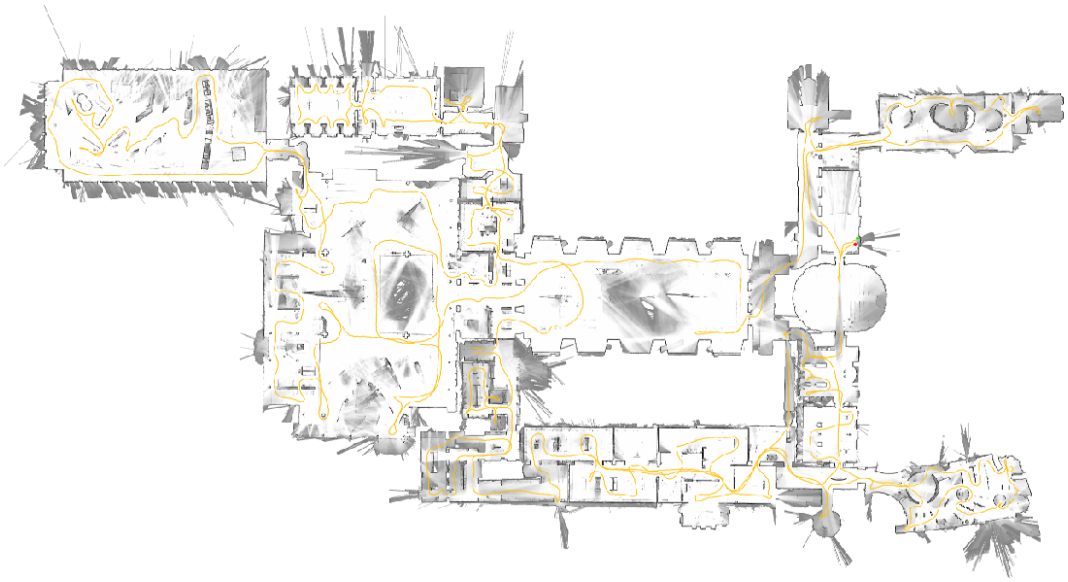


Figure 2.13: Cartographer SLAM example[16]

2.3 Similar products available on the market

2.3.1 Terabee TeraRanger Tower

TeraBee offers an off-the-shelf solid-state LIDAR system with the purpose of collision avoidance for drones. In their solution 8 sensors are evenly distributed around the vertical axis with a controller board in the middle. TeraRanger's interface is compatible with Pixhawk 4 flight controller board and PX4 flight controller software, that makes integration easy into systems based on these components.

Each block of the array is a standalone LIDAR sensor, that can be used separately and supports different mount configurations. The company offers a long-range and fast-ranging

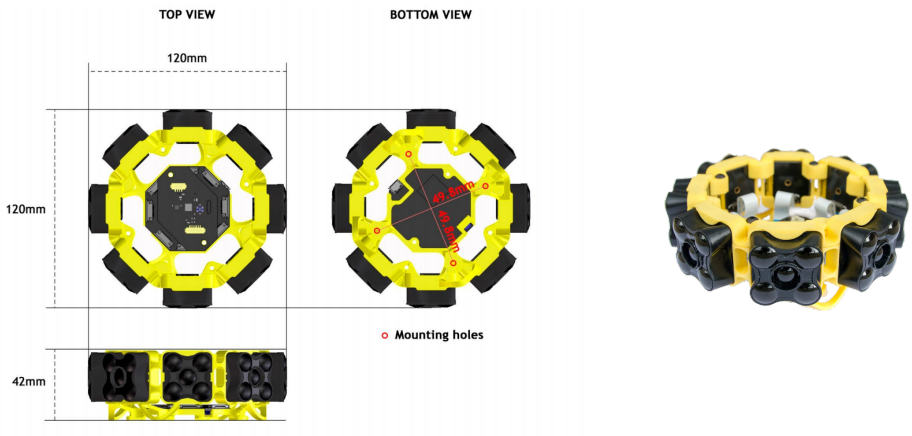


Figure 2.14: Terabee TeraRanger Tower Evo dimensions

version of these sensors and depending on the type an update rate of 320 Hz can be achieved. The fact that system comes with a ROS package and a 2° field of view makes it a potential candidate for indoor mapping and positioning in two dimensions. The price of this setup starts from 599 euro[19].

	Long-range	Fast-ranging
Range	0.5m up to 60m	0.75m up to 8m
Update rate	120Hz/sensor	320Hz/sensor
Field of View	2°	2°
Accuracy	$\pm 4cm$ in the first 14m, 1.5% above	$\pm 12cm$

Table 2.1: TeraRanger Tower Evo specifications

Terabee provides a tutorial video on their website[19] how to connect the sensor array to a UAV that uses Pixhawk 4 flight controller and the process of configuration in two different GCS programs. I have learned that PX4 flight stack supports lidar measurements for obstacle avoidance and it can be configured from a GCS program. It seems a reasonable choice to integrate this feature into such product.

2.3.2 Crazyflie Multi-ranger deck

The company Bitcraze has developed a mini quadcopter mainly for educational purposes. The current version is called Crazyflie 2.1 and measures only 92x92mm with a height of 29mm and a weighs 27g. Extra sensors and peripherals can be attached to the top of the quadcopter using extension boards.

The extension board called Multi-ranger deck has 5 VL53L1X sensors by STMicroelectronics facing forwards, backwards, left, right and up. This project is similar to the product of Terabee described in 2.3.1, but in a smaller size factor and with significantly lower weight.

An introduction video can be found on the product website [11], where a SLAM algorithm is used to create a map and localize the drone. This serves as a proof of concept, that stationary VL53L1X sensors can be used for mapping and positioning in two dimensions. The company provided no information of the SLAM algorithm used or from the quality of the map.

Figure 2.15: Crazyflie Multi-ranger deck, SLAM example project

2.3.3 Skydio 2

2.3.4 Velodyne Velobit

Chapter 3

Components review

Before actually building the system and experimenting on an expensive quadcopter in real world environment, it is cheaper, safer and easier to do tests in a simulated environment.

The PX4 Firmware repository[1] contains Gazebo simulation files to give developers a head start in simulation of their product. Upon these simulation quadcopter models LIDAR sensors can be placed and moved around with ease, without the need of wiring or external infrastructure. This simulation is suitable for Software in the loop (SITL) testing, meaning that the same software developed for the simulation can be used on a real quadcopter with little or no modifications.

3.1 Robot Operating System

Robot Operating system (ROS) is a set of software libraries and tools that help developers to build robust general-purpose robot applications[18]. ROS provides hardware abstraction of the underlying robot, generalizes the interfacing of these robots and allows simple high-level usage.

The core component of ROS is a public-subscribe communication protocol similar to MQTT protocol used on the field of IoT. The ROS system is comprised of a number of independent communicating parties called nodes. Each node can subscribe to multiple topics and receive a stream of messages published to these topics by other nodes. For example a temperature filter node might subscribe to a topic of "/temperature_sensor" and publish the filtered temperature values to a topic called "/filtered_temperature_sensor". The benefit using this solution is that the nodes achieve communication exclusively via this protocol so each node lives independently from another. Nodes in ROS do not need to be on the same system or even on the same architecture. Nodes can be run on different computers or even on microcontrollers or smartphones making the system flexible.

To start a ROS session, a ROS Master needs to be started first. After starting the master, nodes can be started and each one registers the topics it wants to subscribe/publish to. The master handles these requests and helps the nodes to establish connection between each other so communication can be started.

ROS is language independent so nodes can be implemented in different programming languages. While C++ is the most common language for product development, official Python wrappers can be used for fast prototyping.

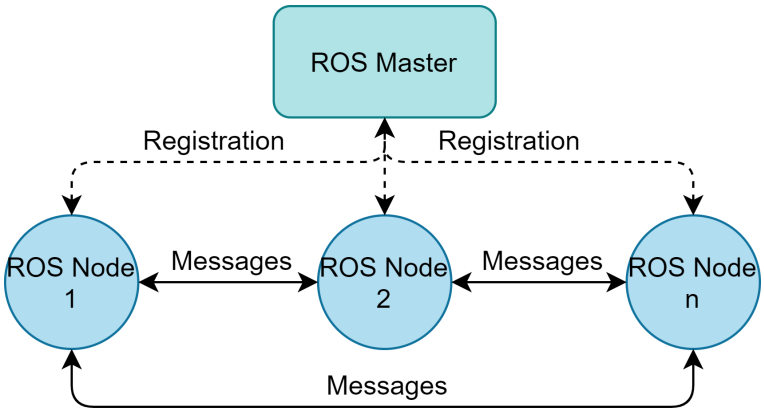


Figure 3.1: ROS publish subscribe communication

3.2 Gazebo Simulator

Gazebo[15] is an open-source 3D simulator for robotics, with the ability to accurately simulate robots in complex indoor or outdoor environments. It is similar to game engines, but produces more realistic, physically correct behavior. Gazebo is free, open-source, actively developed and has gained high popularity during the last years.

Gazebo comes with a growing number of robots and sensors. For simulations one can choose from the officially available robots or create a custom robot using SDF files. Any of these robots can be customized and any number of sensors can be added. Using simulations sensor data can be generated and development can be started without having the actual hardware.

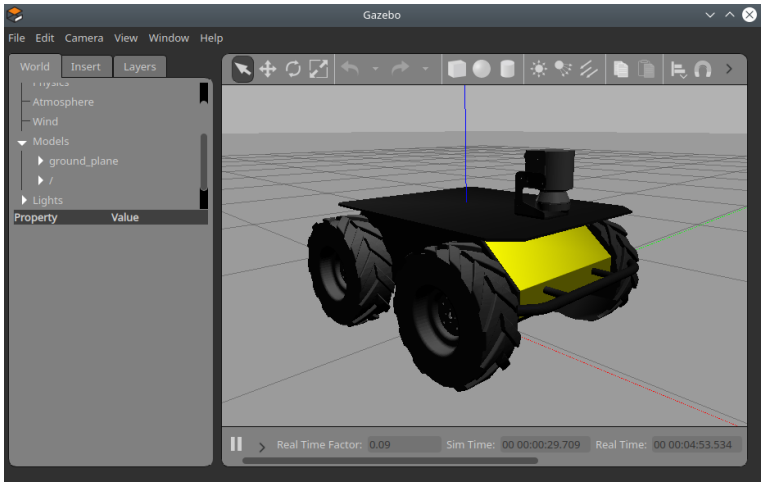


Figure 3.2: Gazebo ROS Husky robot demo

Why Gazebo is particularly interesting, is because it can be started in a ROS compatible mode. Gazebo started with ROS wrapper will actively channel all sensor readings, states and many more to ROS topics, allowing nodes outside of the simulation to subscribe and publish to these topics. Using this technique nodes can be developed in a truly hardware independent manner. The simulation can be replaced by a real robot and the nodes developed during this phase can be used without changes on the real hardware. For example a Husky robot can be simulated in Gazebo and controlled through ROS. The Gazebo simulation of a Husky can be seen on figure 3.2.

3.3 PX4 Autopilot Software

PX4 is a powerful open-source autopilot flight stack. PX4 can control many different vehicle frames and vehicle types, supports wide variety of sensors and peripherals. The project is a part of Dronecode, a nonprofit foundation working on open-source projects to achieve standardization of drones.[2]

The core component of PX4 is the flight stack or flight controller that controls the motors based on sensor measurements and executes commands received from ground control. Besides the flight stack the repository also contains Gazebo simulation files of multiple vehicles and an extensive `Makefile` that makes starting all components of the simulations relatively easy.

3.3.1 PX4 simulation using MAVROS

The logical units of the simulation can be seen on figure 3.3. By executing a `make` command the Gazebo Simulator, the PX4 SITL flight stack and the API/Offboard units are started at once. The PX4 SITL flight stack connects on a TCP port to the simulator, receives sensor data, calculates and sends motor and actuator values to the simulated vehicle. There are two options for the navigation of the simulated drone: using an Offboard API or a Ground Control Station. For autonomous flight control an Offboard API needs to be used and MAVROS is a suitable choice, it basically offers a ROS interface for the drone. It forwards all messages from the flight stack to ROS topics and vice versa.

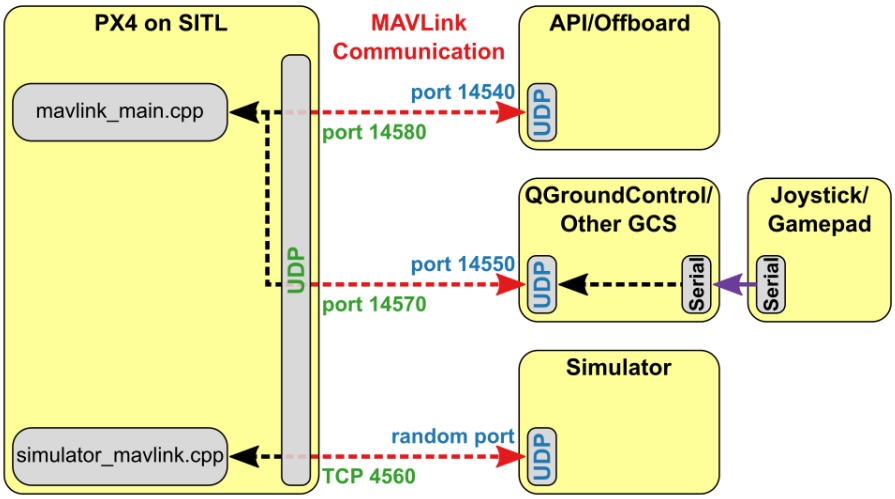


Figure 3.3: PX4 SITL Simulation Outline [4]

3.3.2 PX4 simulation using Gazebo with ROS wrapper

Drone models are described using SDF files that are XML based and by modifying these files new sensors can be added. Gazebo loads the selected vehicle and the sensors, but if a selected sensor is not supported by PX4, its measurements will not be forwarded to MAVROS and therefore to ROS topics. It can be troublesome to always use sensors that are supported by the flight controller, not to mention that the MAVLink protocol is designed for small messages so the bandwidth can be low in some cases. Luckily Gazebo can be started with a ROS wrapper and therefore all sensors can be accessed on ROS topics.

However this technique cannot be used on a real drone with the original PX4 firmware, because it purely relies on MAVLink messages, but for simulation purposes it's an elegant solution. This modified overview of the units used for simulation and a screenshot of an Iris drone simulated in Gazebo can be seen on figure 3.4 and 3.5. MAVROS is still needed to be run, because there are some messages that would not be published to ROS otherwise.

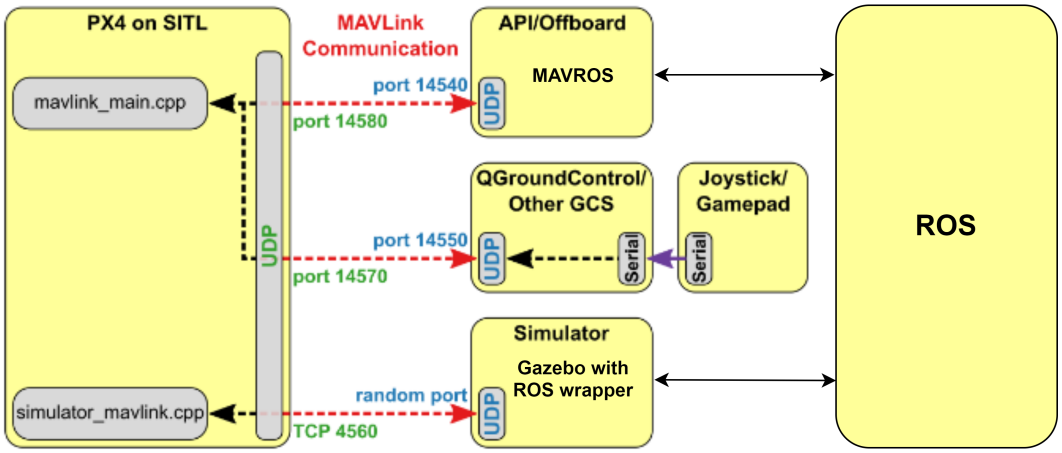


Figure 3.4: PX4 Simulation logical unit overview using Gazebo with ROS wrapper

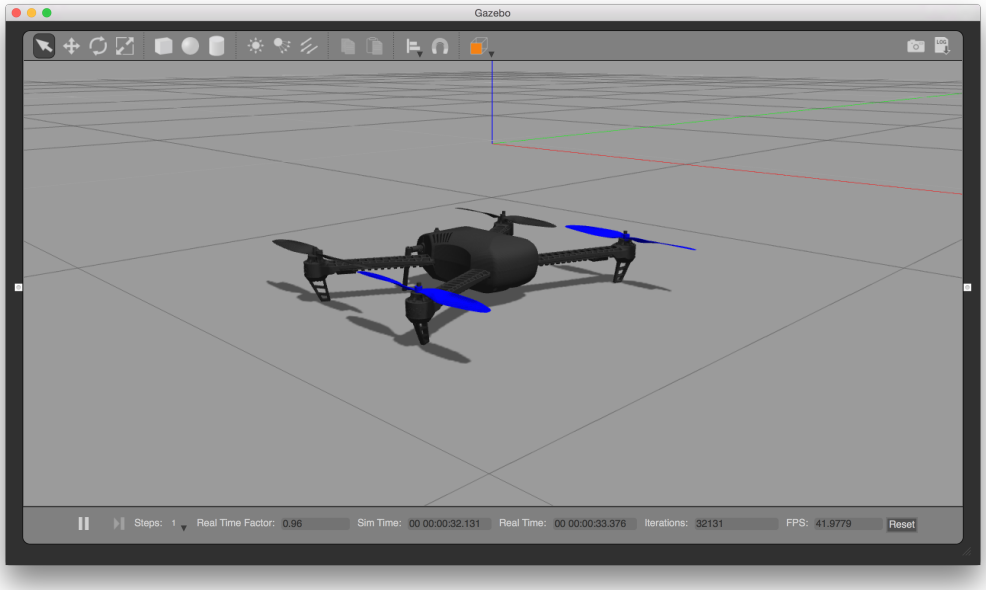


Figure 3.5: Iris drone in gazebo

3.4 QGroundControl

QGroundControl(QGC) is an open-source ground control station software, developed as a part of Dronecode. It provides full flight control and vehicle setup for PX4 and ArduPilot powered vehicles and for any flight controller that uses MAVLink communication protocol.

It provides easy and straightforward usage for beginners, while delivering high end features for experienced users too.

QGC is mainly used for flight controller setup and configuration. When setting up a PX4 autopilot based flight controller, the controller needs to know the type of the vehicle and the frame that is used. Besides selecting vehicle type, internal sensors need to be calibrated and telemetry messages need to be configured. Low level parameters can also be adjusted and obstacle avoidance can also be setup here, as seen in the product of Terabee in 2.3.1, but even low level flight controller parameters can be changed.

QGC can also be used for mission planning for autonomous flight and gives instant feedback of the progress of a mission by visualizing the drone on a map and the most important sensor data.

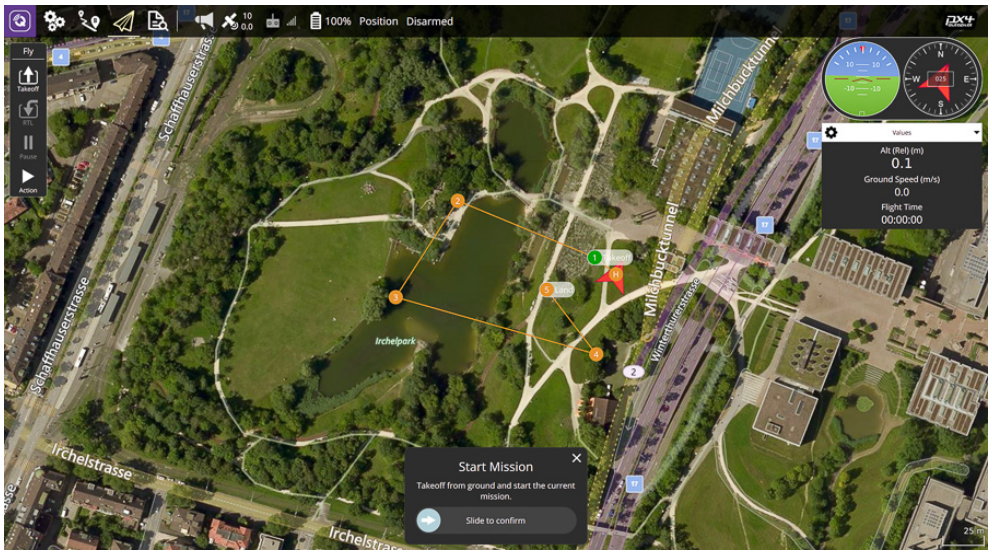


Figure 3.6: QGroundControl main screen[14]

3.5 VL53L1X LIDAR sensor

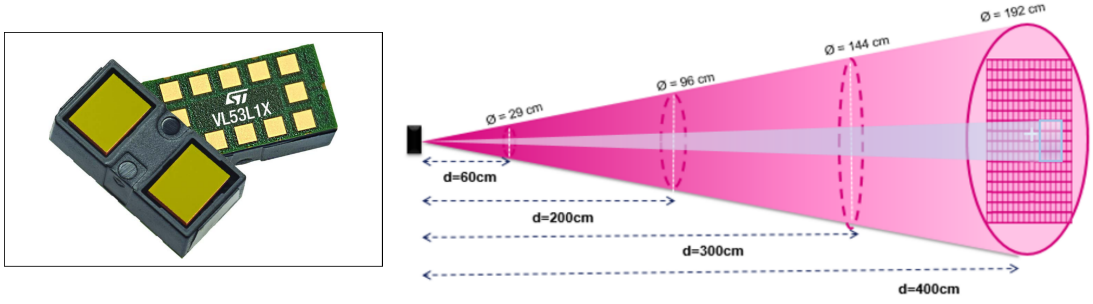


Figure 3.7: VL53L1X sensor [21] and Region of Interest [20]

VL53L1X is described as a long distance ranging Time-of-Flight sensor by STMicroelectronics [21]. The size of the module is 4.9x2.5x1.56mm, it emits 940nm invisible laser and uses a SPAD(single photon avalanche diode) receiving array with a field of view of 27°. The module runs all digital signal processing on a built in low power microcontroller that frees up the host processor from these tasks. The sensor according to the datasheet is ca-

pable of 50Hz ranging frequency and up to 400cm distance measurements. The size of the laser receiving array can be programmatically changed by setting region-of-interest(ROI) size. This way the sensor provides multizone operation and a higher resolution, than by using the whole SPAD array.

I chose to use this sensor, because of its small size, availability in the university laboratory, low hardware infrastructure needs and previous experience.

Chapter 4

System design plan

In this chapter I'm introducing the planned steps towards building the previously described system. Due to the COVID-19 outbreak I have no access to the university laboratory when writing this thesis and therefore the planned system cannot be built physically.

4.1 VL53L1X measurements

In order to simulate the chosen VL53L1X sensor accurately, experience needs to be gained using the device. There are several factors that affect the quality of the map produced by the SLAM algorithm. Choosing the right settings for the LIDAR sensor therefore is a crucial part that cannot be determined independently from the SLAM algorithm. At this point it is unsure if lower resolution and lower distance but higher sampling rate is beneficial for SLAM algorithm or the other way around. My approach is to first explore the capabilities of the sensor in all relevant operating modes, measure the ranging performance like update rate, maximum ranging distance and noise level of the measurements. Later on the optimal operating mode can be chosen based on the SLAM performance.

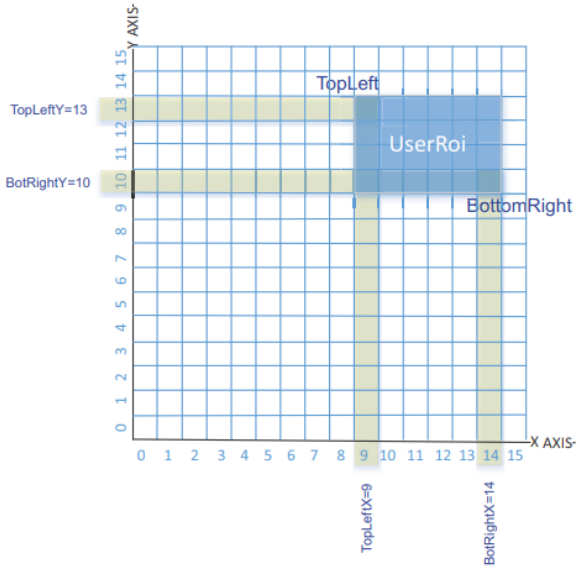


Figure 4.1: VL53L1X Region of Interest setting [20]

The sensor has three parameters, that affect the ranging performance and require tuning for each application. These three parameters are the distance preset, the timing budget and the Region of Interest(ROI). The device has three different distance presets for short, medium and long distance ranging. The second parameter is the timing budget, that basically sets how much time is available for the sensor to complete a range. Bigger timing budget means more accurate measurement, but on the downside it lowers the sampling rate.

As described in section 3.5 the sensor has a configurable SPAD array that can be used to narrow down the field of view to a specific region. This region in the datasheet is called Region of Interest and will be further referenced to as ROI. By lowering the size of the ROI and changing its position, scanning can be achieved resulting in a higher resolution up to 4x4. Increased resolution means smaller receiving array that lowers sensitivity and maximum distance that can be measured. Scanning also requires more time because each ROI can take time up to the timing budget.

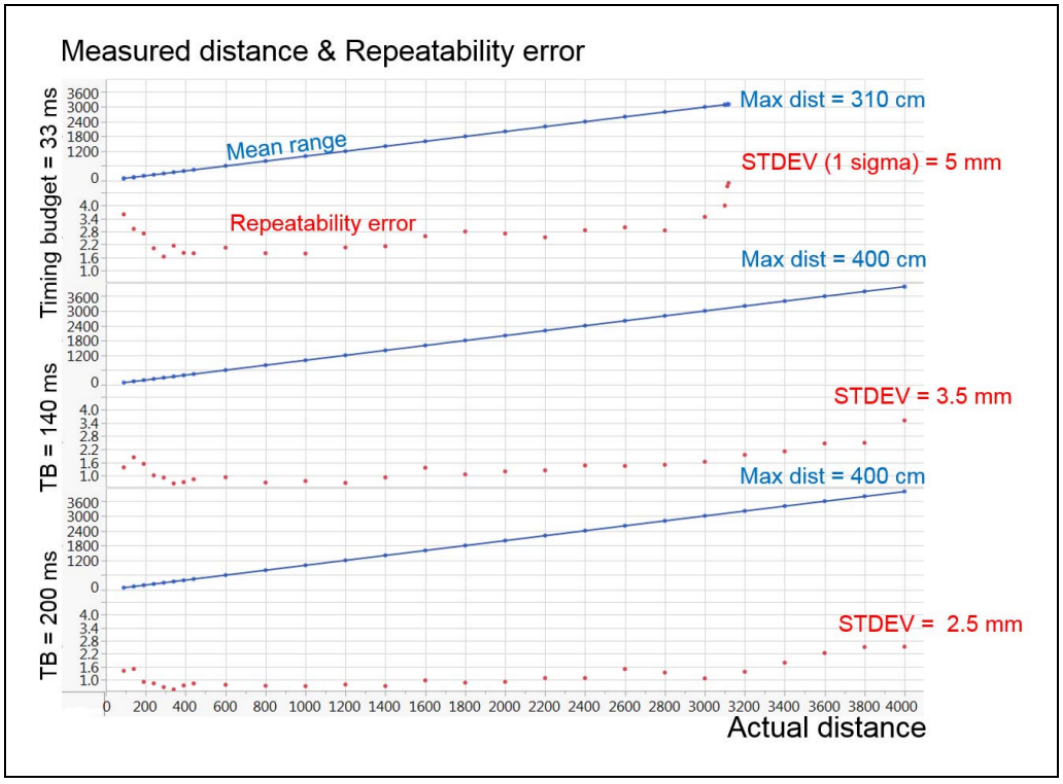


Figure 4.2: VL53L1X maximum distance, error vs timing budget [21]

4.1.1 Selected operating modes

To have a better overview how these parameters affect the ranging performance, a number of operating modes have been chosen for testing. Each operating mode is a group of three parameters that are the distance preset, timing budget and the resolution. The performance of an operating mode is measured by the update rate, maximum ranging distance and standard deviation on a given distance.

According to the VL53L1X datasheet, the 4m maximum distance can only be achieved in long preset, 3m in medium and 1.3m in short. The datasheet contains a plot for

maximum distance, standard deviation vs timing budget in long ranging preset with ROI set to maximum size. The timing budgets in the datasheet are 33ms, 140ms and 200ms, therefore I have selected these timings for test measurements.

As an additional setup an operating mode with the highest update rate has been selected, because update rate is expected to improve SLAM performance. This can be achieved by using an inter-measurement period of 20ms and timing budget of 18.5ms, but only in short range preset and ROI size set to maximum. The estimated highest update rate using this operating mode is 50Hz.

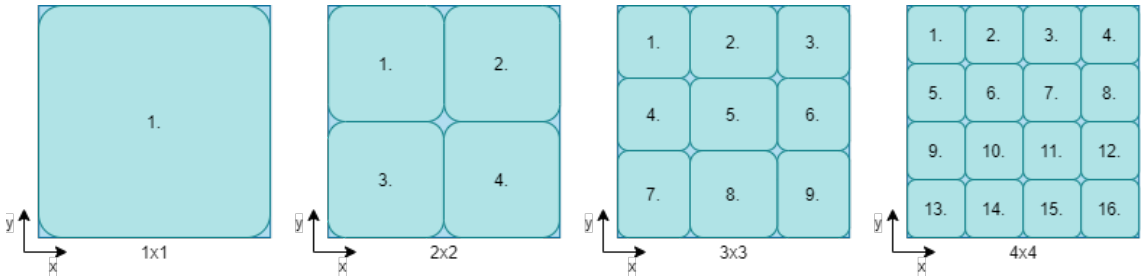


Figure 4.3: VL53L1X ROI setups

As seen on figure 4.2 that the sensor provides measurements with a standard deviation under 5mm, but it's not mentioned how the accuracy changes by lowering the ROI size. To investigate the performance of ranging measurements using lower ROI size, four Region of Interest scan patterns have been selected. The regions are selected evenly on the SPAD array to provide 1x1, 2x2, 3x3, 4x4 resolution output. These setups have been selected, because this way the regions do not overlap and are evenly distributed. The sensor would be capable of more complex patterns or even changing the resolution between scans, but these setups should provide a good understanding of the sensor and the effects on the SLAM performance.

Timing budget	Resolution	Range preset	Distance
18.5 ms	1x1, 2x2, 3x3, 4x4	Short	1 m
33 ms	1x1, 2x2, 3x3, 4x4	Medium, Long	1 m
140 ms	1x1, 2x2, 3x3, 4x4	Medium, Long	2.5 m
140 ms	1x1, 2x2, 3x3, 4x4	Medium, Long	3 m
200 ms	1x1, 2x2, 3x3, 4x4	Long	3.5 m

Table 4.1: Selected operating modes

In table 4.1 a summary can be found of selected operating modes and a distance to be tested on. These settings are planned to be evaluated incrementally starting with a timing budget of 18.5ms and incrementally moving up to 200ms. With smaller ROI size the maximum distance is degrading, therefore it is expected that some combinations will not be working properly. For example medium preset on 3 meters is expected to be working only on resolution 1x1, but likely to fail on higher resolutions.

4.2 LIDAR layout

As mentioned before, multiple aspects need to be taken into account when placing LIDAR sensors on a quadcopter with the purpose of using the measurements for SLAM. After

gaining experience of the performance of each operation mode of the sensor, the next step is to determine the optimal layout and number of sensors to be used.

In Cartographer 3D SLAM is basically an extension of their 2D solution, but significantly more complex and harder to tune for optimal performance. With this in mind to better understand the effects of layout it is reasonable to first reduce complexity and test layouts for 2D mapping. Based on the experience gained during this experiment it can be better estimated how many sensors and in what layout are needed for 3D SLAM. A possible outcome is that VL53L1X is not suitable at all for this purpose and a different sensor needs to be used, with more suitable parameters. If mapping and localization accuracy is poor in 2D, it is expected to be worse in 3D.

4.2.1 Layout design for 2D SLAM

Without experience with Cartographer or other SLAM algorithms in general, it is not advisable to give an estimation for the number of sensors to be used or the layout of the sensors at this point. My approach is to start with high number of sensors in a reasonable layout, make it work with Cartographer and then incrementally decrease the number of sensors until the result is still acceptable.

For 2D SLAM I chose to place LIDARs evenly to cover the whole 360° range of the horizontal plane, without overlapping fields of the sensors. Due to the high sampling rate and parallel ranging operations, overlapping fields could would have a higher probability of crosstalk between neighbors and would produce additional noise in real-world applications, that is better to avoid. Altogether 13 sensors are needed to cover 97.5% of the plane, with 27.69° in between, leaving lass 1° of uncovered zone in between fields. 13 sensors can produce 13 points per scan with ROI resolution set to 1x1 and up to 208 points with the resolution set to 4x4.

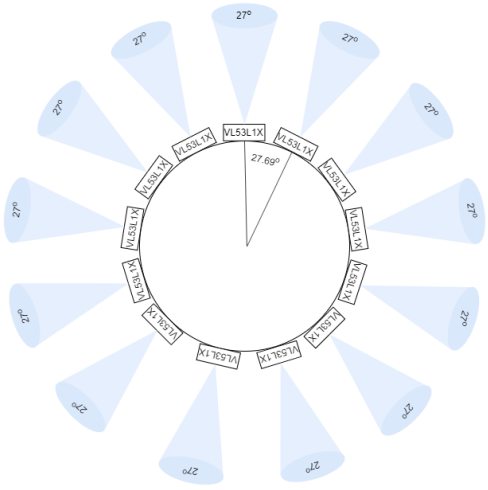


Figure 4.4: Layout of 13 LIDAR sensors

In this layout the scanning on the horizontal plane is done purely by the evenly placed the sensors around the vertical axis and doesn't take advantage of the movement of the drone during flights, especially turns around the yaw axis. After evaluating the performance of this layout, the number of sensors will be incrementally decreased, by removing a number of sensors at a time. The reduced set of sensors will always be placed evenly, because

multicopters can fly in any direction. The flying style is highly dependent on the pilot, so it is best to scan the plane evenly without assuming any dominant flying direction.

The planned bottom limit for the number of sensors is what can be seen on the solution of Bitcraze, the Multi-ranger deck[11] that uses 4 sensors on the horizontal plane facing forward, backwards, left and right. In their demonstration video, the map of a simple room is built in 20 seconds with a ranging frequency of about 10Hz.

4.2.2 Layout design for 3D SLAM

The layout for 3D SLAM depends on the experience gained during the evaluation of 2D SLAM, but has more freedom in sensor placement. Because multicopters change their pitch and roll angles to move horizontally, this movement can be used for scanning of the vehicle's environment.

Just like in case of 2D SLAM, in the first layout sensors are placed evenly to cover the whole sphere, without overlapping fields. In the following iterations a number of sensors will be removed until the minimum number of sensors is found that still provide a reliable SLAM performance.

4.3 Data collection in Gazebo Simulator

To compare the performance SLAM setups using different layouts and operating modes, I have decided to work on the same set of data in each iteration. ROS offers a great tool to record data that is being transferred in ROS topics into a format that they call rosbag. This tool subscribes to the requested topics and saves all ongoing data into a file with .bag extension. The same tool can be used to play back previously recorded data and the tool will publish messages at the same rate as it was published originally. ROS also provides a C++ and a Python application programming interface for rosbags that can be used to read or write the contents of a bag file.

My plan is to place a high resolution 360° LIDAR sensor that covers the whole sphere on a simulated quadcopter, fly it around in a building while recording the necessary topics into a rosbag. The ranges in this rosbag can be then filtered in a way, that its parameters match the selected operation mode and layout. This way the performance of the SLAM using different layouts and sensor operating modes can be compared.

4.3.1 Development environment in Visual Studio Code

After building the repository using the guide on PX4 website, I have learned that new models, launch files and world files need to be placed in a subdirectory of the repository that is only generated at build. This also means that if the project is rebuilt, the newly added files might be overwritten. To avoid overwriting on builds, I have started working on these files in a different folder and copy it to their final destination just before starting the simulation.

Cartographer also uses the same mechanism, all files need to be copied to subfolders in its install location. For editing these files I use Visual Studio Code and to make copying easier, I have decided to add a build task in my editor, that copies PX4 files and cartographer files to the corresponding folder. This makes the development much easier and less troublesome.

The project folder structure can be seen on figure 4.5. Measurements folder will contain the output of each SLAM configuration and simulation folder is for ongoing development. Under simulation, the bag folder contains recorded and filtered bag files. All cartographer specific files are in subfolders of cartographer_files folder. Catkin_ws folder holds the Cartographer_ros repository and its dependencies. Firmware is for storing PX4 repository and firmware_file contains custom files to be copied to the PX4 build directory. QGC stands for QGroundControl, that is not necessary for this simulation, but can be useful in some cases. Lastly scripts folder is for scripts that are used for rosbag filtering, trajectory extraction or other purposes.

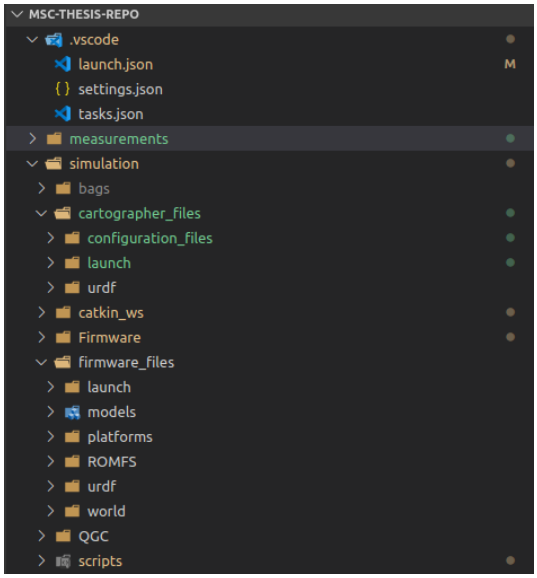


Figure 4.5: Project folder structure in Visual Studio Code

4.3.2 Adding LIDAR sensor to a quadcopter model

Instead of overwriting files generated during the build procedure of PX4 repository, I have decided to create a custom quadcopter model, that extends the already available Iris drone. Any sensor that is supported Gazebo can be added to it, by editing the model descriptor SDF file.

The equivalent of a LIDAR sensor in Gazebo is called a Ray sensor, that is highly customizable using parameters in the SDF model file. The goal is that the rays of the sensor cover the whole sphere around the quadcopter. The rays of the LIDAR are blocked by the simulated drone so a single sensor cannot be used to cover the whole sphere. I have placed two sensors on the drone, one on the top and one on the bottom as close to the body of the vehicle as possible.

Since the VL53L1X sensors in real-world application would be measuring simultaneously, the simulated LIDARs are also set to scan every angle at once and send all ranges in a single message. The maximum range is set to 4 meters and the update rate is at 50Hz. Gazebo is able to simulate predefined measurement noise, but at first I have decided not to add additional noise. During data filtering these measured point clouds will be downsampled by averaging on regions and noise can be simulated in addition at this stage. In each scan 128 horizontal and 64 vertical points are sampled per sensor, resulting in a total number of 128*128 measurement points and angular resolution of 2.8° both vertical and horizontal.

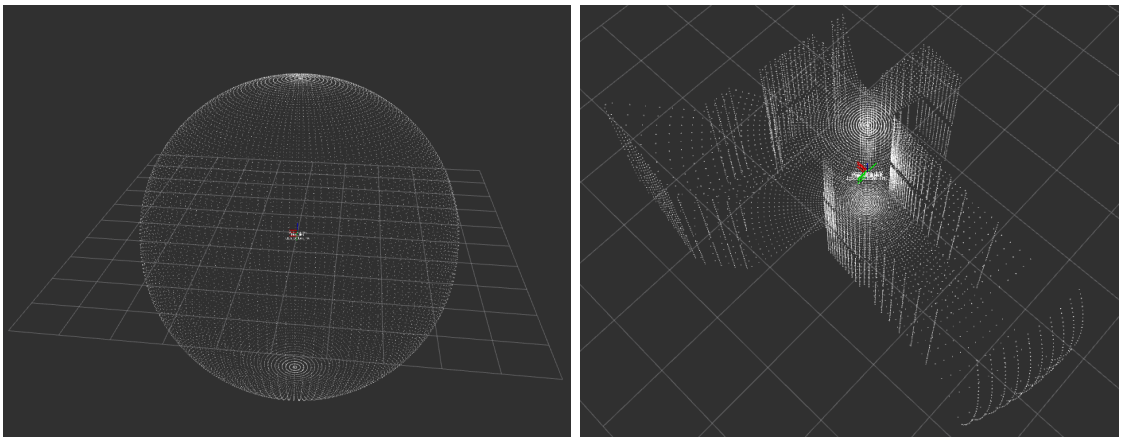


Figure 4.6: LIDAR output visualization in rviz

The maximum update rate of each simulated LIDAR is 30Hz, because Ray sensor uses the CPU for computations instead of the GPU. It would be possible to use an alternative called GPU Ray, but I decided not to use it, because of its output message type. Cartographer expects ranges aggregated into PointCloud2 messages, while Ray sensor's output is PointCloud and the GPU alternative has an output in LaserScan format. PointCloud2 is a newer version of PointCloud and conversion is relatively easy. If 30Hz update rate is proved to be too low during future steps, it is possible to use GPU Ray instead, that requires modifications in the filtering algorithm.

4.3.3 Building model

Using Gazebo Building Editor I have created a building model to be mapped. The building is minimalistic and not furnished, but contains enough details like columns and curves to allow Cartographer to match scans more efficiently. For example SLAM is not expected not work properly on a corridor without any details in range, because it needs a reference points that can be used for scan matching. If each scan looks exactly the same the algorithm will lose track resulting in bad map quality. This is increasingly in focus in this case, because the maximum distance of the VL53L1X sensor to be used is 4m and the sensor has a wide field of view that hides details of faraway objects.

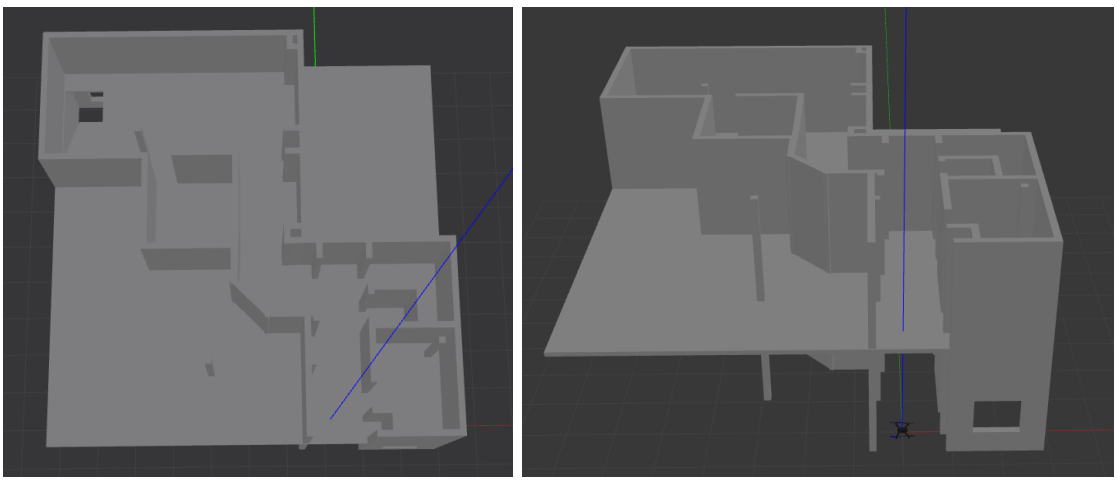


Figure 4.7: Two story building in Gazebo

The second floor of the designed building is seen on figure 4.7. The ground level has exactly the same floor map, except it also has a staircase in the top left room, windows and what is important, it has a ceiling. Ceiling is expected to play an important role in 3D slam, adding an extra reference point for vertical positioning. Using this building it is possible in the future to test the efficiency of 3D SLAM by flying up the stairs and around the second floor.

4.3.4 Remote controller

After following the previous steps and starting the simulation, the quadcopter with the LIDAR sensor is being spawned at the entrance of the building, but it needs to be controlled to fly around the house. I chose to use a virtual controller that has been developed by a student at the laboratory of my department. The controller uses MAVROS topics to send and receive commands to and from the drone. Using this program, the quadcopter can be flown by using the PC keyboard.

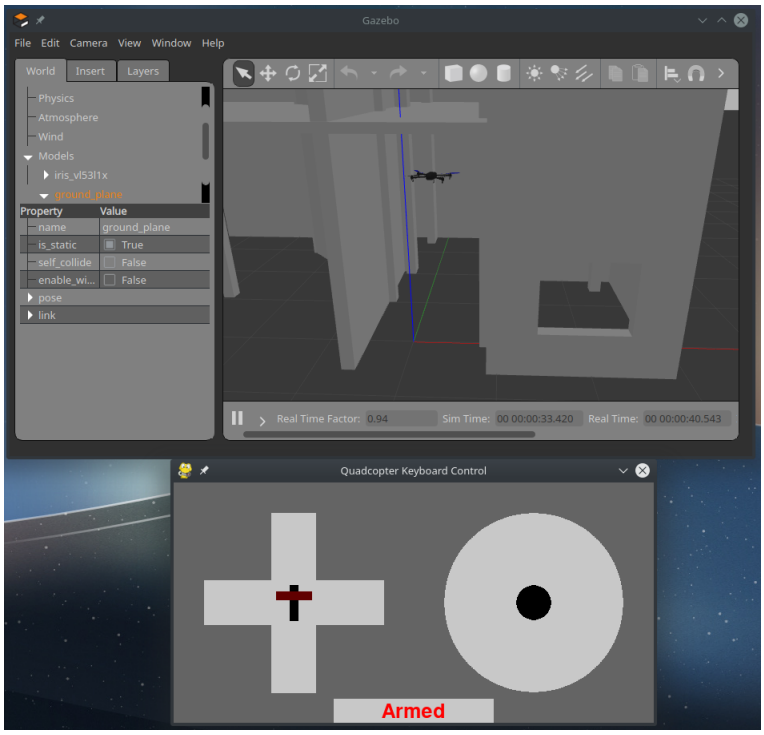


Figure 4.8: Virtual remote controller for the simulated drone

For evaluation of SLAM the recorded flight time is kept low, so the data filtering and Cartographer can run faster. About 2 minutes of data is enough for comparing the performance of SLAM configurations and further flights can take place to further test the evaluated configurations.

4.3.5 Rosbag filtering

Filtering is the process of reading rosbag messages, filtering LIDAR ranges to show behavior of the selected VL53L1X settings and writing the filtered data to a new bag file. Due to the high data rate, rosbags tend to take up lots of disk space. To preserve disk space and increase processing speed, only necessary topics described in 4.4 are being written to the filtered file.

The recorded rosbag contains LIDAR measurements on two topics, the top sensor covers the top hemisphere, the bottom one covers the bottom hemisphere. To simulate a VL53L1X sensor behavior, a field of view with a selected opening angle is cut out of the point cloud. Then an average of ranges inside this view is calculated. The averaged distances are then maximized to a set maximum distance threshold and the sampling time is reduced according to the selected device settings. For simulating higher resolutions, the same methodology can be used, by using multiple field of views with smaller opening angles.

The raw measurements read from the collected rosbag are PointCloud type, but Cartographer requires PointCloud2 messages. At the end, the filtered points are converted to PointCloud2 and written to the filtered bag file.

4.3.5.1 Update rate

LIDAR messages are recorded in a rosbag at a certain rate and it needs to be downsampled to match the desired update rate. Every PointCloud message has a header field that contains the current time at publish. During filtering each message is read from an input bag in order as they were published. A PointCloud message is only considered for filtering, if the time difference between the last written and the current message is greater than the defined update rate. Otherwise the message is discarded.

Using this procedure the update rate will not match exactly the desired rate, but keeps data integrity and order with other messages.

4.3.5.2 Field of View

Each PointCloud message contains a complete scan from either the top or bottom sensor, that scan the top or bottom hemisphere accordingly. As the message type suggests, these ranges are represented by point clouds, a list of 3 dimensional point coordinates in reference to the sensor coordinate system. During angle filtering this point cloud is being downsampled to lower resolution that simulates the sensors placed on the quadcopter.

VL53L1X sensor has a cone shaped field of view in which the reflected signal is detected and converted to a range measurement. To match this behavior, a cone shaped subset of the points are selected for further processing. A point is selected if the angle between the point and the center of the field of view is less than a predefined θ angle threshold.

The angle between two vectors, or in this case between a point from the point cloud and the center vector can be calculated using the dot product of the two. Equation 4.1 and 4.2 are two forms of dot product. These are used to derive equation 4.3 that is used to calculate the angle. In this case vectors $\vec{u}$ and $\vec{v}$ are the center vector and a selected point from the point cloud.

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \, \|\vec{v}\| \, cos(\alpha) \tag{4.1}$$

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y + u_z v_z \tag{4.2}$$

$$\alpha = cos^{-1} \frac{u_x v_x + u_y v_y + u_z v_z}{\|\vec{u}\| \, \|\vec{v}\|} \tag{4.3}$$

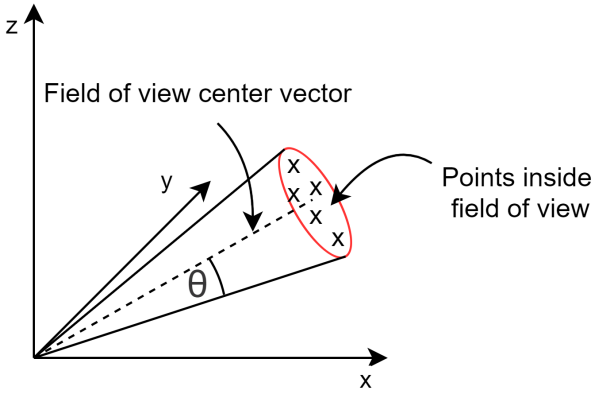


Figure 4.9: Point cloud angle filter

$$\|\vec{u}\| = \sqrt{u_x^2 + u_y^2 + u_z^2}, \|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{4.4}$$

For every point inside the point cloud this angle is calculated and points that have an angle less than the predefined θ are added to the list of points inside the field of view. As a final step average of the selected points is calculated. Points once again are represented by coordinates, therefore average is calculated by averaging the x, y and z coordinates resulting in the coordinates of a single point.

The steps described above are for the filtering a single sensor, but the same steps apply for multiple sensors. Higher resolutions are constructed by multiple field of views, with a lower angle threshold of θ. Figure 4.10 shows an example for field of view selection for a resolution of 2x2. Field of view center line and angle threshold are not drawn, to keep the figure simple.

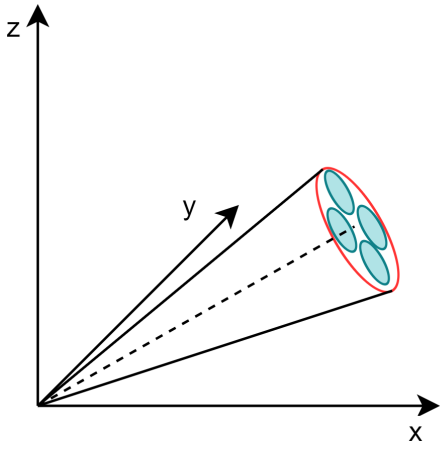


Figure 4.10: Representation of 2x2 LIDAR resolution filtering

Points inside a PointCloud message always take the same order, therefore it's enough to filter for angles only once and use the indexes of these points from that on. This way lots of processing capacity can be saved and it proved to be much more efficient on high number of simulated sensors.

4.3.5.3 Distance filter

Angle filter returns a single point for each predefined field of view. For each sensor it can mean a single point or up to 16 points in case of a resolution of 4x4. As the second step, distance filter receives these points and calculates the distance of each point from the origin, basically the same as calculation of length of a vector. If the distance is greater than the defined maximum distance, the length of the vector is shortened to the maximum distance. On figure 4.11 p_o represents the point and p_s is the shortened vector.

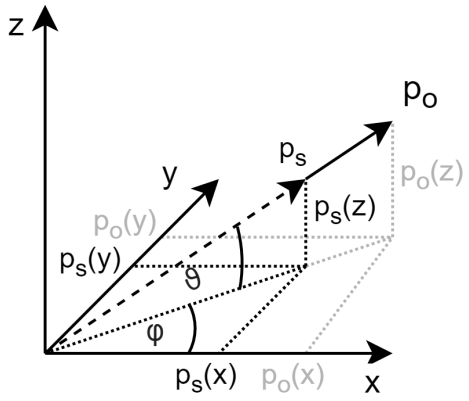


Figure 4.11: Point cloud distance filter

$$\|\vec{p_o}\| = \sqrt{p_o^{x\,2} + p_o^{y\,2} + p_o^{z\,2}} \tag{4.5}$$

To shorten the length of a vector, all three coordinates need to be recalculated in such way, that the attitude of the vector does not change. To do so, vertical ϑ and horizontal φ angles for vector p_o are calculated using equation 4.6 and 4.7.

$$\vartheta = tan\frac{p_o^z}{\sqrt{p_o^{x\,2} + p_o^{y\,2}}} \tag{4.6}$$

$$\varphi = tan\frac{p_o^y}{p_o^x} \tag{4.7}$$

Using the vertical and horizontal angles, the coordinates of the shortened p_s is calculated using trigonometric equations 4.8, 4.9 and 4.10.

$$p_s^x = d_{max} \cdot cos\vartheta \cdot cos\varphi \tag{4.8}$$

$$p_s^y = d_{max} \cdot cos\vartheta \cdot sin\varphi \tag{4.9}$$

$$p_s^z = d_{max} \cdot sin\vartheta \tag{4.10}$$

4.4 Cartographer environment

Cartographer can be installed on Ubuntu operating system easily by using the apt-get tool, but this method would overwrite the system installed Protobuf with a newer version. Protobuf is used by Gazebo and it requires a certain version, replacing it with another version causes incompatibility and simulations cannot be started. For this reason I have installed Cartographer and Cartographer_ros packages inside a Catkin workspace and set them to use locally installed Protobuf inside the same workspace. Using these settings, Cartographer and Gazebo can be used on the same PC.

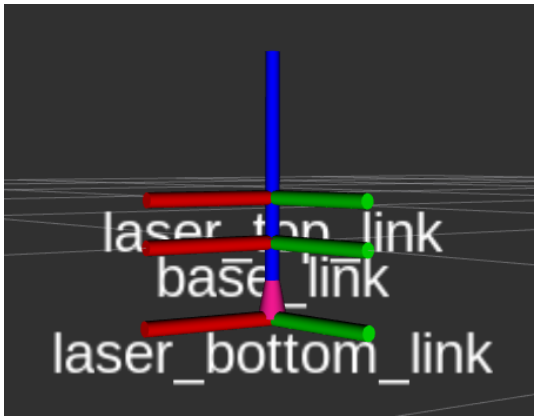


Figure 4.12: Cartographer top, bottom LIDAR and IMU link relations

To attach the LIDAR sensors onto the simulated quadcopter in Gazebo, two links have been introduced that connect to the base_link of the drone, one on the top and one on the bottom. The LIDAR scanner messages are sent in reference to these links and therefore Cartographer needs to know the transformation from the sensor frames to the base_link frame. Two official packages are required called robot_state_publisher and joint_state_publisher. These take a robot_description parameter that can be set using an urdf file and they start publishing the link and joint relations. Besides the LIDAR holder links, a link for IMU transformation is also need to be added. The simulated IMU in the PX4 repository is used and it is located in the origin of base_link. Laser_top_link is located 6cm up, laser_bottom_link is located 10cm down on the z axis in reference to base_link as seen on 4.12.

While running Cartographer on filtered data, the following tf tree is built as on 4.13. Transformations between map, odom and base_link are published by Cartographer and the rest is published from the created urdf file. This tf tree matches the recommendation from the Cartographer documentation website [16].

IMU input for 3D mapping is a must and optional for 2D mapping. Cartographer waits for IMU to be supplied on topic /imu. PX4 simulation publishes IMU data to /mavros/imu/-data, but this topic can be remapped to /imu using the launch file without the need of any extra nodes. Filtered range messages with PointCloud2 type can be accessed on topics /points2_1 and /points2_2. List of nodes and interactions via topics can be seen on figure.

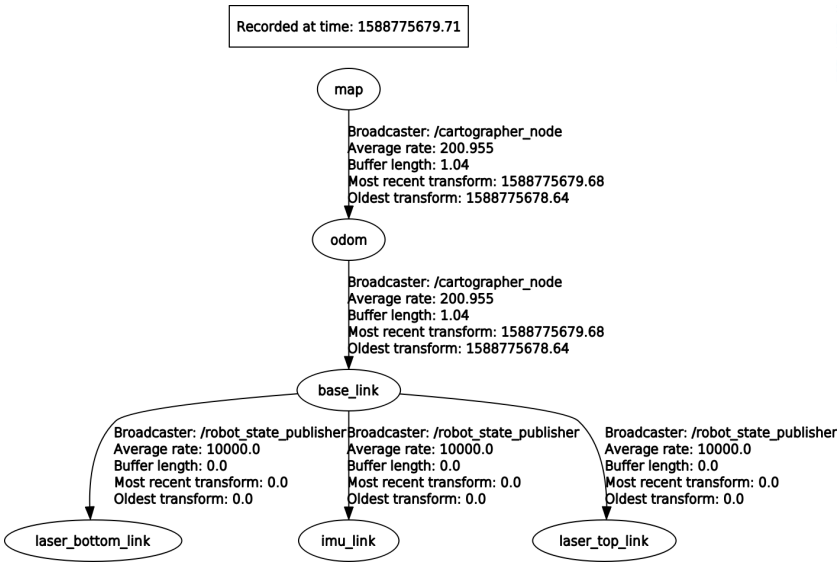


Figure 4.13: Cartographer tf tree

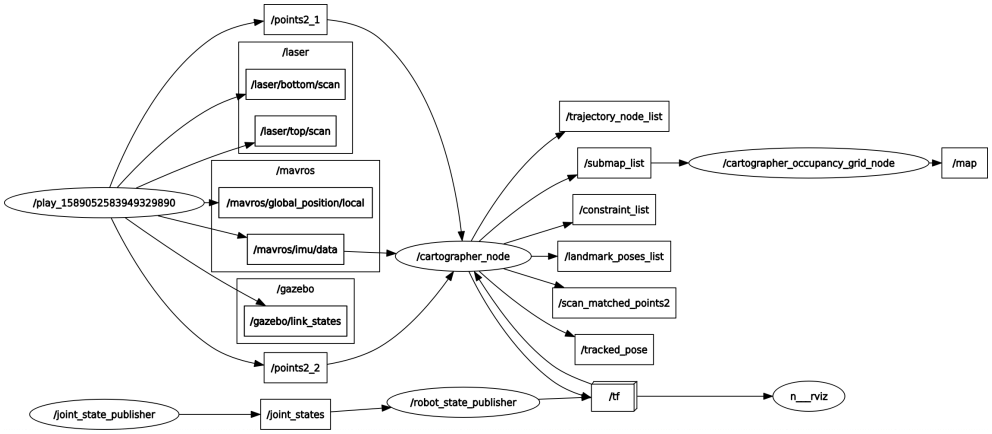


Figure 4.14: Cartographer nodes and topics

4.4.1 Configuration and tuning

Cartographer can be configured using a lua configuration file. Multiple range measurement types are supported, but the filtered rosbag contains PointCloud2 messages on two topics. The num_point_clouds parameter is set to 2 to match the two topics coming from the top and bottom sensors.

Cartographer applies a bandpass filter on the incoming range messages to filter out irrelevant measurements. In every case max_range parameter needs to be set a bit lower, than the maximum distance parameter used for filtering. This is important, because the algorithm inserts a range with a distance under max_range as a hit, but inserts anything above as a miss. Wrongly setting this parameter results in adding all range measurements as hits and no misses will be inserted that will confuse cartographer.

Each sensor setup require the SLAM algorithm to be tuned and therefore a new configuration file is created for every setup being tested. Tuning starts with setting POSE_GRAPH.optimize_every_n_nodes to 0 to disable global SLAM. Good and clear submaps are needed for global SLAM to find loop closures. Submaps are the building blocks of maps. On small maps like the building I chose for LIDAR setup evaluation,

global SLAM is not making much difference, but when tuning submaps it can cause unwanted artifacts that are hard to differentiate from submap errors.

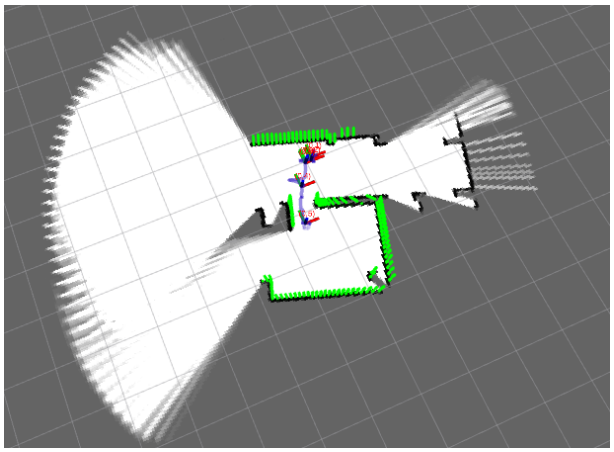


Figure 4.15: Building of submaps visualized in rviz

Typically a small number of parameters need to be tuned for submaps to be working. These are the following:

- TRAJECTORY_BUILDER_nD.ceres_scan_matcher.translation_weight
- TRAJECTORY_BUILDER_nD.ceres_scan_matcher.rotation_weight
- TRAJECTORY_BUILDER_nD.ceres_scan_matcher.occupied_space_weight
- submaps.num_range_data

The first parameter sets how high the cost is for cartographer to move the a scan. In my experience, it needs to be set low when used on a drone, because it changes place quickly. The second parameter sets how much trust should be put on the IMU data. The lower this parameter is set, the easier it is to rotate a scan. Occupied space weight is sets the trust in the LIDAR sensor. I haven't found this parameter particularly useful. Changing the submap size, by num_range data parameter is important, because by changing the update rate, different number of range measurements are needed, while the submap size should not change.

4.4.2 Trajectory extraction

Evaluating the mapping quality is complex relative to evaluating the location tracking performance of a SLAM setup. Gazebo provides a topic where all link states are published and this transformation can be used as ground truth and trajectory extracted from SLAM can be compared to it.

TODO: Wrong configuration picture

4.5 Plans for hardware implementation

On a real-life drone, the communication of LIDAR sensors and measurement forwarding to ROS topics need to be solved. There needs to be a device that communicates with the VL53L1X LIDAR sensors using I2C protocol and forwards messages to a ground station that handles data collection and processing.

The data collector device needs to be fast enough to keep up with the pace of 10-20 sensors, each with a maximum of 50Hz update rate. The distance data from VL53L1X sensor consists of 12bytes, so by designing for 20 sensors with 50Hz update rate generate about 12kb per second data rate. This load is calculated using the effective data size, but on the I2C bus other configuration messages need to be sent, which means even higher load.

To evaluate the SLAM algorithms on real-world measurements, data needs to be collected. For evaluation it is not necessary to send data in real-time on wireless network, but saving data on an SD card serves this purpose. Logging on SD card has low complexity and high speed.

PX4 is an open-source firmware, it seems an easy choice to use the Pixhawk 4 autopilot board as a data collector device. It has an SD card slot for logging and I2C connectors available to communicate with the LIDAR sensors and even comes with an I2C splitter board to connect more sensors. The microcontroller inside Pixhawk 4 is powerful, it has an ARM Cortex-M7 core running on 216MHz. It is used to run highly timing sensitive control loops to produce actuator values for stable flights, besides many other tasks. If these control loops are delayed by other processes, that can cause instable flight or even crash.

Because of lack of deep understanding of PX4 firmware, to make sure that timing of control loops are untouched, I decided to design a standalone data collector, using a dedicated microcontroller that handles I2C communication with the LIDAR sensors and writes data to an SD card using SPI protocol.

4.5.1 Standalone data collector design

In I2C protocol every device needs to have a 7 bit address, that needs to be unique on every I2C line. VL53L1X sensor has a default address of 0x29 when the sensor is booted, but it can be changed by using I2C commands. Address conflicts need to be avoided by having exactly 1 sensor active per channel. It is hard to find a microcontroller that has 20 I2C buses, one for each sensor, so multiple sensors need to be placed on the same bus. By having multiple sensors on the same channel, the sensors need to be released from reset one by one to change their addresses, this way address conflicts can be avoided. VL53L1X can be kept in reset by pulling its shutdown pin to ground and activated by bringing it to supply voltage.

The sensors have altogether 4 lines that need to be driven: 2 I2C, 1 shutdown and 1 interrupt pin. In the simplest case, by connecting each wire to the microcontroller, 80 wires would be needed for 20 devices and would take 42 pins of the microcontroller. Although it is possible to find a microcontroller that has enough pins and connect all sensors one by one to the same I2C bus, but by grouping them into groups of 5 and connecting each group to different I2C buses, a significant improvement can be achieved on the number of connections.

Sensors on different I2C buses can have the same address, therefore the same 5 addresses can be used in each group. Shutdown pin of sensors sharing the same addresses can be common, this way number of wires needed for shutdown is reduced from 20 to 5.

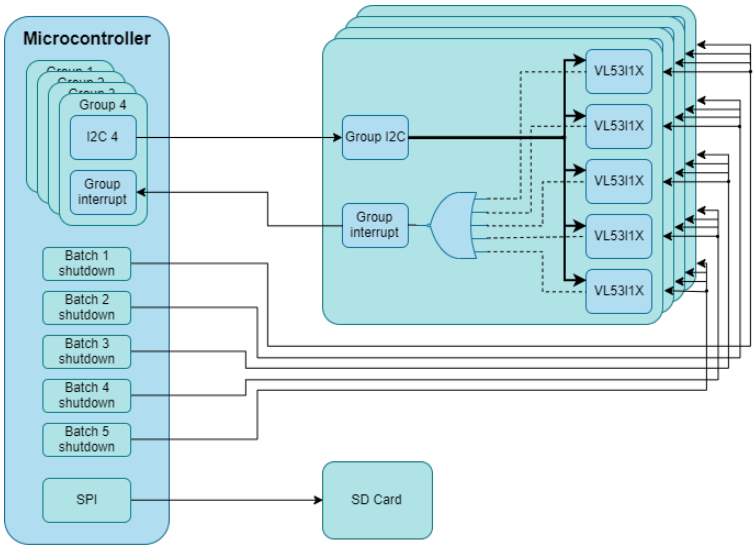


Figure 4.16: Design of data collector

Handling 20 interrupt lines can be overwhelming for any CPU, while bringing little or no extra benefit. Using only one interrupt line per group is enough to signal the microcontroller that measurements are ready to be read out in all sensors in that group. Interrupt pins are active-low, so to produce the group interrupt signal a NOR gate needs to be used. When the state of all interrupt pins are low, group interrupt signal will become logical 1 and 0 otherwise.

In the simplified design seen on figure 4.16 only 17 wires are connected to the microcontroller, while no functionality is lost. This solution also makes software development easier and the built system is clearer.

Bibliography

[1] PX4 Drone Autopilot. Firmware repository. `www.github.com/PX4/Firmware`, . Accessed: 2019-11-27.

[2] PX4 Drone Autopilot. Website. `www.px4.io`, . Accessed: 2019-11-28.

[3] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35, 2006.

[4] PX4 Simulation Documentation. Website. `dev.px4.io/v1.9.0/en/simulation`. Accessed: 2019-11-28.

[5] David Droeschel and Sven Behnke. Efficient continuous-time slam for 3d lidar-based online mapping. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9. IEEE, 2018.

[6] David Droeschel, Jörg Stückler, and Sven Behnke. Local multi-resolution representation for 6d motion estimation and mapping with a continuously rotating 3d laser scanner. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5221–5226. IEEE, 2014.

[7] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[8] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European conference on computer vision*, pages 834–849. Springer, 2014.

[9] Jakob Engel, Jörg Stückler, and Daniel Cremers. Large-scale direct slam with stereo cameras. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1935–1942. IEEE, 2015.

[10] Sebastian Hening, Corey A Ippolito, Kalmanje S Krishnakumar, Vahram Stepanyan, and Mircea Teodorescu. 3d lidar slam integration with gps/ins for uavs in urban gps-degraded environments. In *AIAA Information Systems-AIAA Infotech@ Aerospace*, page 0448. 2017.

[11] Bitcraze Crazyflie multiranger deck. Website. `www.bitcraze.io/multi-ranger-deck`. Accessed: 2019-11-30.

[12] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5): 1147–1163, 2015.

[13] NASA Official. Aircraft rotations - body axes. `www.grc.nasa.gov/www/k-12/airplane/rotations.html`. Accessed: 2019-12-02.

[14] QGroundcontrol. User manual. https://docs.qgroundcontrol.com/en/. Accessed: 2019-12-04.

[15] Gazebo Simulator. Website. www.gazebosim.org. Accessed: 2019-11-28.

[16] Cartographer SLAM. Documentation. https://google-cartographer-ros.readthedocs.io/en/latest/algo_walkthrough.html. Accessed: 2019-12-05.

[17] Hauke Strasdat, J. M. M. Montiel, and Andrew J. Davison. Scale drift-aware large scale monocular slam. In *Robotics: Science and Systems*, 2010.

[18] Robot Operating System. Website. www.ros.org. Accessed: 2019-11-27.

[19] Terabee TeraRanger. Terabee teraranger tower evo. www.terabee.com/shop/lidar-tof-multi-directional-arrays/teraranger-tower-evo. Accessed: 2019-11-18.

[20] VL53L1X. Application note. www.st.com/content/ccc/resource/technical/document/application_note/group0/53/ea/8b/72/2d/07/4f/21/DM00516219/files/DM00516219.pdf/jcr:content/translations/en.DM00516219.pdf, . Accessed: 2019-11-30.

[21] VL53L1X. Datasheet. www.st.com/resource/en/datasheet/vl53l1x.pdf, . Accessed: 2019-11-30.